Two Approaches to Portable Macros

Fengyun Liu
EPFL
fengyun.liu@epfl.ch

Eugene Burmako
Twitter, Inc.
eburmako@twitter.com

Abstract

For any programming language that supports macros and has multiple implementations (each with different AST definitions), there is a common problem: *how to make macros that operate on ASTs portable among different compiler implementations*?

Implementing portable macros is especially important for statically typed languages like Scala, as IDE vendors usually have different implementations of the language in order to support rich IDE features. Unportable macros compromise IDE features and degrade programming experience.

We describe two approaches to the portability problem based on two different views on macros: (1) the *tree-based* approach, which views macros as operations on abstract syntax trees, solves the problem by defining *standard abstract syntax trees*; (2) the *syntax-based* approach, which views macros as operations on abstract syntax, solves the problem by defining *standard abstract syntax*. We show that the latter has significant practical advantages, especially in supporting semantic macros that use type information of ASTs to transform user code.

Based on the idea, we implemented a new macro system, *Gestalt*, for the experimental Scala compiler, Dotty. The new implementation solves several long-standing problems of the current Scala macro system and demonstrates advantages over alternative approaches. Our solution has been adopted in the official new Scala macro system.

Keywords macros, Scala, meta-programming, AST, quasiquotes

ACM Reference format:
Fengyun Liu and Eugene Burmako. 2017. Two Approaches to Portable Macros. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, EPFL, July, 2017 (Technical Report)*, 13 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Scala has experimental support for macros since version 2.10 (Burmako 2013). The introduction of macros gives more power to programmers, but also brings more troubles to them. The most noticeable problem is that IDEs for Scala have suboptimal support for macros.

This can be illustrated by the following example. Suppose we have defined an annotation macro `@className` that adds a method `className` to any given class. The code below will be compiled by the standard Scala compiler without any problem:

```
@className class Expr

val exp = new Expr
println(exp.className)
```

The code above can be compiled without problem by the standard Scala compiler, because during compilation, it will run the macro `className` to transform `class Expr` to the following:

```
class Expr { def className = "Expr" }
```

However, when the programmer is writing the code in a popular IDE like IntelliJ IDEA, it will report an error saying that `Expr` does not have a field named `className`. What a poor programming experience!

Currently, macro annotations in Scala are defined is defined in a way that binds tightly to the ASTs (abstract syntax trees) of the standard Scala compiler, an IDE like Intellij with its own language implementation is unable to expand the macro due to the mismatch in AST format. Thus the original code fails to type check in the IDE.

Since Scala is a statically typed language, many Scala programmers rely heavily on IDEs for real-time and interactive type checking. When certain Scala features are unsupported by an IDE, the programming experience is significantly degraded.

We believe that this user experience problem reveals a more fundamental problem of how to implement portable macros. While languages are usually standardized, ASTs are not, and as a result many other macro systems may face the same problem that Scala does.

In this paper we describe two different approaches to implement portable macros, namely the *tree-based approach* and the *syntax-based approach*.

The tree-based approach views macros as operations on abstract syntax trees. It defines *standard abstract syntax trees*, against which macros are defined. During macro expansion, compiler ASTs are converted to the standard ASTs as input to macros. After expansion, the expanded standard ASTs are then converted back to compiler ASTs.

The syntax-based approach views macros as operations on the *abstract syntax* of the language. Instead of defining standard abstract syntax trees, the syntax-based approach defines *standard abstract syntax*, which is represented by abstract methods to construct and inspect the abstract syntax. We call the abstract methods *constructors* and *extractors* respectively. Macros are defined in terms of these abstract

Technical Report, EPFL
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

constructors and extractors. Each compiler has its own implementation of the standard constructors and extractors. During macro expansion, a concrete implementation of constructors and extractors is provided to macros to actually manipulate the abstract syntax on top of the compiler ASTs. No conversions of ASTs happen during macro expansion.

The syntax-based approach makes it easy to access attributes of trees, such as types and symbols, which are essential for macros that transform user code based on type information of ASTs.

Concretely, our contributions are as follows:

- We identified two different approaches to implement portable macros. As far as we know, the syntax-based approach is new for Scala and it's our original contribution. We also share some design principles of extractors and constructors.
- We implemented a macro system for the experimental Scala compiler Dotty. It advances the state-of-the-art of meta-programming in Scala in terms of API friendliness, simplicity, portability, robustness and hygiene. With significant benefits over the tree-based approach, the syntax-based approach has been adopted in the new Scala macro system.

The rest of the paper is organized as follows. We first introduce the problem of AST heterogeneity (Section 2), then discuss the *tree-based* approach and *syntax-based* approach (Section 3, 4) respectively. Section 5 describes the implementation of a macro system `Gestalt` for Dotty. Section 6 reviews related work.

2 The Problem: Heterogeneity of ASTs

"The world is the totality of facts, not of things."

- *Wittgenstein, Tractatus*

Nearly all popular programming languages enjoy multiple implementations. Usually different implementations define different sets of ASTs due to different conceptualization of the syntactical elements or different problems at hand. In the case of Scala, besides the standard compiler, IDEs like IntelliJ have their own (partial) implementation of the language, in order to support rich IDE features.

While programming languages are usually standardized, ASTs are not, nor is it necessary. For an example of the heterogeneity of ASTs, let's look at the example of Scala for-comprehension illustrated by the following code:

```
for (p <- ps) print(p)
for (p <- ps) yield f(p)
```

The second line uses the keyword `yield`, while the first line does not. The first `for` expression is used when we don't care about the return value of the body. The second one is used when the return value of the body is expected.

Dotty, a prospective Scala compiler, uses two data structures to represent the two different kinds of `for`:

```
case class For(ts: List[Tree], body: Tree)
case class ForYield(ts: List[Tree], body: Tree)
```

However, we could imagine another way to represent the same syntax:

```
case class For(ts: List[Tree], body: Tree)
case class Yield(expr: Tree)
```

The former representation defines two different data structures for two kinds of `for` expressions, while the latter defines one data structure for `for` expression, one for `yield` expression. Both representations capture all the syntactic information, thus both are valid. And it is impossible to tell which is better, as they may serve different purposes.

The heterogeneity of ASTs is not a problem in itself. It only becomes a problem when the language supports features that enable programmers to manipulate ASTs, e.g., macros. Typical operations on ASTs are (1) compose new trees; and (2) inspect structures of trees. In such cases, there is an important question to answer: *which implementation are the AST operations defined against*?

The response to the question above depends on one's view about *what macros are.* A natural view is that macros are operations on ASTs. This view leads to the *tree-based* approach. A different view is that macros are operations on abstract syntax of the language, which leads to the *syntax-based* approach.

3 The Tree-Based Approach

If macros are operations on ASTs, to make the operations portable we just need to define a standard set of ASTs. Then, macros are defined in terms of the standard ASTs. Compiler ASTs are converted back and forth to the standard ASTs during macro expansion. This is the basic idea of the *tree-based* approach.

Let's give the standard ASTs the type `meta.Tree`, the ASTs of compiler X the type `x.Tree`, the whole process of macro expansion for the compiler X can be illustrated by the following code:

```
type Macro = meta.Tree => meta.Tree
def toMeta: x.Tree => meta.Tree
def fromMeta: meta.Tree => x.Tree
def expand(macro: Macro, t: x.Tree) =
   fromMeta (macro (toMeta t))
```

As the type `Macro` shows, macros are defined in terms of the standard ASTs. During macro expansion, the compiler executes `toMeta` to convert compiler ASTs to standard ASTs. Then, the compiler executes `macro` to transform the standard ASTs. Finally, the compiler executes `fromMeta` on the resulting ASTs to convert them back to compiler ASTs to continue compilation.

While this approach does solve the portability problem, it has several drawbacks.

(1) *There is no simple way to get type information of trees.* Statically typed languages like Scala usually have type information attached to ASTs, so macros can profit from the type information when transforming user code. Conversion of trees makes it difficult to access type information due to the disparity of ASTs and the heterogeneity of data structure for types. Keeping type information of trees in the round-trip conversion is also an intricate engineering problem.

(2) *Conversion incurs performance penalty.* Imagine that there is a macro that adds a field to any given class definition. Now if there is a class of 500 LOC, by the tree-based approach every node of the huge AST representing the class has to be converted to `meta.Tree` and converted back, despite the fact that only the root of the huge AST is manipulated.

(3) *It's difficult to keep meta-data of trees.* Trees usually carry meta-data with them, such as positions, types, docs and other attachments. Due to the disparity of ASTs, to correctly carry all the meta-data in the round-trip is not a trivial issue.

(4) *Conversion incurs implementation cost.* Not all syntactic constructs are inspected or should be inspected in macros. As the user code may contain any syntactical construct of the language, the tree-based approach has to define data structures for all possible syntactic constructs and handle them in the conversion, no matter such syntactic constructs are ever inspected or not in macros. Syntactic sugars pose a major problem here, as it's difficult to reliably handle some complex sugars, like `for` comprehensions in Scala.

4 The Syntax-Based Approach

4.1 Introduction

As we see from the previous section, most problems of the tree-based approach are caused by conversions between ASTs. But what if we avoid the conversions at all?

The key insight here is that *macros are operations on abstract syntax, instead of abstract syntax trees.* The abstract syntax of a language captures the essence of the concrete syntax by ignoring inessential details like new lines, spaces, comma, braces, parentheses and etc. The ASTs in different compilers are just different representations of the abstract syntax. The abstract syntax of a language is basically determined by the language specification. However, there could be differences at the micro-level, which is the source of heterogeneity of ASTs as we have seen in the case of `For/Yield` and `ForYield/ForDo`. If we can abstract away the difference in micro-level abstract syntax and define macros in terms of the *standard abstract syntax*, then portability of macros will not be an issue. The view leads to the *syntax-based approach.*

The syntax-based approach fixes a *standard abstract syntax*, which are represented by abstract methods to construct and inspect the abstract syntax. We call the abstract methods *constructors* and *extractors* respectively. Macros are defined in terms of these abstract constructors and extractors. The idea can be illustrated with the following code:

```scala
trait Toolbox {
  type Tree

  def Ident: IdentImpl
  trait IdentImpl {
    def apply(name: String): Tree
    def unapply(tree: Tree): Option[String]
  }

  def Tuple: TupleImpl
  trait TupleImpl {
    def apply(args: List[TermTree]): TermTree
    def unapply(tree: Tree): Option[List[TermTree]]
  }
}
```

The macro system defines a trait `Toolbox` which is composed of abstract constructors and extractors. For simplicity, we only show the constructor and extractor for identifiers and tuples. The abstract methods `apply` and `unapply` capitalize on Scala language features to enable programmers to use the name `Ident` as if it's a data structure:

```scala
val id = Ident("x")

tree match {
  case Ident(name) => ...
  case _ => ...
}
```

In Scala, the first `Ident` will be translated to `Ident.apply("x")`, while the second one will be translated to the method call `Ident.unapply(tree)`.

Macros are defined in terms of the standard constructors and extractors [1]:

```scala
def plus(tb: Toolbox)(a: tb.Tree, b: tb.Tree):
    tb.Tree =
{
  import tb._
  q"$a + $b"
}
```

The macro `plus` transforms `plus(2, 3)` to `2 + 3`. In the body of `plus`, it uses quasiquotes instead of constructors and extractors directly. Quasiquotes are just a handy way to use the toolbox. The quasiquote is translated into to the equivalent code: `Infix(a, "+", b)`, where `Infix` is a constructor for infix expressions. The definition is compiler-independent

[1]Scala programmers can write macros in a more friendly way, the code is just for better illustration of the idea.

as the macro is expressed in terms of the *abstract* constructor `Infix`, it doesn't bind to a particular compiler when the macro is defined. In another word, the macros are defined in terms of abstract syntax, instead of abstract syntax trees.

A compiler X provides its own implementation of the constructors and extractors that construct or deconstruct the standard abstract syntax on top of its own AST trees, as the class `XToolbox` shows:

```
class XToolbox extends Toolbox {
  type Tree = x.Tree

  object Ident extends IdentImpl {
    def apply(name: String): Tree = ???
    def unapply(tree: Tree): Option[String] = ???
  }
}
```

During macro expansion, a concrete implementation of the constructors and extractors `XToolbox` is passed to the macros for constructing and deconstructing abstract syntax on compiler ASTs:

```
plus(new XToolbox)(a, b)
```

The input tree to the macro and the resulting tree from the macro are both compiler trees (`x.Tree`), thus no conversion is required.

4.2 Why it works

But how we can be sure that defining a *standard abstract syntax* is possible? This is based on the following observations:

- Each set of ASTs correspond to one abstract syntax.
- Different abstract syntaxes agree at macro-level.

As the name "abstract syntax trees" suggests, each set of ASTs corresponds to one formulation of abstract syntax. Different abstract syntaxes may differ at the micro-level. However, at the macro-level, different abstract syntaxes agree on what syntactic information they can provide. For example, in the case of `For\Yield` and `ForYield\ForDo`, different compilers differ at the micro-level about the abstract syntax for the body of `for`-comprehensions. At the macro-level, if we take `for`-comprehensions as a whole, no matter which abstract syntax a compiler chooses, they can all provide equivalent information about `for`-comprehensions: what are the enumerators, what's the body, whether the body is an `Yield`, and so on.

If we define the constructors and extractors carefully to capture macro-level abstract syntax (Section 4.3), we can abstract away the micro-level differences in abstract syntax, thus create a *standard abstract syntax*. Macros are expressed in terms of the standard abstract syntax via abstract constructors and extractors, thus solves the portability problem.

4.3 Design of Constructors and Extractors

For the syntax-based approach to work, the constructors and extractors have to be carefully designed. Here we introduce some guiding design principles and discuss how they lead to more portable constructors and extractors.

4.3.1 Prefer macro-structures over micro-structures

This principle can be illustrated with `for` expressions in Scala. Remember from section 2 that there are two equivalent ways to represent `for` expressions: (1) use the pair `ForYield` and `ForDo`; or (2) use the pair `For` and `Yield`.

Both representations capture the syntactic information, but they make a difference for constructors. Imagine that we have two abstract constructors `For` and `Yield`. For compilers that also define `For` and `Yield`, the implementation of the two constructors is easy. However, if a compiler chooses the representation `ForDo/ForYield`, how will it implement the constructor `Yield`? While it's still possible to do that, the implementation must resort to some hack.

In contrast, if the macro system defines two abstract constructors `ForDo` and `ForYield`, there are no such problems. If a compiler has the representation `ForDo` and `ForYield`, the implementation is straight-forward. If a compiler has the representation `For` and `Yield`, we can implement the two constructors as follows:

```
def ForYield(enums: List[Tree], body: Tree): Tree =
    x.For(enums, x.Yield(body))
def ForDo(enums: List[Tree], body: Tree): Tree =
    x.For(enums, body)
```

In the code above `x.For` and `x.Yield` are the tree constructors defined in the compiler.

In the design of Gestalt (Section 5) there are many such cases. For example, in both the Scala 2.x compiler and the Dotty compiler there's a data structure named *template*. It's used to capture the syntactic similarities for the body of class, object, trait and anonymous class definitions. As templates are micro-structures, we don't have templates in Gestalt. Instead, we have constructors for `Object`, `Trait`, `Class` and `AnonymousClass` separately which capture macro-structures of the language syntax.

4.3.2 Design both for syntax and semantics

The design of constructors and extractors is mostly driven by the syntax of the programming language under study. But syntax is not the only concern. For example, syntactically we could represent `f(1)` in the Scala code `f(1) = 3` as a function call. The representation can be justified as it captures all syntactic information of the code. Indeed, that's what the Dotty compiler does.

But doing so in constructors and extractors harms portability. It's quite reasonable to imagine that there might be compilers that represent the code more semantically with a special AST tree `Update`. Now without a proper constructor for `Update`, it's difficult to correctly represent the code `f(1) = 3` in terms of constructors for the particular compiler.

In the design of Gestalt (Section 5), we define different trees for *patterns* for the same reason: syntactically patterns are similar to some other trees, but semantically they are different. Providing special constructors for them better abstracts compiler differences and protects the macro system from potential changes of compiler ASTs.

Programming languages tend to reuse the same syntax for different language features to reduce complexity of the language and make the language easier to learn. In such cases, design of constructors and extractors purely based on syntax is a trap. It's important to design for both syntax and semantics.

4.3.3 Have fewer constructors and extractors

Each abstract constructor and extractor is an assumption on the compiler ASTs about how they represent the abstract syntax. Minimizing the number of constructors and extractors certainly boosts portability. There are several ways to minimize the assumptions: (1) don't provide extractors for syntactic sugars that are difficult to implement; (2) don't provide constructors for deprecated language features; (3) don't provide extractors if there are arguments against its usage. Note that the restrictions are only on macro definitions. In the user code where macros are used, programmers may still use these language features.

In Gestalt (Section 5), we make a lot of such decisions:

(1) No extractors for `for` expressions and some other syntactic sugars (Section 4.4).
(2) No constructors or extractors for deprecated Scala features, like existential types and early initializers for traits.
(3) No extractors for *type trees*, annotations, patterns, etc.

The last decision relies on our assumptions about macro usage. The syntax-based approach enables us to make such assumptions safely while still retain the possibility to provide the API when a use case emerges without breaking compatibility.

4.4 Syntactic Sugars

Syntactic sugars complicate the picture. When does desugaring happen is a compiler implementation detail that's usually not covered by language specifications. Some compilers desugar before macro expansion, some after macro expansion, and some desugarings are irreversible. For example, `for` is a syntactic sugar in Scala, the Scala 2.x compiler desugars it before macro expansion, while Dotty desugars it after macro expansion [2], and it's notoriously difficult to reverse the desugaring. This means that a macro that inspects the structure of `for` expressions may have different semantics under different compilers, which is unacceptable.

The desugaring problem is indeed intriguing. However, it's not a particular problem with the syntax-based approach, the tree-based approach also faces exactly the same problem. A practical solution is to disallow inspecting structures of complex syntactic sugars, as it's impossible to guarantee stable semantics of macros across compilers.

[2]Strictly speaking, Dotty desugars 'for' after the expansion of annotation macros, but before the expansion of def macros.

4.5 Advantages

The syntax-based approach has the following advantages:

(1) *Type Information.* It's easy to access to the type information as the trees are not converted, it's straightforward to define APIs to get the type information associated with the trees.
(2) *Meta-data.* No conversion of ASTs is required, keeping positions, documentations and other attachments on trees is no longer a problem for macro expansion.
(3) *Performance.* Now the whole AST conversion is avoided, there's only necessary cost for AST nodes that are actually manipulated by specific macros.
(4) *Engineering Effort.* Different compilers of the language only need to provide an implementation of the standard extractors and constructors. No need to provide extractors for all syntactic structures, only for extractors that are actually used in macros.
(5) *Feature Control.* We can restrict what constructors or extractors to provide for macros authors, while the tree-based approach has to define data structures for all possible syntactic constructs. For example, in the context of Scala, we can prevent macro authors from inspecting or constructing deprecated language constructs like existential types or early initializers, while in macro user's code these deprecated features can still be used freely.
(6) *API Evolution.* In contrast to the tree-based approach, which has to define all the data structures for all possible language constructs in the beginning, the syntax-based approach prefers a conservative approach by defining a minimum interface of extractors and constructors, and then grow the interface gradually when concrete use cases emerge. This results in better APIs as it avoids the huge upfront design risk of the tree-based approach.
(7) *Boundary of Responsibility.* The syntax-based approach enable us to draw a clear line of responsibility between compilers and the macro system. All contracts with the compilers are clearly stated in the abstract constructors and extractors. It makes it easy to review the contracts and attribute responsibilities. In the implementation, if some abstract constructors or extractors are not implemented, there is going to be typing error.

5 Gestalt: Portable Macros for Scala

Based on the syntax-based approach, we implemented a new macro system for the experimental Scala compiler Dotty: *Gestalt*. The code is hosted on Github: *https://github.com/liufengyun/gestalt*.

Gestalt has some significant improvements over current Scala macros. It solves or improves many problems of the current Scala macro system, including portability, hygiene, owner-chain management, incorrect nesting of trees, path-dependent inconvenience, etc.

We first introduce the current Scala macro system and its problems. Then we introduce Gestalt and discuss how it solves the problems.

5.1 The Current Scala Macro System

The current Scala macro system supports two categories of macros: *annotation macros* and *def macros*. Annotation macros are usually used to transform definitions, e.g. add fields or methods to classes. A typical annotation macro looks like the following:

```
class main extends StaticAnnotation {
  def macroTransform(annottees: Any*): Any =
      macro Main.impl
}
object Main {
  def impl(c: Context)(annottees: c.Expr[Any]*):
       c.Expr[Any] = {
      import c.universe._
      ...
  }
}
```

The abridged code above defines an annotation macro `main`, which can be used to transform the following code:

```
@main
object Test {
  println("hello, world")
}
```

to the following:

```
object Test {
  def main(args: Array[String]): Unit =
       println("hello, world")
}
```

Annotation macros are expanded before type checking, thus the trees provided to annotation macros are devoid of type information. Sometimes, we say annotation macros are syntactical, because they can only use syntactical information of trees to transform user code.

Def macros are usually used to extends the semantics of methods. A typical def macro looks like the following:

```
def assert(cond: Boolean) = macro assertImpl
def assertImpl(c: Context)(cond: c.Expr[Boolean])
     : c.Expr[Unit] = ...
```

The abridged code above defines the macro `assert`, which can be used to produce friendly failure messages if the condition is false. For example, when the code `assert(1 == 2)` is executed, it produces the following output in the console, which is impossible without macros: `1 is not equal to 2`.

Def macros expand after the arguments are type checked, thus they can benefit from the type information of trees to transform user code.

Def macros are widely used in Scala libraries like ScalaTest, Async, Shapeless and etc. The programming idioms enabled by def macros form important extensions to the core Scala language.

Despite the utility of the current Scala macro system, there are several complaints about its usability. The problems are identified in (Burmako 2017) and discussed in `macrology201`[3].

5.1.1 Inconvenience with path-dependent universes

Macro authors have to import the universe in every macro implementation:

```
def impl(c: Context)(annottees: c.Expr[Any]*):
     c.Expr[Any] = {
    import c.universe._
    ...
}
```

The verbosity above might be tolerable, but when a macro author writes a macro helper, the Scala compiler produces a typing error:

```
class Helper(u: Universe) {
    def foo(n: Int): u.Tree = ...
}
val helper = new Helper(u)
val a: u.Tree = helper.foo(3) //error:type mismatch
```

This is because the compiler fails to figure out that the type of the method call `helper.foo(3)` is the same as `u.Tree`. The following solution is proposed in (Burmako 2017), which is an improvement, but still not user-friendly:

```
class Helper[T <: Universe](u: T) {
    def foo(n: Int): T = ...
}
val a: u.Tree = new Helper[u.type](u).foo(3)
```

5.1.2 Problems with owner chains

In the Scala compiler, each definition introduces a new symbol, and each symbol has an owner symbol. The owners of a symbol form a chain, the hierarchy of the owner chain must correspond to the nesting hierarchy of the trees.

It is easy for macros to mess with the owner chain due to restructuring of trees. Suppose there is a macro `map`, which transforms the following code:

```
map(Some(5)) { x =>
  val y = x * x
  println(y)
}
```

to the following:

```
  val x = 5
  val y = x * x
```

[3]https://github.com/scalamacros/macrology201

```
  println(y)
```

A macro author might think that the macro `map` only needs to combine the tree of `val x =5` with the function body. That's incorrect! It also needs to rewire the occurrence of `x` in the function body to the newly defined `x`. Unfortunately, the solution still crashes the compiler! The problem is with the definition of `val y =x * x`. In the original code, the owner of the variable `y` is the anonymous method that defines the function. However, in the macro expansion we forget to change the owner of `y` to the owner of current context.

Owner chain problems cause hard to debug compiler crashes. For macro authors, they need a lot of knowledge about the particular compiler in order to use some hack to deal with owner chain crashes. Burmako (2014) implemented a simple macro for allocation-free option type for Scala. The implementation has 89LOC in total, of which 30LOC are dedicated to deal with owner chains [4]. Needless to say the hacks used in the code are not portable. For a macro author without significant knowledge about compiler internals, it would be very difficult to implement such a macro.

5.1.3 Invalid nesting of typed and untyped trees

Def macros in Scala accept typed trees as input, and produce trees that might be partly typed, partly untyped. Generally, the compiler only supports limited nesting patterns of typed and untyped trees. In particular, nesting of untyped trees inside typed trees is problematic and is rejected. This discipline makes it much easier for the compiler to decide whether to type check a tree or not. This is because blind re-type checking is expensive, and selective type checking of sub-trees is hard to make work due to incomplete context information.

However, current macro system allows programmers to produce arbitrary patterns of nesting, which causes hard to debug compiler crashes.

To alleviate the problem, the macro system introduces APIs for macro authors to manually type check untyped trees. It mitigates the problem, but macro writers need non-trivial compiler knowledge in order to use the APIs. The usage of type checking facilities involves undocumented conventions about compiler internals, which sacrifices usability and portability.

5.1.4 Hygiene of macros

Hygiene is a common problem in macros. Generally, hygiene becomes a problem when a name either in user-program or meta-program changes its originally intended meaning. There are mainly three cases:

(1) Introduction of new names (i.e. definitions) in the meta-program that cause names in user-program to have a different meaning than intended.
(2) Use of names in the meta-program when mixed in user-program has a different meaning than intended.
(3) Restructuring of user-program that changes the intended meaning of names.

The second problem is the most common in Scala macros. Macro authors are recommended to use fully-qualified names to avoid the problem, but there's no facility to enforce the discipline.

5.1.5 Verbosity in syntax

As the following example shows, the current Scala macro system requires the separation between macro declaration and macro implementation, which is not user-friendly:

```
def assert(cond: Boolean) = macro assertImpl
def assertImpl(c: Context)(cond: c.Expr[Boolean])
    : c.Expr[Unit] = ...
```

5.1.6 Portability and IDE experience

As explained in the introduction, the current Scala macro system fails to handle portability of macros, which results in poor IDE experience.

5.2 Gestalt: Introduction

Gestalt has some significant improvements over current Scala macros. It solves or mitigates the problems of the current Scala macro system mentioned above:

(1) It completely removes the annoyances about path-dependent universes.
(2) It frees macro authors from owner chain management completely by handling it automatically.
(3) It rejects the nesting of untyped trees inside typed trees by the type system.
(4) It improves hygiene by the type system, which only accepts fully qualified names by default.
(5) It reduces the boilerplate to define a macro
(6) It solves the portability problem based on the syntax-based approach (Section 4).

This is how to write a dummy macro in Gestalt:

```
import scala.gestalt.api._

object Test {
  def plus(a: Int, b: Int): Int = meta {
    q"$a + $b"
  }
}
```

We follow the `inline/meta` proposal described in Burmako (2017) to use the keyword `meta` as a marker to highlight the semantic difference inside the meta block. We no longer require programmers to separate macro declaration from implementation. More importantly, programmers even don't need to know the existence of universes (or toolbox in our case), which is pervasive in the current macro system.

[4]To be fair, those lines can be reused as pointed out by the author.

We reimplemented some complex macros like *monadless* (Brasil 2017) and *optionless* (Burmako 2014) in Gestalt. Our implementation is easier and more solid. For the example of *optionless*, the implementation (Liu 2017) is 39LOC compared to the original 89LOC. There is not a single line that handles owner chains. The macro is implemented in about 10 minutes and passes the original test set without a single change. Given the existence of heavy hacks to deal with compiler crashes, the original implementation would be very difficult for programmers without significant knowledge about compiler internals.

In the following, we describe the design that enables Gestalt to achieve the goals.

5.3 Gestalt: Design

The core design of Gestalt is centered around the trait `Toolbox`, which defines abstract types, constructors, extractors, and services that compilers should implement. The following code demonstrates the main design of `Toolbox` [5]:

```
trait Toolbox extends Types with Symbols
with Denotations {
  type Tree     >: Null <: AnyRef
  type TypeTree >: Null <: Tree
  type TermTree >: Null <: Tree
  type DefTree  >: Null <: Tree
  type PatTree  >: Null <: Tree
  type Ident    <: TermTree with PatTree

  val tpd: tpdImpl
  trait tpdImpl {
    type Tree     >: Null <: AnyRef
  }

  def Ident: IdentImpl
  trait IdentImpl {
    def apply(name: String): Ident
    def apply(symbol: Symbol): tpd.Tree
    def unapply(tree: Tree): Option[String]
    def unapply(tree: tpd.Tree): Option[Symbol]
  }
}
```

The trait `Toolbox` serves as a contract between the macro system and compilers. It follows the philosophy of syntax-based approach by define abstract constructors and extractors for ASTs.

Scala is a typed language, the trees passed to a macro may be already typechecked and hold type information. Macros can profit the type information to transform the code. That's the reason why inside the toolbox there are both the type `Tree` and `tpd.Tree`. This is signifies the *separation between typed and untyped trees*, we'll explain this major design decision below (Section 5.3.4).

`Toolbox` extends `Types`, `Symbols` and `Denotations`, which enable macro authors to operate on semantic information of ASTs. The trait `Types` defines abstract types and operations for types:

```
trait Types { this: Toolbox =>
  type Type >: Null <: AnyRef
  type TermRef <: Type
  type TypeRef <: Type
  type MethodType <: Type

  def Type: TypeImpl
  trait TypeImpl {
    def =:=(tp1: Type, tp2: Type): Boolean
    def <:<(tp1: Type, tp2: Type): Boolean
    // ...
  }

  def MethodType: MethodTypeImpl
  trait MethodTypeImpl {
    def paramInfos(tp: MethodType): List[Type]
    def instantiate(tp: MethodType)(params:
         List[Type]): Type
    def unapply(tp: Type): Option[MethodType]
    // ...
  }
}
```

The trait `Symbols` define common operations on symbols:

```
trait Symbols { this: Toolbox =>
  type Symbol <: AnyRef

  def Symbol: SymbolImpl
  trait SymbolImpl {
    def name(mem: Symbol): String
    def asSeenFrom(mem: Symbol, prefix: Type): Type

    def isCase(sym: Symbol): Boolean
    def isTrait(sym: Symbol): Boolean
    // ...
  }
}
```

The trait `Denotations` defines `denotation`, a concept borrowed from Dotty, which is basically the meaning of a name:

```
trait Denotations { this: Toolbox =>
  type Denotation

  def Denotation: DenotationImpl
  trait DenotationImpl {
    def name(denot: Denotation): String
    def info(denot: Denotation): Type
    def symbol(denot: Denotation): Symbol
  }
}
```

[5]For presentation purpose, the code is greatly simplified.

Note that in Scala, a symbol alone doesn't give exact meaning to a name because of path-dependent types, type members and generics. The exact meaning of a name in a context has to be the combination of the referred symbol and its type in the specific context.

5.3.1 Safety by construction

By *safety by construction* we mean that the type system will reject construction of syntactically incorrect trees. Safety by construction comes in varying degrees. The tree-based approach supports a high degree of safety by construction by giving different types to different syntactic constructs and enforcing type constraints in the construction of trees.

The syntax-based approach allows us to achieve the same goal, but a little differently. Instead of using concrete types of trees, the macro system defines abstract types for different syntactic constructs and enforce type constraints on constructors to reject ill-formed trees.

More abstract types are not more assumptions on compiler ASTs. For example, in the implementation of the toolbox for Dotty we equate both `TypeIdent` and `Ident` to just `Ident` in Dotty. The compiler doesn't have to define a different data structure for each different abstract type.

5.3.2 Removal of the path-dependent curse

We get rid of the path-dependent curse by the following:

```
object api extends Toolbox {
  private val toolbox: ThreadLocal[Toolbox] =
    new ThreadLocal[Toolbox]

  def Ident =
      toolbox.get.Ident.asInstanceOf[IdentImpl]
}
```

Before macro expansion, the compiler sets its own implementation of the toolbox in the thread-local store. After macro expansion, the toolbox is removed from the store.

Macro authors don't need to know the existence of universes or toolboxes. When they decide to write a macro helper, there's no longer need to pass universes around and work around the type system:

```
import scala.gestalt.api._
object Helper {
  def foo(n: Int): Tree = ...
}
```

5.3.3 Separation of user API from compiler API

The contract between compilers and the macro system is defined by the APIs in `Toolbox`. However, those APIs are not assumed to be used directly by macro users, as they are designed for easy implementation by compilers but not for friendly usage by macro authors. Instead, we define a large amount of friendly helpers that wrap the compiler APIs, thanks to the support of implicits in Scala:

```
object api extends Toolbox {
  implicit class UntypedOps(tree: Tree) { ... }
  implicit class TypedOps(tree: tpd.Tree) { ... }
  implicit class SymbolOps(sym: Symbol) { ... }
  implicit class TypeOps(tp: Type) { ... }
}
```

The separation of user API and compiler API enables us to adopt two design strategies which result in both friendliness and integrity: (1) user APIs are driven by use cases; (2) compiler APIs are generalized over the requirements from user API. For example, when the user API needs to query whether a symbol is a case class of not, the compiler APIs satisfy the requirement by providing APIs to query all flags of symbols.

In development, the separation makes it easy to review and manage the contracts with compilers. It serves as a well-defined boundary between the macro system and compilers, which makes it easy to attribute responsibilities.

5.3.4 Type-safe nesting of typed and untyped trees

The separation of typed and untyped trees enable us to reject the nesting of untyped trees inside typed trees by the type system. The code below shows how the separation is implemented:

```
trait Toolbox {
  type Tree    >: Null <: AnyRef
  type Splice  <: TypeTree with TermTree with
       DefTree
  // ...
  val tpd: tpdImpl
  trait tpdImpl {
    type Tree     >: Null <: AnyRef
  }

  def TypedSplice: TypedSpliceImpl
  trait TypedSpliceImpl {
    def apply(tree: tpd.Tree): Splice
    def unapply(tree: Tree): Option[tpd.Tree]
  }

  def If: IfImpl
  trait IfImpl {
    def apply(cond: TermTree, thenp: TermTree,
        elsep: Option[TermTree]): TermTree
    def unapply(tree: Tree): Option[(TermTree,
        TermTree, Option[TermTree])]

    def apply(cond: tpd.Tree, thenp: tpd.Tree,
        elsep: tpd.Tree): tpd.Tree
    def unapply(tree: tpd.Tree): Option[(tpd.Tree,
        tpd.Tree, tpd.Tree)]
  }
}
```

At the user-facing API, there is an implicit conversion from typed trees to untyped trees to improve programming experience:

```
object api extends Toolbox {
  implicit def tpd2untpd(tree: tpd.Tree): Splice =
      TypedSplice(tree)
}
```

It's impossible to nest untyped trees inside typed trees because the constructors for typed trees only take typed trees.

Note that the separation of typed trees and untyped trees in the macro system doesn't assume the same separation in compilers. For example, the Scala 2.x compiler doesn't have the separation of typed and untyped trees, it can just implement the constructor `Splice` as the identity function, while still enjoys the guarantee that there are no untyped trees nested inside typed trees.

5.3.5 Owner Chain Made Easy

The separation of typed trees and untyped trees enables us to maintain owner chains automatically without programmers knowing their existence. Conceptually, when a programmer composes a complex typed definition by nesting some typed child tree – for instance to define a typed method – it's possible to traverse the child tree to get all the symbols of non-nested definitions in the child tree and update their owners to the symbol of the current definition.

This allows the macro system to completely ignore issues about owner chains. They are technical details that are internal to particular compilers. For example, in the implementation for Dotty, we do it as follows in the typed constructor of `Function`:

```
def Function
(params: List[(String, Type)], resTp: Type)
(bodyFn: List[tpd.Tree] => tpd.Tree): tpd.Tree = {
  val meth = ctx.newSymbol(
    ctx.owner, nme.ANON_FUN,
    Flags.Synthetic | Flags.Method,
    Types.MethodType(params.map(_._1.toTermName),
         params.map(_._2), resTp)
  )
  t.Closure(meth, paramss => {
    ensureOwner(bodyFn(paramss.head), meth)
  })
}
```

Functions are implemented by a method and a closure in Dotty. The method call `ensureOwner` will traverse the non-nested definitions in the function body and set their owners to the current function. That's why we can always set the owner of the method to the current context owner `owner.ctx` – the owner will be updated automatically when it's nested inside another definition.

5.3.6 The Adaptation Problem

Manually constructing typed trees implies programmers need to do apply-insertion, type parameter synthesis, overloading resolution, implicit resolution manually, which is extremely verbose.

In the current Scala macro system, it provides APIs for macro authors to manually typecheck trees. This approach involves undocumented conventions about compiler internals, which sacrifices usability and portability.

The adaptation problem is a special variant of type inference problem where the parameters of methods are well-known. We solve this problem by using the type inference facilities of the underlying compiler. When constructing a typed `Apply` tree, the macro system will automatically insert missing `apply`s, implicit parameters, infer type parameters, resolve overloadings, etc.

5.3.7 Hygiene

In Gestalt, we have two guarantees about hygiene: (1) names in meta-program cannot be polluted by user-program; (2) names in user-program cannot be polluted by meta-program[6]. These two guarantees cover most of the hygiene problems in practice.

To achieve the first goal, the type system enforces that macro authors have to use fully-qualified names beginning from `_root_` or `scala` [7]. The solution is simple, we restrict that the only untyped identifiers that programmers can use is `_root_` and `scala`. This restriction can be implemented with the help of literal types:

```
def Ident(name: "_root_"): Tree
def Ident(name: "scala"): Tree
```

Of course, programmers can freely create typed identifiers based on symbols, which are hygienic. With the facilities above, the following code will not compile:

```
q"List(2, 3)" // q"scala.List(2, 3)" works
```

If the programmer wants to break hygiene intentionally, they have to import an unsafe capability:

```
import scala.gestalt.options.unsafe
q"List(2, 3)"  // it works now
```

As you can imagine, the unsafe constructor is guarded by a capability:

```
def Ident(name: String)(implicit c: Unsafe): Tree
```

Of course, it's always safe to use a resolved name in the meta-program, as the meaning of a typed tree will not change during type checking:

```
def Ident(symbol: Symbol): tpd.Tree
```

The separation of typed and untyped trees enables to achieve the second goal easily: in def macros user-code is

[6]The second guarantee is only valid for def macros.

[7]Supporting prefix `scala` is not necessary, but it increases friendliness.

already type checked, i.e. the meanings of names in user-code are already fixed, it's impossible to pollute the names in user-code unless we confuse typed trees with untyped trees. This can be illustrated by a dummy macro `hygiene`:

```
def hygiene(x: Int): Int = meta {
    q"""
    val x = 4
    $x
    """
}
```

The macro `hygiene` will transform the following code:

```
val x = 5
hygiene(x) // x = 5
```

to the following:

```
val x = 5
{    val x = 4
     x          }    // x = 5
```

Note that after macro expansion, the variable `x` still refers to the original variable, as its meaning is already fixed before macro expansion. We don't have such guarantees for annotation macros, because they are expanded before type checking. Luckily, typical use cases of annotation macros are to enrich class definitions with well-defined fields or methods, where introduction of well-known names to classes is an intended effect.

5.3.8 Platform-Independent Quasiquotes

Another nicety of Gestalt is that quasiquotes are platform-independent. It means that compilers only need to write several lines of glue code in order to reuse the quasiquote implementation. Or no glue at all if we implement quasiquotes as macros in Gestalt, then we get portability for free.

As quasiquotes are meta-programs that construct or deconstruct trees, seemingly quasiquotes can be implemented easily in terms of standard constructors and extractors. However, it is not so obvious if we check the *imaginary* type signature of simple quasiquotes (without unquotes):

```
q : String => Tree[Tree[_]]
```

We use the notation `Tree[T]` to mean that code of the tree when executed will produce a value of type `T`. Given the quasiquote `q"3"`, how it can be represented by the constructors? The obvious answer `Literal(3)` is incorrect, as quasiquotes should return trees that represent trees, not trees themselves! The correct answer should be trees that represent `Literal(3)`, i.e.:

```
Apply(
  Ident("Literal"),
  Literal(3)
)
```

To execute the code above, a toolbox instance is required. Meanwhile, the result of the execution can be executed again, and they need another toolbox instance. The two toolboxes are different, because they are used at completely different stages: the first toolbox is used during macro definition, the second is used during macro expansion.

5.4 Assumptions

A good design depends on good assumptions. In the implementation of Gestalt, we use many assumptions to minimize the number of supported extractors. The syntax-based approach enables us to make bold assumptions on this matter – when a concrete use case surfaces, we can always provide support for the required extractors.

We list the major assumptions below. We are conscious that the assumptions are opinionated, as it's the case in most design activities. However, as they are important to the design, we think it's good to document them so that they can be debated, validated or invalidated, thus serve as a basis to improve the design.

1. Macros should never match modifiers exactly
2. Quasiquotes should never be used as patterns
3. No extractors are required for type trees
4. No extractors are required for pattern trees
5. No extractors are required for `for` comprehensions
6. No extractors are required for `Self`
7. No extractors are required for `this` and `super`
8. No extractors are required for parent definition list
9. No extractors are required for `import` statements

We could add more to the list: no extractors for primary and secondary constructors, etc. We justify some of the above assumptions by reasoning and/or empirical evidence.

Macros should never match modifiers exactly. This is more like a rule. The reason is that modifiers like `private`, `implicit`, `lazy` can be reordered without changing the semantics of programs, and it's impossible to know in advance how many modifiers a definition could have. As a result, all macros that match modifiers exactly are buggy. This leads us to model modifiers as a black box of flags, which programmers can query or set. If a macro author attempts to match modifiers exactly, the macro system will instruct the compiler to produce an error message.

Quasiquotes should never be used as patterns. We think that use of quasiquotes as patterns is an abuse. First, patterns tend to be simple. Complex patterns with rich details make too many assumptions on user code, which is usually buggy. With extractors and extractor helpers, use of extractors makes the pattern match code cleaner (no strings and dollars). Second, the semantics of extractors are more transparent and predictable than quasiquotes. Macro authors can get type information about extractors from IDEs. Third, it's easy to make mistakes in quasiquote patterns. For example, forget to put `$_` both before the class definition and the argument list to handle possible modifiers when matching against a class definition. If extractors are used, the type system guards against such mistakes. Fourth, from our experience with macros and impression with libraries, extractors are much more widely used than quasiquote patterns.

No extractors are required for type trees. First, Scala supports type alias and imports, the syntactic information of types is not informative at all. Programmers can't tell whether a type is `Int` by checking whether it's shape is the same as `scala.Int`. Second, Scala supports local type inference, type arguments are optional. Macros that operate on the shape of type trees will be unable to handle inferred type trees. In a word, the programmers can only do wrong things by inspecting the structure of type trees.

No extractors are required for pattern trees. There is no good reason to transform patterns in user code. If there's indeed a use case, it should be discouraged, as it reduces maintainability of the code and harms predictability of program behaviors. The only macro we know that manipulates pattern trees is quasiquotes, but we have discussed that the usage is problematic.

No extractors are required for 'for' comprehensions. As we discussed in Section 4.4, it's tricky to guarantee stable semantics across compilers for macro code that inspect structures of `for` comprehensions. Meanwhile, macros that only handle `for` but not `map` or `flatMap` are problematic, as the former desugars to the latter. Finally, in all Scala compilers we know of, `for` comprehensions are already desugared before the expansion of def macros.

No extractors are needed for 'Self'. The self-annotation in class or object definition is optional. Note that the most important information in self-annotation is the types, but we have argued above that no extractors are required for type trees. We don't know any macros that inspect self-annotations.

No extractors are needed for 'this' and 'super'. We don't know any macros that profit from inspecting `this` or `super`.

No extractors are needed for parent definition list. First, parent definition is mostly about types, and we have argued type trees should not be inspected. Second, doing any changes to parent list will greatly reduce maintainability and the predictability of program behavior, thus discouraged. Third, we don't know any macros that manipulate parent definitions.

No extractors are needed for 'import' statements. We don't know any macros that manipulate `import` statements. Changing the semantics of `import` is discouraged as it reduces predictability of programs.

6 Related Work

As far as we know, portability of macros is largely ignored in the research of meta-programming, while there are plentiful studies on hygiene (Dybvig et al. 1993; Kohlbecker et al. 1986) and type-directed meta-programming (Davies and Pfenning 2001; Ganz et al. 2001; Lämmel and Jones 2003; Sheard and Jones 2002; Taha and Sheard 2000).

`scala.reflect` is the first experiment to introduce macros to Scala. It has brought significant benefits to programmers. It takes a similar approach based on constructors and extractors as proposed in this paper. However, the vision is quite different. For example, `scala.reflect` defines the trait `IfApi` which is intended to be inherited by all compiler data structures that represent `if`/`else` expressions. Imposing a common set of traits is just one step away from imposing that all compilers should use the same ASTs, which is even harder to achieve than standardizing ASTs just for macros.

The `scalameta` project intends to be the successor of the experimental `scala.reflect` macros. It defines a standard set of ASTs and macros are defined in terms of the standard ASTs. The compiler does conversion between compiler trees and Scalameta trees during macro expansion. The `scalameta` project eventually got blocked with the implementation of semantic APIs. Nevertheless, the project directly inspired us to invent the syntax-based approach, which is impossible without the experiments in `scalameta`.

7 Conclusion

In this paper we presented the syntax-based approach to implement portable macros based on standard constructors and extractors. We showed that this approach has many advantages over the tree-based approach. We implemented a macro system based on this idea and reaped significant benefits compared to alternative solutions. Our solution outlined in this paper has been adopted in the official new Scala macro system.

Acknowledgments

We thank Mark Lemay for reading the draft and providing valuable feedback. The idea of separating typed and untyped trees in macros comes from Martin Odersky. We thank Denys Shabalin for his tutorial on quasiquotes (Shabalin 2014). We thank Valdis Adamsons for his contribution to the Gestalt project.

References

Flavio Brasil. 2017. monadless. http://monadless.io. (2017). Accessed: 2017-06-20.

Eugene Burmako. 2013. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*. ACM, 3.

Eugene Burmako. 2014. Allocation-free option type for Scala. https://github.com/scalamacros/macrology201/blob/part1/macros/src/main/scala/Macros.scala. (2014). Accessed: 2017-06-20.

Eugene Burmako. 2017. *Unification of Compile-Time and Runtime Metaprogramming in Scala*. PhD Thesis, EPFL.

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *Journal of the ACM (JACM)* 48, 3 (2001), 555–604.

R Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1993. Syntactic abstraction in Scheme. *Lisp and symbolic computation* 5, 4 (1993), 295–326.

Steven E Ganz, Amr Sabry, and Walid Taha. 2001. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. *ACM SIGPLAN Notices* 36, 10 (2001), 74–85.

Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM, 151–161.

Ralf Lämmel and Simon Peyton Jones. 2003. *Scrap your boilerplate: a practical design pattern for generic programming*. Vol. 38. ACM.

Fengyun Liu. 2017. Allocation-free option type in Gestalt. https://github.com/liufengyun/gestalt/blob/master/macros/src/main/scala/gestalt/macros/Optional.scala. (2017). Accessed: 2017-06-20.

Denys Shabalin. 2014. Joyquote. https://github.com/densh/joyquote. (2014). Accessed: 2017-06-26.

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, 1–16.

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science* 248, 1 (2000), 211–242.