Unification of Compile-Time and Runtime Metaprogramming in Scala

THÈSE N^{O} 7159 (2017)

PRÉSENTÉE LE 6 MARS 2017

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Eugene BURMAKO

acceptée sur proposition du jury:

Prof. J. R. Larus, président du jury
Prof. M. Odersky, directeur de thèse
Dr D. Syme, rapporteur
Prof. S. Tobin-Hochstadt, rapporteur
Prof. V. Kuncak, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

To future travels.

Acknowledgements

I would like to heartily thank my advisor Martin Odersky for believing in my vision and trusting me with the autonomy to explore its avenues. It was astonishing to see my work shipped as part of a programming language used by hundreds of thousands of developers, and it would not have been possible without Martin's encouragement and support.

It is my pleasure to thank the members of my thesis jury, Viktor Kunčak, James Larus, Don Syme and Sam Tobin-Hochstadt for their time and insightful comments.

I am grateful to my colleagues at EPFL and students whom I supervised during my PhD studies. We hacked many fun things and shared many good times together. My special thanks goes to Denys Shabalin whose relentless passion for excellence shaped our numerous joint projects. A lot of the ideas described in this dissertation were born from discussions and collaborations with Denys.

I am continuously humbled by the privilege of being part of the Scala community. During my time at EPFL, I had the chance to work together with Scala enthusiasts all over the world, and that was an amazing experience. Were I to undertake my projects alone, some of the most interesting discoveries would most likely not have happened at all. I did my best to give credit for the most important contributions in the individual chapters of the dissertation.

Finally, I would like to thank my family - Nikolai, Elena, Julia, Andrey and cute little Anna. Whatever was going on at work, however much I was hacking during our rare get-togethers, I always knew that I had their love and support.

27 October 2016 Eugene Burmako

Abstract

Metaprogramming is a technique that consists in writing programs that treat other programs as data. This paradigm of software development contributes to a multitude of approaches that improve programmer productivity, including code generation, program analysis and domain-specific languages. Many programming languages and runtime systems provide support for metaprogramming.

Programming platforms often distinguish the notions of compile-time and runtime metaprogramming, depending on the phase of the program lifecycle when metaprograms execute. It is common for different lifecycle phases to be hosted in different environments, so it is also common for different kinds of metaprogramming to provide different capabilities to metaprogrammers.

In this dissertation, we present an exploration of the idea of unifying compile-time and runtime metaprogramming in Scala. We focus on the practical aspect of the exploration; most of the described designs are available as popular software products, and some of them have become part of the standard distribution of Scala.

First, guided by the motivation to consolidate disparate metaprogramming techniques available in earlier versions of Scala, we introduce scala.reflect, a unified metaprogramming framework that uses a language model derived from the Scala compiler to run metaprograms both at compile time and at runtime.

Secondly, armed by the newfound metaprogramming powers, we describe Scala macros, a language-integrated compile-time metaprogramming facility based on scala.reflect. Thanks to the comprehensive nature of scala.reflect, macros are able to work with both syntactic and semantic information about Scala programs, enabling a wide range of previously impractical or impossible use cases.

Finally, based on our experience and user feedback, we identify key strengths and weaknesses of scala.reflect and macros. We propose scala.meta, a new unified metaprogramming framework, and inline/meta, a new macro system based on scala.meta, that take the best from their predecessors and address the most important problems.

Keywords: Programming Languages, Compilers, Metaprogramming, Reflection, Macros.

Astratto

Metaprogrammazione è la tecnica che consiste nel creare programmi che trattano gli altri programmi come dati. Questo paradigma di sviluppo software contribuisce alla multitudine di approcci che migliorano la produttività di programmatore, compresi la generazione di codici, i procedimenti automatizzati di analisi del software, e i linguaggi specifici di dominio. Molti linguaggi di programmazione e sistemi a tempo di esecuzione consentono la metaprogrammazione.

Le piattaforme software spesso fanno distinzione tra le nozioni di metaprogrammazione a tempo di compilazione e a tempo di esecuzione, a seconda della fase del ciclo di vita del software quando sono eseguiti i metaprogrammi. È normale che cicli di vita diversi siano presentati in ambienti diversi, ed è altresì normale che diversi tipi di metaprogrammazione forniscano capacitá diverse ai metaprogrammatori.

In questa dissertazione presentiamo la ricerca dell'idea di unire la metaprogrammazione a tempo di compilazione e a tempo di esecuzione in Scala. Particolare risalto viene dato all'aspetto pratico della ricerca: la maggior parte dei descritti design sono disponibili come popolari prodotti software, ed alcuni di loro sono parte integrante della distribuzione standard di Scala.

Innanzitutto, guidati dalla motivazione di consolidare le tecniche disparate di metaprogrammazione disponibili in versioni precedenti di Scala, introduciamo scala.reflect, un framework unificato di metaprogrammazione che usa il modello di linguaggio derivato dal compilatore Scala per eseguire i metaprogrammi sia a tempo di compilazione, sia a tempo di esecuzione.

Secondariamente, muniti delle nuove capacità fornite dalla metaprogrammazione, descriviamo le macroistruzioni, un'infrastruttura di metaprogrammazione a tempo di compilazione integrata al linguaggio, basata su scala.reflect. Grazie alla natura esaustiva di scala.reflect, le macroistruzioni possono adoperare sia informazioni sintattiche, sia semantiche sui programmi Scala, consentendo una vasta gamma di casi d'uso che precedentemente erano impraticabili o impossibili.

Infine, basati sulla nostra esperienza e commenti degli utenti, abbiamo identificato i punti di forza e deboli di scala.reflect e macros. Proponiamo scala.meta, un nuovo framework

unificato di metaprogrammazione, e inline/meta, un nuovo sistema di macroistruzioni basato su scala.meta, che unisce gli elementi migliori dei suoi predecessori, risolvendone allo stesso tempo le pecche più evidenti.

Parole chiave: linguaggi di programmazione, compilatori, metaprogrammazione, riflessione, macroistruzione

Contents

Acknowledgements i

Abstract (English/Italiano) iii

Table of contents x

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Target audience . . . 2
- 1.3 Preliminaries . . . 2
- 1.4 Terminology . . . 3
- 1.5 Thesis statement . . . 4
- 1.6 Contributions . . . 5

I Unification 7

2 Ad hoc metaprogramming 9
- 2.1 JVM reflection . . . 9
- 2.2 Scala signatures . . . 12
- 2.3 On-demand reification . . . 14
- 2.4 Compiler plugins . . . 16
- 2.5 Conclusion . . . 18

3 Scala.reflect 19
- 3.1 Intuition . . . 20
  - 3.1.1 High-level architecture . . . 20
  - 3.1.2 Example: the isImmutable metaprogram . . . 26
  - 3.1.3 Compile-time execution . . . 29
  - 3.1.4 Runtime execution . . . 31
  - 3.1.5 Sharing metaprograms . . . 34
- 3.2 Language model . . . 37
  - 3.2.1 Overview . . . 38
  - 3.2.2 Trees . . . 41

3.2.3 Symbols . . . 43
3.2.4 Types . . . 48
3.3 Notations . . . 50
3.3.1 Reify . . . 51
3.3.2 Quasiquotes . . . 55
3.3.3 TypeTags . . . 58
3.3.4 Hygiene . . . 60
3.4 Environments . . . 62
3.4.1 Scala compiler . . . 63
3.4.2 Dotty . . . 64
3.4.3 IDEs . . . 64
3.4.4 JVM . . . 65
3.4.5 JavaScript . . . 66
3.4.6 Native . . . 67
3.5 Operations . . . 67
3.6 Conclusion . . . 72

II Scala macros **75**

4 Def macros **77**
4.1 Motivation . . . 78
4.2 Intuition . . . 79
4.3 Macro defs and macro impls . . . 84
4.3.1 Term parameters . . . 85
4.3.2 Type parameters . . . 88
4.3.3 Separate compilation . . . 89
4.3.4 Split or merge? . . . 90
4.4 Macro expansion . . . 92
4.4.1 Expansion pipeline . . . 93
4.4.2 Effects on implicit search . . . 95
4.4.3 Effects on type inference . . . 95
4.5 Macro APIs . . . 97
4.6 Feature interaction . . . 97
4.6.1 Macro defs . . . 98
4.6.2 Macro applications . . . 100
4.6.3 Java interop . . . 102
4.7 Conclusion . . . 102

5 Extensions **105**
5.1 Pattern macros . . . 105
5.1.1 Motivation . . . 106
5.1.2 Pattern expansion via def macros . . . 107

5.1.3 Conclusion . . . 111
5.2 Type macros . . . 112
5.2.1 Motivation . . . 112
5.2.2 Type expansion . . . 113
5.2.3 Conclusion . . . 116
5.3 Macro annotations . . . 117
5.3.1 Motivation . . . 117
5.3.2 Definition expansion . . . 118
5.3.3 Enabling semantic APIs . . . 120
5.3.4 Ensuring consistent compilation results . . . 122
5.3.5 Revisiting the separate compilation restriction . . . 125
5.3.6 Conclusion . . . 126

6 Case studies **127**
6.1 Materialization . . . 128
6.1.1 Essence of materialization . . . 128
6.1.2 Integration with vanilla implicits . . . 129
6.1.3 Tying the knot . . . 130
6.1.4 Fundep materialization . . . 131
6.1.5 Conclusion . . . 135
6.2 Type providers . . . 136
6.2.1 Public type providers . . . 136
6.2.2 Anonymous type providers . . . 137
6.2.3 Conclusion . . . 140
6.3 Type-level programming . . . 140
6.4 Internal DSLs . . . 143
6.5 External DSLs . . . 145

7 Tool support **147**
7.1 Code comprehension . . . 148
7.2 Error reporting . . . 149
7.3 Incremental compilation . . . 150
7.4 Testing . . . 151
7.5 Debugging . . . 151
7.6 Documentation . . . 153
7.7 Conclusion . . . 153

III Beyond unification **155**

8 Scala.meta **157**
8.1 Intuition . . . 158
8.1.1 High-level architecture . . . 158

8.1.2 Revisiting isImmutable . . . 162
8.1.3 Executing isImmutable . . . 164
8.2 Language model . . . 166
8.3 Notations . . . 171
8.4 Environments . . . 174
8.5 Operations . . . 177
8.6 Conclusion . . . 181

9 New-style macros **183**
9.1 Motivation . . . 184
9.2 Intuition . . . 185
9.3 Inline and meta . . . 189
9.3.1 Inline definitions . . . 189
9.3.2 Inline reduction . . . 191
9.3.3 Meta expressions . . . 192
9.3.4 Meta expansion . . . 194
9.3.5 Losing whiteboxity . . . 195
9.4 Meta APIs . . . 199
9.5 Extensions . . . 200
9.6 Tool support . . . 200
9.7 Conclusion . . . 201

10 Related work **203**

11 Conclusion **209**

1 Introduction

1.1 Background

Scala has a strong tradition of metaprogramming. In addition to runtime reflection available out of the box on the Java Virtual Machine, its main target platform, Scala has always featured additional metaprogramming facilities of its own.

Since its very beginnings, the Scala compiler could persist a limited subset of its symbol table and syntax trees in a form designed to be loaded and analyzed using functionality provided in the standard distribution. As Scala grew in features and popularity, the demand for metaprogramming also grew, and the scope of metaprogramming organically expanded to include compiler plugins and limited type persistence.

These early metaprogramming capabilities were developed independently from each other and, as a result, ended up being limited and incompatible. For example, even though a combination of symbol table, AST and type persistence theoretically enables introspection of the entire language, in reality these persistence mechanisms were implemented by separate subsystems of the compiler and used incompatible data structures.

Later on, prompted by the ideas of my predecessor Gilles Dubochet, the Scala compiler went through a series of internal refactorings that reconciled the differences between the previously available metaprogramming facilities. We modified all these facilities to use compiler internals as their API, and then changed the compiler itself to be able to transparently launch and operate at runtime. This marked the birth of scala.reflect - a unified metaprogramming framework that works both at compile time and at runtime.

Scala.reflect became part of the standard distribution and made several important advancements. First, it enabled macros - compile-time metaprograms that are executed inside the compiler. Secondly, it provided a way to perform runtime reflection in terms of Scala features, as opposed to a limited view exposed by the JVM. Being fit for these quite different tasks has become possible thanks to the unified nature of scala.reflect.

Scala.reflect made a profound impact on metaprogramming in Scala. The power and novelty of its API catalyzed metaprogramming research in the Scala ecosystem and enabled many practical use cases that were previously out of reach or required significant amounts of boilerplate. At the time of writing, dozens of influential open-source projects in the Scala community use scala.reflect, including: Akka, Algebird, Breeze, Circe, Finagle, Monocle, Pickling, Play, Sbt, Scala.js, ScalaTest, Scalding, Shapeless, Simulacrum, Slick, Spark, Specs2, Spire, Spray and others.

Below, I will provide a first-person account of unification that happened in Scala metaprogramming and its effects on the ecosystem. We will see what worked, what did not and what directions were taken by further evolution of metaprogramming in Scala.

1.2 Target audience

We hope that this dissertation will be useful to programming language enthusiasts and compiler hackers looking to enrich their languages with metaprogramming facilities.

First and foremost, this dissertation is designed as an experience report of developing and evolving a language-integrated metaprogramming framework. We will highlight original motivations, document practical experiences and provide actionable insights.

1.3 Preliminaries

This dissertation is markedly Scala-centric. We will talk about metaprogramming Scala, which is done in Scala using tools designed and implemented in Scala. We believe that our results are applicable to other programming languages, but understanding them fully will require some knowledge of Scala.

We will introduce more advanced aspects of Scala and its infrastructure where necessary, but being comfortable with the basics of the language is required for reading the dissertation. Luckily, most readers should find Scala code familiar, because it uses ubiquitous C-style syntax and shares many language features with Java. For further information on distinctive Scala features such as case classes, objects, traits and others, we recommend "Programming in Scala" which currently has its first edition freely available online [88].

The main platform supported by Scala is the Java Virtual Machine [76]. While there exist compiler backends that can produce JavaScript [34] and native code [105], the JVM is by far the most popular platform for Scala programs. Therefore, readers may find this dissertation occasionally using terminology specific to the JVM. Concretely, we will be talking about *class files*, *classpaths* and *classloaders* to denote units of software distribution, configurations that specify locations of class files and entities that define execution environments by supervising loading of class files [73].

Finally, we will often refer to developer tools used the Scala ecosystem. The most important of such tools is the Scala compiler [43], also called `scalac`. The work described in this dissertation began early in the development cycle of the Scala compiler 2.10.0, and the currently available stable release is 2.11.8.

`scalac` has long been the only Scala compiler. However, over the last several years, several alternative compilers have emerged. The most influential of them is Dotty [42], a next generation compiler for Scala developed by Martin Odersky, the creator of the language, and his research group at EPFL.

There are three IDEs available for Scala: a Scala plugin for IntelliJ IDEA [65] (referred to as just IntelliJ), a Scala plugin for Eclipse [99], and ENSIME [44], a headless server that provides semantic information about Scala programs to text editors such as Emacs, Vim, Sublime Text and Atom. Under the covers, Eclipse and ENSIME use the Scala compiler running in a special mode, whereas IntelliJ features their own implementation of a compiler frontend.

Other tools mentioned in this dissertation are the Scala REPL [43] (referred to as just REPL), which we will be frequently using to run code examples, sbt [74], the most popular Scala build tool, and scaladoc [43], the Scala documentation tool.

1.4 Terminology

In this dissertation, we assume that readers have a background in programming languages and compilers, so we will be freely using the terminology from that area of knowledge.

Additionally, in order to concisely refer to certain notions, we will also utilize more specialized language. In section 1.3, we introduced some Scala jargon, and in this section, we follow up with metaprogramming terminology.

A *metaprogramming framework* (also, a *metaprogramming library*) is a collection of data structures and operations that allow to write *metaprograms*, i.e. programs that manipulate other programs as data. For example, scala.reflect which is developed and described in this dissertation is a metaprogramming framework, because it contains classes like `Tree`, `Symbol`, `Type` and others that represent Scala programs as data and exposes operations like prettyprinting, name resolution, typechecking and others that can manipulate this data.

We say that metaprogramming frameworks provide the ability to *reflect* programs, i.e. *reify* program elements as data structures that can be *introspected* by metaprograms. For instance, scala.reflect provides facilities to reflect Scala programs. These facilities include the ability to reify definitions written in the source code as instances of class `Symbol`. Metaprograms can call methods like `Symbol.info` to introspect these definitions.

In the context of this dissertation, *environment* means an environment where metaprograms are executed. For example, a compiler run defines a compile-time environment, and a JVM instance defines a runtime environment. There are many other environments, e.g. code editors and code analysis tools, but in this work we will almost exclusively on compile-time and runtime environments.

A *language model* consists of data structures, such as abstract syntax trees, symbols or types, that are used by a metaprogramming framework to represent fragments of language syntax, concepts from the language specification, etc. For instance, the language model of scala.reflect consists of classes like `Tree`, `Symbol`, `Type` and others.

We can call a language model *portable* if it can be used to represent programs in multiple environments. For example, the language model of scala.reflect is portable, because it uses the same collection of data structures to represent Scala programs both at compile time and at runtime.

A *language element*, *language artifact*, *reflection artifact* or just *artifact* is one of the data structures constituting the language model. Occasionally, we will overload these terms to also mean instances of these data structures. For example, class `Tree` is an element of the scala.reflect language model. Additionally, trees, i.e. instances of class `Tree`, are also elements of that language model.

A *mirror* is an entity that can reflect an environment into reflection artifacts. For instance, a scala.reflect `Mirror` is a mirror, because it reflects an environment into a symbol table, i.e. a collection of symbols that represents definitions in that environment. The notion of mirrors used in this dissertation is inspired by the notion of mirrors introduced by Gilad Bracha and David Ungar in [7].

A *unified metaprogramming framework* is a metaprogramming framework that uses the same language model to reflect both compile-time and runtime environments. In such a framework, the same API can be used to write compile-time and runtime metaprograms, and same metaprograms can be run both at compile time and at runtime. For example, scala.reflect is a unified metaprogramming framework. In his dissertation [35], Gilles Dubochet pondered over unified metaprogramming in the context of domain-specific programming, and his initial ideas were a major source of inspiration for our work.

1.5 Thesis statement

A metaprogramming framework that unifies compile-time and runtime reflection in a statically-typed programming language is feasible and useful. A portable language model that naturally arises from unification fosters reuse of metaprogramming idioms across a multitude of environments. Such a common vocabulary of idioms can simplify and stimulate development of code analysis and generation tools.

1.6 Contributions

This dissertation describes design, implementation and applications of unified metaprogramming frameworks based on the examples of scala.reflect, Scala macros and their emerging successors. Concretely, the contributions of this dissertation are the following:

- Scala.reflect, a unified metaprogramming framework that distills the language model of the Scala compiler. We describe the architecture that allows the implementation of the framework to reuse the existing compiler codebase and validate this architecture by demonstrating compile-time and runtime implementations of scala.reflect that have been shipped with Scala 2.10 (chapter 3).
- Def macros, a language-integrated compile-time metaprogramming facility based on scala.reflect that expands applications of specially-defined methods. We describe the design of def macros that has become part of Scala 2.10 and highlight the importance and the consequences of its tight integration with the infrastructure behind regular methods (chapter 4).
- Exploration of the design space of Scala macros. Similarly to how we did with def macros and regular methods, we integrate compile-time metaprogramming with other language features. We discuss several extensions to the macro system that gained popularity in the community, including pattern macros, type macros and macro annotations (chapter 5).
- Validation of utility of Scala macros. Through a number of real-world case studies, we show how macros can be used to implement code generation, program analysis and domain-specific languages (chapter 6).
- Survey of problems in developer tools caused by Scala macros. Disregarding tool support in the initial implementation of macros was our biggest design mistake, so we document this experience and hope that our account will prevent similar mishaps in other programming languages (chapter 7).
- Scala.meta, a unified metaprogramming framework that is created from scratch to ensure interoperability with developer tools. We describe how our experience with scala.reflect led to creation of a minimalistic implementation-independent language model whose prototypes have been able to reach and surpass the functionality provided by scala.reflect (chapter 8).
- Inline/meta, a new macro system that takes the best from the design of its predecessor - its smooth integration into the language - and reimplements it on top of scala.meta to improve user experience and tool support (chapter 9).

Unification Part I

2 Ad hoc metaprogramming

In this chapter, we provide a survey of some of the metaprogramming techniques available in the Scala standard distribution before unification, briefly describing JVM reflection (section 2.1), Scala signatures (section 2.2), on-demain reification (section 2.3) and compiler plugins (section 2.4). The work of designing and implementing this functionality was done before my time at EPFL, so credit for the techniques described below goes to their respective authors.

2.1 JVM reflection

The oldest way of doing metaprogramming in Scala is by utilizing runtime reflection capabilities of the JVM. These capabilities are provided by runtime classes, i.e. instances of the class `java.lang.Class` defined in the Java Development Kit, and are available in Scala since day one.

In Scala, there are two ways to obtain runtime classes. First, there is the standard `classOf[T]` method that returns a runtime class corresponding to the type argument passed to this method. Secondly, all values in Scala have a `getClass` method inherited from `Any`, the top type of the Scala type system. This method returns a runtime class underlying the given object.

```
scala> class Point(val x: Int, val y: Int) {
     |   override def toString = s"Point($x, $y)"
     | }
defined class Point

scala> val staticClass = classOf[Point]
staticClass: Class[Point] = class Point

scala> val point = new Point(40, 2)
point: Point = Point(40, 2)
```

```
scala> val dynamicClass = point.getClass
dynamicClass: Class[_ <: Point] = class Point
```

As we can see, `classOf` provides a runtime class of a statically known type, therefore `staticClass` is typed as `Class[Point]`. By contrast, `Any.getClass` obtains a runtime class of a dynamic object, so `dynamicClass` is typed as `Class[_ <: Point]`, since an object of type `Point` can actually be one of the subclasses of `Point`.

With runtime classes - instances of `Class` - it is possible to introspect the structure of the program being executed, i.e. enumerate fields, methods and constructors declared in classes, and get runtime representations for them. Moreover, using these representations, metaprogrammers can alter the state of the program, i.e. get and set values of fields, invoke methods and constructors, etc.

For details, we refer the interested reader to the specification of the Java Development Kit [91], and here we will limit ourselves to a simple illustration. In the example below, we ask a runtime class of `Point` for the list of declared fields, then obtain a field that corresponds to `Point.x` and set its value in a given object.

```
scala> classOf[Point].getDeclaredFields
res2: Array[java.lang.reflect.Field] =
Array(private final int Point.x, private final int Point.y)

scala> val fieldX = classOf[Point].getDeclaredField("x")
fieldX: java.lang.reflect.Field = private final int Point.x

scala> fieldX.setAccessible(true)

scala> point
res4: Point = Point(40, 2)

scala> fieldX.set(point, 42)

scala> point
res6: Point = Point(42, 2)
```

While almost the entire example is fairly straightforward, the call to `setAccessible` sticks out like a sore thumb. Without this call, `Field.set` fails with an exception saying "class can not access a member of class Point with modifiers private final", which is a surprise, because we expect `val x` in `class Point(val x: Int, ...)` to be public.

This experience highlights the problem with using JVM reflection in Scala programs. Runtime classes reflect the structure of the program *as seen by the JVM* (or its emulation layer on other target platforms), not as it was originally written.

Even though Scala programs compile down to JVM class files, the semantics of Scala is significantly richer than what is supported by the JVM. Vals/vars, objects, traits, implicits, higher-kinded types - this is just a small subset of features that are present in Scala, but do not have direct analogues on the JVM.

As a result, the Scala compiler has to either encode such features with what is available on the target platform or to erase these features altogether. Encoded features, e.g. vals and objects, can be introspected with JVM reflection and then reverse-engineered like we did with `setAccessible`. Erased features, e.g. implicits and higher-kinded types, are available only in their JVM-compatible form, which most of the time cannot be reconstructed.

Below we can see how `class Point(val x: Int, val y: Int)` is compiled on the JVM. This output is obtained by running the Scala compiler on this snippet of code to produce the `Point.class` class file and then executing the `javap` utility from the standard distribution of the JVM to decompiles that class file into textual form.

```
public class Point {
  private final int x;

  private final int y;

  public int x();
    Code:
       0: aload_0
       1: getfield      #14 // Field x:I
       4: ireturn

  public int y();
    Code:
       0: aload_0
       1: getfield      #18 // Field y:I
       4: ireturn

  public Point(int, int);
    Code:
       0: aload_0
       1: iload_1
       2: putfield      #14 // Field x:I
       5: aload_0
       6: iload_2
       7: putfield      #18 // Field y:I
      10: aload_0
      11: invokespecial #23 // Method java/lang/Object.<init>:()V
      14: return
}
```

Note how each `val` parameter of the primary constructor was turned into a triplet that consists of a constructor parameter, a private underlying field and a public getter method. With this knowledge in mind, we should no longer be surprised that that the field `Point.x` loaded by JVM reflection turned out to be private.

From the discussion above, we can see that JVM reflection is easily available, but insufficient for introspecting Scala programs. Many language features of Scala can not be mapped directly on JVM features and therefore have to be encoded or erased, which makes which makes the representation of a Scala program as viewed by JVM reflection unfaithful and quite divergent from what the programmer actually wrote.

2.2 Scala signatures

In order to support separate compilation, the Scala compiler has to store information about definitions and their signatures. Due to the reasons mentioned in section 2.1, encoding definitions with JVM-compliant metadata is not going to work, so there clearly has to be a backchannel that deals with features unsupported by the JVM. This backchannel is called Scala signatures, and is available in Scala since the earliest versions.

For every top-level definition, the Scala compiler serializes a fragment of its internal symbol table that represents this definition and all globally-accessible definitions inside it (vals, vars, type members, methods, constructors, nested classes, traits and objects, etc). Serialized payload, also called a Scala signature, is then stored in the class file corresponding to that top-level definition in the format that depends on the version of the Scala compiler.

Below we can see a fragment of the ouput of the `javap` utility run in verbose mode on the class file produced by compiling `class Point(val x: Int, val y: Int)`. Unlike normal mode of `javap` that prints just the definitions in the given class file and maybe their bytecode, as shown in section 2.1, verbose mode exposes internal structural parts of the class file.

```
Classfile /Users/xeno_by/Projects/2118/sandbox/Point.class
  Last modified Jul 14, 2016; size 805 bytes
  MD5 checksum 75f38596a547216ffd8cba4e39f94b0f
  Compiled from "Point.scala"
public class Point
  SourceFile: "Point.scala"
  RuntimeVisibleAnnotations:
    0: #6(#7=s#8)
    ScalaInlineInfo: length = 0x13
     01 00 00 03 00 13 00 14 00 00 09 00 0C 00 00 0B
     00 0C 00
```

```
    ScalaSig: length = 0x3
     05 00 00
  minor version: 0
  major version: 50
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Utf8               Point
   #2 = Class              #1             //  Point
   #3 = Utf8               java/lang/Object
   #4 = Class              #3             //  java/lang/Object
   #5 = Utf8               Point.scala
   #6 = Utf8               Lscala/reflect/ScalaSignature;
   #7 = Utf8               bytes
   #8 = Utf8               }1A!\t)k\5oi*\t1!A=K6H/ MA\t)i)1oY1m...
   ...
```

`javap` demonstrates that the class file contains the `ScalaSig` custom attribute, whose bytes signify the major and the minor version of the format (i.e. 5.0), as well a mysterious runtime-visible annotation represented by `#6(#7=s#8)`. Knowing the convention that `#N` stands for the N-th entry in the constant pool of the class file, also printed by `javap`, we can decipher that this annotation is called `scala.reflect.ScalaSignature` and it has an argument called `bytes` with what looks to be binary payload. Deserializing this payload will yield the list of members defined in `Point` as seen by the Scala compiler.

Instead of loading Scala signatures, obtaining the binary payload and manually deserializing it, one can also use the `scalap` utility from the standard distribution, available since Scala 1.0. This utility deserializes the signatures on its own and then prettyprints them to the console, which represents a significant user experience improvement in comparison with decoding the binaries manually. This was such a popular technique back in the day that even now some applications still launch `scalap` programmatically and then parse its output to do introspection.

Below we can see the output of `scalap` for the class file produced by compiling the class Point. Note how unlike `javap`, it can merge underlying fields and associated getters into vals. Nonetheless, the correlation between the parameters of the primary constructor and the accompanying vals remains implicit. This is the artifact of how the Scala compiler internally represents constructors and has to be manually taken care of by metaprogrammers.

```
class Point extends scala.AnyRef {
  val x: scala.Int = { /* compiled code */ }
  val y: scala.Int = { /* compiled code */ }
  def this(x: scala.Int, y: scala.Int) = { /* compiled code */ }
}
```

Scala signatures are an internal, implementation-dependent mechanism to perform introspection. There is no specification of the format, no third-party implementation that reads or writes these signatures and everything there can change at the discretion of the compiler developers. Nonetheless, for a long time, using Scala signatures was the only way to reliably reflect the structure of Scala programs.

2.3 On-demand reification

In addition to automatically reifying the definition structure - in the form of both JVM-compliant metadata (section 2.1) and Scala signatures (section 2.2) - the Scala compiler is capable of selective reification of its trees and types. Exact ways of doing that have changed as Scala evolved, and in this section we will outline the earliest, pre-unification approaches. Since the goal of this chapter is to provide a brief overview, we will avoid details and will focus on simple examples revealing the main idea.

Tree reification in earlier versions of Scala (from v1.4 to v2.9) was based on type-directed code lifting. The Scala standard library used to include the magic type `scala.reflect.Code[T]` that stood for a statically typed code snippet wrapping an abstract syntax tree of type `scala.reflect.Tree`. These trees also carried symbols - data structures that provide information about resolved identifiers including their full names, types, etc.

```
class Code[T](val tree: Tree)

abstract class Tree
case class Ident(sym: Symbol) extends Tree
case class Select(qual: Tree, sym: Symbol) extends Tree
...

abstract class Symbol { def tpe: Type; ... }
abstract class LocalSymbol extends Symbol
abstract class GlobalSymbol(val fullname: String) extends Symbol
...

abstract class Type
case class SingleType(pre: Type, sym: Symbol) extends Type
case class PrefixedType(pre: Type, sym: Symbol) extends Type
...
```

Whenever the programmer was providing an expression of a regular type where an expression of type `Code[T]` was expected, the compiler would automatically transform that expression to an equivalent instance of `Code`. In the REPL session below, we can see how a simple lambda expression gets transformed into a corresponding AST wrapped in `Code[Int => Int]`. Note that, in addition to the syntax of the lambda, the compiler also reifies the fact that `x` is a local value and remembers the full name of its type.

```
scala> val id: scala.reflect.Code[Int => Int] = x => x
id: scala.reflect.Code[Int => Int] = scala.reflect.Code@147147e4

scala> id.tree
res0: scala.reflect.Tree = Function(
  List(LocalValue(NoSymbol,x,PrefixedType(...,Class(scala.Int)))),
  Ident(LocalValue(NoSymbol,x,PrefixedType(...,Class(scala.Int)))))
```

Type reification in earlier versions of Scala (from v2.7 to v2.9) was based on automatic synthesis of arguments for implicit parameters of the magic type `scala.reflect.Manifest[T]` from the Scala standard library. Unlike `Code` that wrapped an introspectable abstract syntax tree, `Manifest` was an opaque token that did not expose a possibility to inspect the exact representation of their underlying types

The listing below provides a simplified view of manifests and supporting infrastructure - the actual implementation also supports optional derivation, some basic reflective APIs as well as the capability to obtain runtime classes of the underlying types [35].

```
trait Manifest[T]

object Manifest {
  def singleType[T <: AnyRef](value: AnyRef): Manifest[T] = ...
  def classType[T](
    prefix: Manifest[_],
    clazz: Predef.Class[_],
    args: Manifest[_]*): Manifest[T] = ...
  ...
}
```

Whenever the compiler was expecting an implicit argument of type `Manifest[T]` and none was provided by the programmer, it would synthesize that argument using its internal representation of type `T`. Using the analogy between implicits and typeclasses [89], one may view this as a mechanism for automatic derivation of local instances of the `Manifest` typeclass.

In the REPL session below, we call the `manifestOf` method without providing an implicit argument of type `Manifest[T]`, and the compiler then synthesizes one for us. We can see that the compiler uses the information provided by type inference, i.e. the fact that in this method call `T` is instantiated to `List[Int]`, to generate a correct manifest.

```
scala> def manifestOf[T](x: T)(implicit m: Manifest[T]) = m
manifestOf: [T](x: T)(implicit m: Manifest[T])Manifest[T]

scala> manifestOf(List(1, 2, 3))
// The compiler generates an equivalent of:
// manifestOf(List(1, 2, 3))(
```

```
//   Manifest.classType(classOf[List],
//     Manifest.classType(classOf[Int])))
res0: Manifest[List[Int]] = scala.collection.immutable.List[Int]
```

Tree and type reification provided an interesting way to sidestep the limitations of JVM reflection and Scala signatures, but they never grew into a coherent metaprogramming API. Note how they use completely incompatible representations (`Type` vs `Manifest`) even though there is a possibility for reuse (`SingleType` and `PrefixedType` vs `Manifest.singleType` and `Manifest.classType`).

2.4 Compiler plugins

Instead of asking the Scala compiler to save fragments of its internal state and then putting these fragments together at runtime, metaprogrammers can run their programs directly in the Scala compiler, i.e. at compile time, enjoying unfettered access to program representation.

Compiler plugins [112] are user-defined libraries that introduce additional compiler phases into the standard compilation pipeline. Some compiler plugins can modify other aspects of the Scala compiler, but those that add new phases are the most common. Thanks to the compiler plugin architecture, it becomes possible to publish compiler modifications as simple compiler flags without needing to change the standard distribution. This functionality is available since v2.6.

In the listing below, inspired by [112], we can see an implementation of a very simple compiler plugin that checks programs for divisions by literal zero. This compiler plugin registers a new compiler phase `divbyzero` that is inserted into the compilation pipeline. For every compilation unit in the program, the new phase recursively traverses its abstract syntax tree, detects undesirable code patterns and produces compilation errors if such patterns are found.

```
1  import scala.tools.nsc
2  import nsc.Global
3  import nsc.Phase
4  import nsc.plugins.Plugin
5  import nsc.plugins.PluginComponent
6
7  class DivByZero(val global: Global) extends Plugin {
8    import global._
9
10   val name = "divbyzero"
11   val description = "checks for division by zero"
12   val components = List[PluginComponent](Component)
13
```

```
14    private object Component extends PluginComponent {
15      val global: DivByZero.this.global.type = DivByZero.this.global
16      val runsAfter = List("refchecks")
17      val phaseName = DivByZero.this.name
18      def newPhase(_prev: Phase) = new DivByZeroPhase(_prev)
19
20      class DivByZeroPhase(prev: Phase) extends StdPhase(prev) {
21        override def name = DivByZero.this.name
22        def apply(unit: CompilationUnit) {
23          val intType = rootMirror.getRequiredClass("scala.Int").tpe
24          unit.body.foreach {
25            case Apply(
26              Select(qual, nme.DIV),
27              List(zero @ Literal(Constant(0))))
28            if qual.tpe <:< intType =>
29              unit.error(zero.pos, "division by zero")
30            case _ =>
31              // ignore other code patterns
32          }
33        }
34      }
35    }
36  }
```

Compiler phases, like `DivByZeroPhase` that we just defined, are written against compiler internals. `scala.tools.nsc.Global` referenced on line 2 is the Scala compiler itself, so `import global._` on line 8 imports all its internal data structures and functionality working with these data structures.

`CompilationUnit` used to get the AST of a given compilation unit, `rootMirror` used to obtain a representation of type `Int`, as well as most of the APIs used in the listing - they all come from compiler internals. Explaining the entirety of compiler internals is beyond the scope of this dissertation, so we will not go into details here. Suffice it to say that `unit.body.foreach` (line 24) goes through every subtree of a tree representing a given compilation unit and matches that subtree against a pattern that represents a division (line 26) of a value of type `Int` (line 28) by a literal zero (line 27).

Just like Scala signatures (section 2.2), compiler internals lack comprehensive documentation and official compatibility guarantees. If compiler developers decide to change the AST representation of the `x/0` pattern or an API to obtain types by fully-qualified names, the DivByZero compiler plugin will stop working.

2.5 Conclusion

Metaprogramming techniques described in this chapter are pretty ad hoc in the sense that they appeared as solutions to specific problems encountered during evolution of Scala, without much effort dedicated to interoperability with solutions to similar problems.

First, Scala signatures were introduced to store signatures of compilation units, because JVM class files could not accommodate the entirety of Scala language features. Scala signatures persisted some compiler state, but not more than necessary for achieving the original goal of supporting separate compilation, and they did not have any public API, because again that was not necessary for the goal at hand.

Then, in order to experiment with providing better DSL support, Scala 1.4 gained the ability to perform selective tree reification via `Code`. Similarly to Scala signatures, abstract syntax trees used by that feature also persisted some compiler state, including symbols that represented definitions and types that represented their signatures. However, unlike Scala signatures, experimental DSLs that were using `Code` did not need to represent the entire language, so trees and their symbols ended up being limited and, therefore, not reusable in Scala signatures.

Afterwards, compiler plugins in Scala 2.6 solved another pertinent problem of allowing compiler extensions to simplify experiments with the Scala compiler. Both Scala signatures and `Code` were crippled by being useful only to their particular domains, so it was very seductive to use compiler internals as the data model and call it a day.

Finally, when the need had come to selectively overcome type erasure of the JVM around Scala 2.7/2.8, both types from `Code` and types from compiler internals were an overkill for the use cases at hand. As a result, manifests ended up using yet another data model for persisting types. Unfortunately, by the time when use cases started demanding a richer model and non-trivial reflective operations (e.g. subtyping checks), the data structures were already set in stone by existing users and could not be changed.

As we can see from this historic summary, usecase-driven evolution of metaprogramming up until Scala 2.9 has led to introduction of two groups of metaprogramming facilities in the standard distribution: 1) limited language features that support usecase-specific subsets of the language and are incompatible with each other, 2) undocumented and volatile compiler internals that support the entire language. Both of these approaches have obvious technical downsides.

In this dissertation, we explore creation of metaprogramming APIs from first principles. Instead of coming up with partial data models that fit particular use cases, we design a data model that describes the entire language. With this model at hand, we accommodate as many use cases as possible, ending up with a unified metaprogramming framework. In chapter 3 and chapter 8, we present two systems implemented in this spirit.

3 Scala.reflect

Scala.reflect is a unified metaprogramming framework that exposes a curated subset of Scala compiler internals as its public API. It is available as an experimental library in the standard distribution since v2.10.

The main innovation of scala.reflect is a detailed language model (section 3.2) that allows for a rich set of operations (section 3.5) available both at compile time and at runtime. With scala.reflect, it has become possible to subsume most of the metaprogramming facilities available in previous versions of Scala (chapter 2), sharing metaprogramming idioms and even complete metaprograms between environments. Only compiler plugins (section 2.4) that have a long tradition of working with the entirety of compiler internals escaped subsumption.

The most important use case for scala.reflect is macros (chapter 4 and chapter 5), which make it possible to author compile-time metaprograms using a rich and stable API and then seamlessly distribute such metaprograms in regular Scala libraries.

In this chapter, we provide a comprehensive overview of scala.reflect. In section 3.1, we introduce the most important concepts on a real-world example. In section 3.2 and section 3.3, we present the underlying language model. We follow up with supported environments in section 3.4 and conclude by outlining available operations enabled by the language model in section 3.5.

Scala.reflect was a massive undertaking, and it would not have happened without a joint effort of the Scala compiler team. The initial design of scala.reflect was developed and proposed by Gilles Dubochet and Martin Odersky. The initial implementation was done by Martin Odersky and Paul Phillips. Together with Martin Odersky, Jason Zaugg, Paul Phillips, Adriaan Moors and many other compiler hackers, we shipped scala.reflect, and I became its maintainer. During my tenure, I had an honor to receive contributions from many open-source enthusiasts including Denys Shabalin, Andrew Markii, Simon Ochsenreither, Dominik Gruntz, Vladimir Nikolaev and others.

3.1 Intuition

In this section, we will take a tour of scala.reflect. First, we will start with the architecture, overviewing code organization patterns (subsection 3.1.1). Afterwards, we will take an example of a metaprogram written using scala.reflect, explaining how it works (subsection 3.1.2) and how it can be executed in different environments (subsection 3.1.3, subsection 3.1.4 and subsection 3.1.5). During the tour, we will intentionally limit ourselves to basic comments about scala.reflect APIs, because the details are covered further in this chapter.

3.1.1 High-level architecture

The main design goal of scala.reflect was to expose compiler internals in a public metaprogramming framework. In theory, this looks like a great way to build a comprehensive API. After all, the compiler already has a powerful language model and a set of operations to go with it. In practice, however, this presents a significant engineering challenge.

Even in pre-scala.reflect days, the compiler was already exceeding 100k lines of code. As a result, publishing them in their entirety was out of the question, because even with extensive documentation, understanding such a metaprogramming API would be too hard to be practical. Additionally, publishing everything would turn compiler internals into an API, significantly complicating development of new functionality.

Therefore, we decided to expose only the core data structures and operations of compiler internals, encapsulating implementation details behind a public API. In order to ship scala.reflect within a reasonable timeframe, we had to minimize the amount of changes to the compiler required to accommodate this API, and that led to peculiar decisions regarding the design of scala.reflect.

To be more concrete, let us consider an excerpt of the sources of the Scala compiler from the period that preceded unification. The listing below is somewhat adapted for illustrative purposes, but the high-level architecture is depicted exactly as it is implemented in the original code.

```
1  class Global(var settings: Settings, var reporter: Reporter)
2  extends SymbolTable {
3     def compile(filenames: List[String]) { ... }
4     ...
5  }
6
7  abstract class SymbolTable extends Trees
8                                with Symbols
9                                with Names
10                               with ...
```

```
11
12 trait Trees {
13   self: SymbolTable =>
14
15   var nodeCount = 0
16
17   abstract class Tree {
18     val id = nodeCount
19     nodeCount += 1
20
21     def symbol: Symbol = null
22     def symbol_=(sym: Symbol): Unit = { ... }
23
24     // a couple dozen additional methods
25     ...
26   }
27
28   case class Ident(name: Name) extends Tree
29
30   ...
31 }
```

`Global` is the class that implements the Scala compiler. The `scalac` script provided with the standard distribution instantiates `Global` and calls `compile` on the files passed on the command line, launching the compilation pipeline.

Internally, `Global` extends `SymbolTable`, which is built according to the cake pattern, a way of organizing code in which a complicated module, i.e. a cake (in this case, `SymbolTable`), is modelled as a mixin of traits, i.e. slices (in this case, `Trees`, `Symbols`, `Names`, etc).

Slices of the cake can be made interdependent via self-type annotations. An example of a self-type annotation can be found on on line 13. It says that any object of type `Trees` must also be of type `SymbolTable`. This means that inside the definition of `Trees`, we can utilize all data structures from `SymbolTable`, e.g. `Symbol` that comes in handy on lines 21-22 and `Name` that is used on line 29. (Definitions of `Symbol` and `Name` are not provided in the listing, but, as one may guess, these definitions are located in other slices of the cake - `Symbols` and `Names` respectively).

From this listing, it becomes clear that creating a public metaprogramming API for compiler internals is far more complicated than simply moving some data structures from the compiler to a public package and creating interfaces that describe a curated set of operations. Unfortunately, even the data structures used in the compiler are too complicated to be published as is.

For example, it is highly unlikely that users of scala.reflect will often make use of the fact that every tree has a unique id, so `Tree.id` should better be left out of a public metaprogramming API. Also, even if `Tree.id` were acceptable as a public API, it relies on `nodeCount` (line 15), compiler-specific mutable state, that can not be moved outside the compiler. Both problems are quite pervasive and apply to a significant number of other data structures.

While one can imagine a compiler refactoring that would streamline data structures and encapsulate mutable state, it is virtually impossible to pull off such a refactoring in the Scala compiler, an actively developed project with numerous industrial users. Potential loss of development time and risk of regressions are very hard to justify. Therefore, we had to create public interfaces for the data structures.

While creating public interfaces, we again had to avoid significant changes to the compiler codebase, which invalidated a straightforward object-oriented approach summarized in the listing below.

```
1  package scala.reflect.api {
2    trait TreeApi {
3      def symbol: SymbolApi
4      def symbol_=(sym: SymbolApi): Unit
5      ...
6    }
7  }
8
9  package scala.tools.nsc.symtab {
10   trait Trees {
11     self: SymbolTable =>
12
13     abstract class Tree extends TreeApi {
14       def symbol: Symbol = null
15
16       def symbol_=(sym: Symbol): Unit = { ... }
17       def symbol_=(sym: SymbolApi): Unit = {
18         sym match {
19           case sym: Symbol => symbol_=(sym)
20           case _ => sys.error("incompatible symbol " + sym)
21         }
22       }
23       ...
24     }
25
26     ...
27   }
28 }
```

In this example, we limit the API surface to trees with two methods: the `symbol` getter that contains a reflection artifact in an output position (line 3) and the `symbol_=` setter that contains a reflection artifact in an input position (line 4).

Compiler internals contain similar methods, but reflection artifacts in those methods have more specific types, which leads to a problem. It is okay to implement a less specific getter (line 3) with a more specific getter (line 14). However, it is not okay to implement a setter that takes a less specific parameter (line 4) with a setter that takes a more specific parameter (line 16). Indeed, the parameter of the less specific setter can be created in a different instance of `SymbolTable`, while the more specific setter works only with symbols from the same `SymbolTable`.

While the listing above provides a workaround for the problem using method overloading, the situation is unsatisfying for multiple reasons. First, this workaround shows that our API is lacking - it does not statically prevent mixing artifacts from different instances of `SymbolTable`. Secondly, even if we assume that we are fine mixup errors manifesting at runtime not at compile time, the workaround complicates compiler internals. Introducing an overload is not guaranteed to be a source-compatible change, because of effects that it can have on type inference, method signature erasure, eta expansion, named/default parameters, etc. These problems can be worked around with more tricks, but this quickly overflows the acceptable complexity budget for compiler changes.

In his dissertation [35], Dubochet explains how to create a specially crafted interface to compiler internals that hides most implementation details and requires minimal changes to the existing compiler codebase to accommodate it. This design utilizes abstract type members to represent data structures and then makes use of Scala's support for object construction and deconstruction via `apply` and `unapply` methods. Below we can see a simplified fragment of the `scala.reflect.api` package that puts this idea into practice and defines an API for our running example.

```
1  abstract class Universe extends Trees
2                              with Symbols
3                              with Names
4                              ...
5
6  trait Trees {
7    self: Universe =>
8
9    type Tree >: Null <: TreeApi
10   trait TreeApi {
11     def symbol: Symbol
12     def symbol_=(sym: Symbol): Unit
13     ...
14   }
15
```

```
16    type Ident >: Null <: Tree with IdentApi
17    trait IdentApi extends TreeApi {
18      def name: Name
19    }
20
21    val Ident: IdentExtractor
22    abstract class IdentExtractor {
23      def apply(name: Name): Ident
24      def unapply(ident: Ident): Option[Name]
25    }
26
27    ...
28  }
```

The `scala.reflect.api.Universe` class is the gateway into the scala.reflect API. Much like compiler internals, it is also based on the cake pattern. As we will also see in subsection 3.1.5, cakes beget cakes, and scala.reflect has not been able to escape this fate.

Inside the slices of the `Universe` cake, we specify abstract elements or the language model, e.g. `Tree`, `Symbol` and `Name` as abstract types (see line 9 for the `Tree` type). Concrete elements, e.g. `Ident`, are specified as pairs of definitions that include an abstract type (see line 16 for the `Ident` type) and an abstract val (see line 21 for the `Ident` val). The former allows users of the language model to refer to the type of `Ident`, while the latter allows to instantiate and pattern match on instances of `Ident` as shown in the listing below.

```
val universe: Universe = ...
import universe._

val ident = Ident(TermName("x"))
println("printing Ident(TermName(" + ident.name.toString + "))"
printTree(ident)

def printTree(tree: Tree) = tree match {
  case Ident(name) => println(name.toString)
  ...
}
```

Let us go through the example, keeping note of how common language idioms map onto the proposed design. First, we create an identifier by calling `Ident(Name("x"))`. According to the rules of desugaring in Scala, this invocation translates into `Ident.apply(Name("x"))`, going through the abstract val and its `Extractor` interface.

Afterwards, we take the created identifier and call `ident.name`. That works because the type returned by `Ident.apply` is the abstract type `Ident` whose upper bound includes a member called `name` via its `Api` interface.

Then, we pass the identifier to `printTree`, whose parameter is typed as `Tree`, another abstract type. That works, too, again because of the upper bound of `Ident` that in addition to `IdentApi` includes `Tree`.

Finally, we pattern match identifiers via `case Ident(name)`, which, again by the rules of desugaring in Scala, is equivalent to `Ident.unapply(tree)`. Since the result type of `Ident.unapply` says `Option[Name]`, it means that the result of a successful match is a single value of type `Name`.

This way of organizing the scala.reflect API may look very strange. Instead of declaring `Ident` as a one-liner `case class Ident(name: Name) extends Tree`, we go through the trouble of emulating the case class with ten times more code that, additionally, uses advanced language features. However, the trouble is arguably worth it, because the changeset required for the compiler to adopt such an API is fairly minimal.

```
1  abstract class SymbolTable extends Universe
2                                with Trees
3                                with Symbols
4                                with Names
5                                with ...
6
7  trait Trees {
8    self: SymbolTable =>
9
10   abstract class Tree extends TreeApi {
11     def symbol: Symbol = null
12     def symbol_=(sym: Symbol): Unit = { ... }
13     ...
14   }
15
16   case class Ident(name: Name) extends Tree with IdentApi
17   object Ident extends IdentExtractor
18
19   ...
20 }
```

For every abstract class that we publish in the language model, we add the corresponding `Api` to its list of parents. For every concrete class, we additionally define a companion and add the corresponding `Extractor` to its list of parents. Then we take `SymbolTable` and add `Universe` to its parent list. And, without changing any existing methods or adding more definitions to compiler internals, we are done.

By the virtue of having the same name as abstract types in the API cake and conforming to their `Api` interfaces, classes in the internal cake implement those abstract types. This means that all occurrences of these types in the API cake are going to automatically

mean their concrete implementations when viewed from the internal cake. This is how `symbol` and `symbol_=` on lines 11-12 implement their counterparts in the API without running into the problem with described above.

The same goes for companion objects of concrete classes like `Ident` that implement the abstract vals from the API cake. Because of virtual dispatch, `Ident.apply` and `Ident.unapply` in the code that uses scala.reflect API, are going to be redirected to the `apply` and `unapply` methods automatically generated by the case class infrastructure in scala.reflect implementation.

To put it in a nutshell, there is a good reason why the scala.reflect API is imported from universes instead of being available on the top level. The main design goal of exposing Scala compiler internals in a public metaprogramming API, combined with the necessity to avoid high-impact refactorings in the compiler, pretty much dictated this architecture as the only viable choice.

3.1.2 Example: the isImmutable metaprogram

We call a value immutable if it cannot be modified and all its fields themselves have immutable types. In this section, we will use scala.reflect to define a method called `isImmutable` that checks whether a given type is immutable, i.e. whether all its values are guaranteed to be immutable.

For example, the immutability check on `class C(x: Int)` will fail, because someone can create the subclass of `C` and add mutable state to that subclass. Continuing our example, if we add `final` to the definition of class `C`, the immutability check will succeed, because now values of type `C` must be instances of class `C`, and the only field of that class is immutable and has primitive type.

Strictly speaking, even immutable fields can be modified by JVM reflection as demonstrated in section 2.1, which means that `isImmutable` can only really succeed on primitives. However, such use of JVM reflection is strongly discouraged in the Scala community, therefore, without the loss of usefulness of our immutability check, we will assume that immutable fields are allowed.

```
1 def isImmutable(t: Type): Boolean = {
2   val cache =
3     scala.collection.mutable.Map[Type, Boolean]().
4     withEquality(_ =:= _)
5
6   def uncached(t: Type): Boolean = {
7     t match {
8       case AnnotatedType(annots, t) =>
9         cached(t)
```

```
10        case ExistentialType(defns, t) =>
11          cached(t)
12        case RefinedType(parents, defns) =>
13          parents.exists(cached)
14        case _: SingletonType =>
15          cached(t.widen)
16        case TypeRef(_, sym, args) if sym == definitions.ArrayClass =>
17          false
18        case TypeRef(_, sym: ClassSymbol, args) =>
19          if (sym.isFinal || sym.isModuleClass) {
20            val fieldTypes =
21              t.members.collect {
22                case s: TermSymbol if !s.isMethod =>
23                  if (s.isVar) return false
24                  s.typeSignatureIn(t)
25              }
26            fieldTypes.forall(cached)
27          } else {
28            false
29          }
30        case TypeRef(_, sym: TypeSymbol, args) =>
31          val TypeBounds(_, hi) = sym.typeSignature.finalResultType
32          cached(hi.substituteTypes(sym.typeParams, args))
33        case _ =>
34          sys.error("unsupported type: " + t)
35      }
36    }
37
38    def cached(t: Type) = {
39      cache.getOrElseUpdate(t, { cache(t) = true; uncached(t) })
40    }
41
42    cached(t)
43  }
```

The happy path of our algorithm happens on lines 18-29. First, we check whether the given type refers to a final class or to an object (objects are singletons, so they are implicitly final). Because of the way how compiler internals are organized, these are two different checks: `isFinal` and `isModuleClass` respectively.

Afterwards, we go through all members of the type, which are represented by data structures called symbols (see subsection 3.2.3 for details). While iterating through members, we only look into those that define terms, skipping methods (because methods do not define state), terminating execution on `var` and recursively checking types of vals and nested objects.

Now let us get into nitty-gritty details. In our experience, metaprogramming in Scala makes comprehensive handling of corner cases unusually hard. The language model is quite sizeable, so it takes a while to ensure that all possibilities are covered. `isImmutable` is no exception - its full code is almost four times bigger than its happy path.

First, we create the infrastructure to avoid infinite recursion (lines 38-40). We memoize recursive calls in a custom map (lines 2-4) that accounts for the fact that comparing types for equality need a special equality check (`=:=` as opposed to the standard `==` typically used in Scala). In order to handle circular dependencies, i.e. situations when a class A has a field of type B and a class B has a field of type A, we postulate everything be immutable unless proven otherwise (`cache(t) = true` on line 39).

Afterwards, we go through all flavors of types that are supported by Scala, and make sure that our algorithm works correctly on them.

On lines 7-32, we handle types that classify values. Along with such types, the scala.reflect language model also features others, e.g. `MethodType` (a type that encodes method signatures and contains information about parameters and return types) and `WildcardType` (a type that encodes an unknown type during type inference). Immutability check does not make sense for types that do not classify values, so we just error out when encountering them (lines 33-34).

Annotated types (`T @annotation`) and existential types (`T forSome { ... }`) are trivially unwrapped and processed recursively.

Refined types (`T with U { ... }`) are immutable as long as any parent is immutable. If, for such type, one of the parents is immutable (i.e. final), this means that there can not exist any class, apart from such parent, whose instances conform to such type. Therefore, the refined type is either equivalent to that parent (if the parent conforms to the type) or is uninhabited. In both cases, the immutability check succeeds (lines 12-13).

Singleton types (`x.type`, `this.type` and `42.type`) are immutable if and only if the type of the singleton is immutable. In order to check the underlying type, we call `Type.widen` and then recur (lines 14-15).

Finally, there are typerefs, which represent references to type definitions (`foo.bar.Baz` and `Foo#Bar`), possibly applied to type arguments. There are two fundamental possibilities here. The type definition can be: 1) a class, a trait or a module (lines 16-29), or 2) a type member or a type parameter (lines 30-32).

The first case is already handled by the happy path, except for value types and arrays. These special types are modelled as classes, but are not represented as classes at runtime. Since these types are final and have no fields, they will be deemed immutable. That is correct for primitives, but wrong for arrays, so we hardcode arrays on lines 16-17.

The second case should have been part of the happy path, but, due to an unfortunate peculiarity of compiler internals, `typeSignatureIn` sometimes fails to work correctly for members of such types.

Luckily, the workaround is quite simple conceptually. Without loss of generality, we can replace a type that refers to a type member or type parameter with its upper bound and then check that upper bound recursively. Unfortunately, implementing this idea also requires knowledge of compiler internals (lines 31-32).

This analysis covers the full spectrum of Scala types, completing the implementation of `isImmutable`. As we will see in later sections, `isImmutable` can run both at compile time and at runtime using facilities provided by the standard distribution of Scala.

3.1.3 Compile-time execution

In the listing below, we define a trait `Immutable[T]` and a macro `materialize[T]` that generates a value of type `Immutable[T]` if its type argument is immutable and fails with an error otherwise.

Since our goal in this section is to highlight the most important aspects of scala.reflect, below we only provide a brief description of how the macro works. We refer curious readers to chapter 4 for a full explanation of underlying mechanisms.

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.blackbox.Context
3
4  trait Immutable[T]
5  object Immutable {
6    implicit def materialize[T]: Immutable[T] = macro Macros.impl[T]
7  }
8
9  object Macros {
10   def impl[T](c: Context)(implicit tag: c.WeakTypeTag[T]) = {
11     import c.universe._
12
13     def isImmutable(t: Type): Boolean = {
14       // source code taken from subsection 3.1.2
15       ...
16     }
17
18     val t = tag.tpe
19     if (isImmutable(t)) q"null"
20     else c.abort(c.enclosingPosition, t + " is not immutable")
21   }
22 }
```

On line 6, `Immutable.materialize[T]` defines a macro, i.e. a facade for the metaprogram `Macros.impl[T]` (line 10) that will run inside the compiler every time the typechecker encounters a call to `materialize`. Since macros are an experimental language feature, they require a special import (line 1).

The metaprogram `impl` takes a compiler context (line 2) that, among other things, wraps `universe`, a compile-time implementation of scala.reflect provided by the Scala compiler. Therefore, importing `c.universe._` on line 11 brings the entire scala.reflect API in scope, providing definitions like `Type`, `ClassSymbol`, `TypeSymbol` and others necessary for `isImmutable` to operate (lines 13-15).

In addition to the context, `impl` receives a type tag (line 10) that wraps the representation of the type argument of the corresponding call to `materialize`. After unwrapping the type tag on line 17, we check it for immutability.

If the immutability check succeeds, macro expansion succeeds by producing a trivial instance of `Immutable` on line 18. Since `Immutable` is just a marker, and there are no methods to call on it, we return null to avoid runtime performance costs of instantiating and then garbage collecting dummy instances of `Immutable`. `q"null"` used here is a quasiquote, a notation to create abstract syntax trees from snippets of Scala code (refer to subsection 3.3.2 to learn more about quasiquotes).

If the immutatibility check fails, macro expansion fails by calling `c.abort` on line 19, which will result in a compilation error. This error will be positioned at the callsite of the macro, specified by `c.enclosingPosition`, and will provide a helpful error message. The capability to produce domain-specific errors has proven to be one of the strong points of macros.

Let us put the `materialize` macro to good use. Suppose we have a slow algorithm `compute` that takes an input configuration and then runs for a while, probably launching new threads running in parallel with the main program.

```
trait Config {
  def param1: Parameter
  ...
}

def compute(c: Config) = { ... }
```

Now, we want to make sure that, while the algorithm is running, noone can modify its configuration from a different thread. In order to guarantee that, we require callers to provide an evidence that the configuration is immutable.

```
def compute[C <: Config](c: C)(implicit ev: Immutable[C]) = { ... }
```

Thanks to the mechanism of implicits [89], whenever the programmer does not provide the evidence manually (which is typical, because writing evidences by hand is very tedious), the compiler will insert the call to the `materialize` macro that will validate the fact that the static type of the configuration is immutable by running `isImmutable`. If the immutability check fails, a compile-time error will be issued by the macro.

```
final case class MyConfig(param1: Parameter) extends Config
val myConfig: MyConfig = obtainConfig()
compute(myConfig)
// equivalent to: compute(myConfig)(Immutable.materialize[MyConfig])
```

The technique of programmatic generation of implicit arguments that we explored in this toy example is actually very useful in practice. Its main applications lie in the area of generic programming [84], and it powers several macros that are cornerstone to the state of the art in the open-source Scala community. This and other popular use cases for macros are discussed in chapter 6.

3.1.4 Runtime execution

In this section, we continue the example from subsection 3.1.3 with an additional restriction that `compute` no longer knows static types of its input configurations. For example, let us imagine that configurations are now dynamically deserialized from binary payload. In such situation, we can still make use of `isImmutable` as demonstrated in the listing below.

```
1  def compute(c: Config) = {
2    import scala.reflect.runtime.universe._
3
4    def isImmutable(t: Type): Boolean = {
5      // source code taken from subsection 3.1.2
6      ...
7    }
8
9    val mirror = runtimeMirror(c.getClass.getClassLoader)
10   val t = mirror.classSymbol(c.getClass).toType
11   if (!isImmutable(t)) sys.error(t + " is not immutable")
12
13   ...
14 }
```

In the `scala.reflect.runtime` package, scala.reflect provides a runtime implementation of the scala.reflect API, capable of running on top of JVM reflection. This implementation can launch a big chunk of the Scala compiler at runtime and then use it to expose scala.reflect-compatible functionality.

Much like in subsection 3.1.3, where we imported `c.universe._` and gained access to the scala.reflect API at compile time, on line 2 we import `scala.reflect.runtime.universe._` and become able to use the very same scala.reflect API, but at runtime.

There are minor differences between the functionality available during compilation and at runtime. On the one hand, `c.abort` and some other macro APIs (section 4.5) only make sense within a compiler environment. On the other hand, runtime reflection relies on runtime classes (section 2.1) that may be unavailable during compilation. Nonetheless, most scala.reflect APIs, including types and symbols that are necessary to run `isImmutable` are independent of an environment.

Now, when we know how to execute `isImmutable` at runtime, let us figure out how to obtain a scala.reflect type from a given config object in order to get the immutability check going.

When `isImmutable` was running inside the compiler, the entire environment was working in terms of scala.reflect, because scala.reflect is based on compiler internals. At runtime, introspection happens in terms of the JVM object model, so we need to adapt it to the scala.reflect way.

In order to do that, on line 9, we create a scala.reflect mirror based on a class loader, which is an entity that encapsulates a JVM environment. Afterwards, on line 10, we use this mirror to convert the JVM type of the config to a scala.reflect type.

Unfortunately for our use case, Scala and therefore scala.reflect use a type system that is much richer than the type system of the JVM. The only types available for introspection on the JVM are primitives, generic arrays, as well as non-generic classes and interfaces. As a result, the type extracted from the config object is going to be imprecise, with the most important problem of lacking potential type arguments that are erased at runtime. In the listing below, we illustrate this principle on a series of examples in a REPL session.

```
1  scala> final class Metadata { ... }
2  defined class Metadata
3
4  scala> final class MyConfig[T](payload: T, metadata: List[Metadata])
5  defined class MyConfig
6
7  scala> val c = new MyConfig(42, Nil)
8  c: MyConfig[Int] = MyConfig@784c0b8
9
10 scala> c.getClass
11 res0: Class[_ <: MyConfig[Int]] = class MyConfig
12
13 scala> import scala.reflect.runtime.universe._
14 import scala.reflect.runtime.universe._
```

```
15
16 scala> val mirror = runtimeMirror(c.getClass.getClassLoader)
17 mirror: Mirror = ...
18
19 scala> val t = mirror.classSymbol(c.getClass).toType
20 t: Type = MyConfig
21
22 scala> t.member(TermName("payload")).typeSignatureIn(t)
23 res2: Type = => T
24
25 scala> t.member(TermName("metadata")).typeSignatureIn(t)
26 res3: Type = scala.List[Metadata]
```

As it can be seen on lines 10-11, even though the Scala type of the config is `MyConfig[Int]`, its JVM type is just `MyConfig`. Therefore, after we go back from a JVM type to the Scala type on lines 16-17, the resulting Scala type is also just `MyConfig`.

Consequently, on lines 22-23, the type of `c.payload` is calculated as `=> T`, meaning "a getter that returns a T" where `T` is the type parameter of `MyConfig`. As a result, the immutability check for the config will fail, because in the general case `T` can be anything, including a mutable type.

On the bright side, once we get into the realm of Scala types, we can continue operating in terms of Scala types thanks to Scala signatures (section 2.2). Therefore, on lines 25-26, the type of `c.metadata`, which does not depend on `T`, is actually precise, saying `scala.List[Metadata]`.

Type erasure of the JVM can be selectively overcome with type tags by applying manual annotations on case-by-case basis. In the short example below, we explain the basics of this approach. For more details, refer to subsection 3.3.3.

```
1  scala> import scala.reflect.runtime.universe.TypeTag
2  import scala.reflect.runtime.universe.TypeTag
3
4  scala> final case class MyConfig[T](payload: T)(
5       | implicit val tag: TypeTag[MyConfig[T]])
6  defined class MyConfig
7
8  scala> val c = new MyConfig(42)
9  // equivalent to:
10 // new MyConfig(42)(scala.reflect.api.materializeTypeTag[...])
11 c: MyConfig[Int] = MyConfig(42)
12
13 scala> c.getClass
14 res0: Class[_ <: MyConfig[Int]] = class MyConfig
15
```

```
16 scala> val t = c.tag.tpe
17 t: Type = MyConfig[Int]
18
19 scala> t.member(TermName("payload")).typeSignatureIn(t)
20 res1: Type = => Int
```

In order to retain the static type of the config, we add an implicit `TypeTag` parameter wrapping the corresponding type. In the spirit of the technique of implicit materialization, implicit arguments for this parameter can be derived by the compiler, remembering the original scala.reflect type. As we can see on lines 16-20, the type obtained from the config now correctly has a type argument and its `payload` is now correctly typed as an `Int`. This precision comes at a price of additional memory pressure created by a new field in `MyConfig` and instances of `TypeTag` and additional code generated for every instantiation of `MyConfig`, but sometimes better introspection capabilities may be more important than the imposed performance penalty.

This wraps up the overview of scala.reflect on the JVM. Even though there is an abstraction gap between the theoretical language model of Scala and the actual environment of the JVM, scala.reflect does its best to bridge this gap. Information obtained from dynamic values may be incomplete because of type erasure mandated by the JVM, but static program structure is available with full fidelity.

Apart from the JVM, Scala also supports other platforms, namely, JavaScript and native code. We have not implemented scala.reflect for them and doubt that it will ever be possible to do that (subsection 3.4.5 and subsection 3.4.6), which means that running `isImmutable` in those environments is most likely out of the question.

As a result of limited support of scala.reflect on the JVM and non-existent support for other environments, there is a strong tendency in the Scala community to attempt to avoid runtime introspection in favor of compile-time metaprogramming. Therefore, the main part of the value proposition of the scala.reflect framework comes from enabling macros (chapter 4 and section 5.3).

3.1.5 Sharing metaprograms

Sharing metaprograms was stated as one of the goals for a unified metaprogramming API, however until now (in subsection 3.1.3 and subsection 3.1.4) we have not really shared any metaprograms. What we did was copy and paste `isImmutable` between environments, achieving duplication not sharing.

In this section, we finally discuss how to write scala.reflect metaprograms that can be called from different independent environments. In the listing below, we take `isImmutable` from subsection 3.1.2 and apply several small changes.

```
import scala.reflect.api.Universe

def isImmutable(u: Universe)(t: u.Type): Boolean = {
  import u._

  // the rest of the code taken from subsection 3.1.2
  ...
}
```

First, we add a new parameter of type `Universe` to the signature of `isImmutable`. Afterwards, we change the type of the original parameter from `Type` to `u.Type`, because as we have learned in subsection 3.1.1, elements of the scala.reflect language model are defined inside universes as opposed to being independent top-level classes. Finally, we `import u._` to bring necessary APIs like `Type`, `TypeSymbol` and `ClassSymbol` in scope in the body of the method.

This presents is an interesting asymmetry between how the scala.reflect API is consumed from inside and from outside the universe cakes. Inside a universe (see examples from subsection 3.1.1), all data structures and operations are available without qualification. Outside a universe, i.e. in user metaprograms, we have to carry a universe around and explicitly qualify reflection artifacts, which becomes a significant notational inconvenience.

In order to work around the necessity to pass universes around, users of scala.reflect typically create their own mini-cakes parameterized by a universe that is then imported into a scope shared by multiple helpers. Here is how an example from subsection 3.1.3 can be rewritten using this technique.

```
1  // the rest of the code taken from subsection 3.1.2
2
3  object Macros {
4    def impl[T](c: Context)(implicit tag: c.WeakTypeTag[T]) = {
5      import c.universe._
6      val helpers = new Helpers(c.universe)
7      import helpers._
8
9      val t = tag.tpe
10     if (isImmutable(t)) q"null"
11     else c.abort(c.enclosingPosition, t + " is not immutable")
12   }
13 }
14
15 class Helpers(u: Universe) {
16   import u._
17
18   def isImmutable(t: Type): Boolean = {
19     ...
```

```
20    }
21
22    // can add more helpers to the Helpers class
23    // and none of them would need to declare a universe of their own
24  }
```

This looks like an acceptable workaround, because the cost of adding a universe parameter and then importing it is paid only once per a library of helpers than can then be made arbitrarily big. Unfortunately, the use of this particular helper class is going to produce the following compilation error:

```
Macros.scala:10: error: type mismatch;
 found   : c.universe.Type
 required: helpers.u.Type
    if (isImmutable(t)) q"null"
                    ^
```

This error is emitted because the Scala compiler can not remember the fact that `helpers` was created from `c.universe`. This happens because the compiler infers type `Helpers` for `helpers` on line 6, and that type does not carry any indication of where `helpers.u` comes from. For a brief period of time, we considered tightening the typechecker to keep this kind of information around. However, such a change may potentially lead to non-obvious breakages in existing code, so in the end we decided against it.

As a result, metaprogrammers have to manually let the typechecker know about the type of `helpers`, which should be `Helpers { val u: c.universe.type }`. This is quite a mouthful, so we have come up with a workaround that completely avoids the need to spell out the right type and forces the typechecker to infer something analogous instead.

```
1  object Macros {
2    def impl[T](c: Context)(implicit tag: c.WeakTypeTag[T]) = {
3      val helpers = new Helpers[c.universe.type](c.universe)
4      import helpers._
5      ...
6    }
7  }
8
9  class Helpers[U <: Universe](u: U) {
10   ...
11 }
```

Here, the type of `helpers` ends up being `Helpers[c.universe.type]`, which means that `helpers.u` is `c.universe.type`, as necessary to satisfy the typechecker. We have to explicitly specify the type argument of the constructor of `Helpers` on line 3, because otherwise the compiler will again infer a non-specific type for it, defeating the whole purpose of the workaround.

This notational inconvenience associated with the particular way how scala.reflect is organized may not seem like a serious problem, but in practice it has proven to be very significant.

In Scala 2.10, when we added macros to the language, associated metaprograms could only be defined as methods similar to `Macros.impl` from the examples above. As a result, metaprogrammers had to work their way through understanding the error messages and writing necessary boilerplate to help out the compiler. Before the release, we wrote documentation explaining the issue and the workaround, and called it a day.

However, shortly afterwards we faced overwhelming user feedback that forced us to reconsider. For one, understanding the issue and the workaround requires pretty good command of the Scala language, so our solution was suboptimal from the educational point of view. Moreover, the workaround feels suboptimal in the sense that it requires boilerplate that can not be abstracted away and has to be repeated almost verbatim for every declared macro.

Therefore, in Scala 2.11, we introduced another way to write macro-compatible metaprograms. On a very high level, it allows macros to directly reference methods inside `Helpers`-like classes, avoiding the necessity for the workaround. Additional details on this language feature can be found in subsection 4.3.4.

As we can see from the discussion above, scala.reflect can indeed share metaprograms between environments, qualifying for the requirements of a unified metaprogramming framework. Unfortunately, architectural peculiarities explained in subsection 3.1.1 make such sharing unsatisfyingly verbose.

3.2 Language model

Now that we have become familiar with the organization of scala.reflect (subsection 3.1.1) and have seen the three main data structures of scala.reflect - trees, symbols and types - working together (subsection 3.1.2), we proceed to the more detailed part of the overview of the scala.reflect framework.

In this section, we will see how scala.reflect represents Scala programs using its language model. Our goal here will not be to provide a comprehensive list of data structures, but to outline the most important ones and highlight their relations with each other. Moreover, for the sake of clarity, we will not be covering operations available for the elements of the model, leaving that to section 3.5.

3.2.1 Overview

The language model of scala.reflect is based on compiler internals of `scalac`, so one of the best ways to understand the details of scala.reflect is to observe the Scala compiler at work and try to understand what is happening. To that end, we will now compile a simple Scala program with private compiler flags that ask `scalac` to print its state to console after certain phases of the compiler pipeline.

Here is the program that we type in a file and save to disk as `Test.scala`. Despite being seemingly trivial, it illustrates plenty of peculiarities in how the Scala compiler represents and processes programs.

```
object Test {
  // create a list and compute its head
  val xs = List(42)
  val head: Int = xs.head
  println((xs, head))
}
```

In order to compel `scalac` to print snapshots of its abstract syntax trees, we use `-Xprint:parser` and `-Xprint:typer`. The `-Yshow-trees-compact` and `-Yshow-trees-stringified` flags tell the compiler that we are interested in both stringified, i.e. prettyprinted, and compact raw representations of the ASTs. Finally, `-uniqid` make the compiler highlight trees that carry symbols. We could also use `-Xprint-types` to print types of typechecked ASTs, but then we would get overwhelmed.

```
$ scalac -Xprint:parser -Xprint:typer -uniqid\
> -Yshow-trees-compact -Yshow-trees-stringified Test.scala
```

The first part of the printout demonstrates abstract syntax trees after the `parser` phase. At that point, the Scala compiler has just parsed the source files and has not yet started typechecking. Below we see a prettyprinted representation of `Test.scala` which is followed by a nested sequence of data constructors detailing the underlying AST nodes.

```
1  [[syntax trees at end of parser]]// Scala source: Test.scala
2  package <empty> {
3    object Test extends scala#24.AnyRef {
4      def <init>() = {
5        super.<init>();
6        ()
7      };
8      val xs = List(42);
9      val head: Int = xs.head;
10     println(scala#24.Tuple2(xs, head))
11   }
12 }
```

```
13 PackageDef(Ident(TermName("<empty>")), List(
14   ModuleDef(Modifiers(), TermName("Test"), Template(
15     List(Select(Ident(scala#24), TypeName("AnyRef"))),
16     noSelfType,
17     List(
18       DefDef(
19         Modifiers(), termNames.CONSTRUCTOR,
20         List(), List(List()), TypeTree(),
21         Block(List(pendingSuperCall), Literal(Constant(())))),
22       ValDef(
23         Modifiers(), TermName("xs"),
24         TypeTree(),
25         Apply(Ident(TermName("List")), List(Literal(Constant(42))))),
26       ValDef(
27         Modifiers(), TermName("head"),
28         Ident(TypeName("Int")),
29         Select(Ident(TermName("xs")), TermName("head"))),
30       Apply(
31         Ident(TermName("println")),
32         List(Apply(
33           Select(Ident(scala#24), TermName("Tuple2")),
34           List(Ident(TermName("xs")), Ident(TermName("head")))))))
35 ))))
36
37 ...
```

One peculiarity of compiler internals that immediately catches the eye is that the Scala compiler aggressively desugars programs into a normalized representation. Concretely, even in this simple program, `scalac`: 1) drops original comments and formatting, 2) wraps `Test` in a synthetic package (line 2), 3) makes `Test` inherit from `AnyRef` (line 3), 3) creates a synthetic constructor for `Test` (lines 4-7), 4) transforms a tuple literal into a synthetic call to `Tuple2` (line 10).

In the detailed part of the printout, we can see many different AST nodes, including `PackageDef`, `ModuleDef`, `Template` and others. Some fragments of the printout, namely `Modifiers(...)`, `TermName(...)` and `TypeName(...)`, are actually auxiliary data structures that are not ASTs. For example, names are cached objects that come from internal name pools that are introduced to improve compiler performance.

In addition to trees, the code above also includes symbols. We can see `scala#24` appearing several times in both prettyprinted and raw representations of the program, and the `#` part of it indicates that we have a tree that carries a symbol. In a nutshell, symbols are unique identifiers that `scalac` uses to establish bindings. In this case, `scalac` uses symbols to make sure that the synthetic identifier `scala` references the standard `scala` package regardless of the definitions that the programmer has in scope.

The second part of the printout shows the result of the `typer` phase. This phase is the last phase of the Scala compiler frontend, so by the time it finishes, all trees are attributed and ready to enter the pipeline that will progressively lower them and emit executable code.

```
35 ...
36
37 [[syntax trees at end of typer]]// Scala source: Test.scala
38 package <empty>#4 {
39   object Test#7854 extends scala#24.AnyRef#2765 {
40     def <init>#7942(): Test#7855.type = {
41       Test#7855.super.<init>#3112();
42       ()
43     };
44
45     private[this] val xs#7944: List#8198[Int#1850] =
46       immutable#5861.this.List#8199.apply#9078[Int#1850](42);
47     <stable> <accessor> def xs#7943: List#8198[Int#1850] =
48       Test#7855.this.xs#7944;
49
50     private[this] val head#7946: Int#1850 =
51       Test#7855.this.xs#7943.head#19914;
52     <stable> <accessor> def head#7945: Int#1850 =
53       Test#7855.this.head#7946;
54
55     scala#25.this.Predef#2088.println#7621(
56       scala#24.Tuple2#2649.apply#20019[List#8198[Int#1850], Int#1850](
57         Test#7855.this.xs#7943, Test#7855.this.head#7945))
58   }
59 }
60 ...
```

The compiler has done quite some work in the frontend. The code has been heavily transformed, so we had to reformat the prettyprinted output and to completely hide the raw representation of the resulting trees in order to stay readable.

Again, it immediately catches the eye that the frontend has performed further desugaring. In this case, it is: 1) transformation of vals into pairs of a private underlying field / public getter (lines 44-47, 49-52), 2) full qualification of references (e.g. `List` has become `immutable#5861.this.List#8199` on line 45), 3) inference of unspecified types (e.g. the type of `xs` on line 44), 4) transformation of some applications to `apply` calls (lines 45 and 55).

Another significant change that happened to original code is attribution. All eligible trees, i.e. definitions and references, now carry symbols, which means that all bindings in the the code are now resolved and reified. For example, we see that all references

to `Int` say `Int#1850`, which means that all of them refer to the same definition (the printout does not say exactly which `Int` is that, but additional compiler flags can elicit that information).

Moreover, even though the printout does not show that, all trees also carry types computed for them by the typechecker. We could ask the compiler to print those as well via `-Xprint-types`, but the result would be unreadable, which is why we will show only a small fragment of the printout with `-Xprint-types` enabled. Below we can find a manually formatted fragment of the detailed printout for `List(42)`, in which the `{ ... }` parts indicate types of the corresponding AST nodes, with the type of `List` abbreviated for clarity.

```
immutable#5861.this {scala#25.collection#2744.immutable#5861.type}
.List#8199          {...immutable#5861.List#8197.type}
.apply#9078         {[A#9079](xs#9080: A#9079*)List#8198[A#9079]}
[Int#1850]          {(xs#12731: Int#1850*)List#8198[Int#1850]}
(42)                {Int#1850(42)}
                    {List#8198[Int#1850]}
```

Here we can see that every single tree has got an associated type. Even method references, such as `List.apply`, have types, even though they are not even values according to the Scala language specification. Another important observation that we can make is that types, much like trees, carry symbols to indicate resolved bindings.

To sum it up, there are two layers of information available about code in `scalac`. The first layer is syntactic, where we can only see shapes of abstract syntax trees, without knowing what they mean. The second layer is semantic, where we can additionally ask trees for underlying symbols and types to understand their meaning. As we have seen earlier, scala.reflect exposes both, providing a rich model of Scala programs.

The most prominent peculiarity of compiler internals and, by extension, of scala.reflect is pervasive desugaring. In order to be able to effectively use the language model, the metaprogrammer has to be aware of desugarings performed both during parsing and during typechecking. Unfortunately, some of these desugarings are irreversible, so oftentimes there is a tradeoff between using a detailed desugared representation and a lacking faithful representation.

3.2.2 Trees

In scala.reflect, abstract syntax trees are implemented as case classes that inherit from the abstract class `Tree` that provides common functionality. In the public API, these implementations are exposed via a laborious encoding that involves a combination of abstract type members and abstract vals (subsection 3.1.1), so for the sake of simplicity here we will discuss their internal representation.

In the listing below, we can see a simplified excerpt of `Trees.scala`, the source file that defines the majority of abstract syntax trees of the Scala compiler. All APIs shown in the listing are exposed in scala.reflect.

```
abstract class Tree extends Product {
  def pos: Position = { ... }
  def setPos(pos: Position): this.type = { ... }

  def tpe: Type = { ... }
  def setType(tp: Type): this.type = { ... }

  def symbol: Symbol = { ... }
  def setSymbol(sym: Symbol): this.type = { ... }

  ...
}

case class PackageDef(
  pid: RefTree, stats: List[Tree]) extends ...

case class ModuleDef(
  mods: Modifiers, name: TermName, impl: Template) extends ...

case class Template(
  parents: List[Tree], self: ValDef, body: List[Tree]) extends ...
```

Syntactic information. There are several dozen AST nodes in the Scala compiler, but they all are defined along the same lines. An AST node is a case class, whose children are encoded in immutable fields. AST nodes do not have links to parents, which requires metaprogrammer to maintain the parent structure manually.

Not all language features have dedicated AST nodes. Some features are encoded with a combination of simpler AST nodes (in our example, we saw that happen to tuple literals). Some features are represented via data structures like `Modifiers` that are separate from the AST hierarchy. Finally, some features are dropped altogether (in our example, that would be comments and formatting). This simplification and homogenization of language features is done for coding convenience because the same AST model is shared across almost the entire compilation pipeline. As a result, most scala.reflect trees are not roundtrippable, i.e. prettyprinting and parsing are not inverse of each other.

Using scala.reflect, it is possible to create ASTs that violate language syntax and/or typing rules. Data constructors are rarely strongly typed, inheritance hierarchy is open, there is no MetaML-like typing discipline [118]. This design decision is motivated by the fact that the same AST has to encode both high-level and low-level idioms at different stages of compilation.

Positions. Trees can carry positions that link them to particular locations in the corresponding source files. By default, all trees have empty positions, but they can be set via `Tree.setPos` either in the standard parser or manually by metaprogrammers. Below we can see an simplified fragment of position API exposed in scala.reflect.

```
trait Position {
  def source: File

  def start: Int
  def point: Int
  def end: Int

  def line: Int
  def column: Int

  ...
}
```

Positions can be either offset positions (in which case the offset is encoded in `point`) or range positions (in which case the range is encoded in `start` and `end`, with `point` representing location of the caret for error messages). Unless the Scala compiler runs in a special IDE-compatible mode, it only provides offset positions in order to save on memory footprint.

Positions can be used as an approximate indication of the origin of a tree, but they are generally unreliable, because there is no one-to-one correspondence between positions and trees. Some language constructs are desugared and then multiple ASTs have the same position. Some language constructs are not represented by AST nodes and then no ASTs have certain positions. In these cases, manual effort is required to correlate positions and elements of Scala syntax, and sometimes this effort amounts to reimplementing significant parts of the parser.

Semantic information. As we have seen in section 3.2, scala.reflect trees can also carry symbols and types to represent semantic information. We will get to the details of these data structures later in this chapter. Here we note that both `Tree.symbol` and `Tree.tpe` are mutable for performance reasons.

3.2.3 Symbols

Every universe has a symbol table that represents the collection of definitions known to it. This collection can be populated in various different ways (hardcoded, loaded from a classpath, etc), it can be set in stone or it can evolve over time - but nonetheless it is always there. Typically, the task of populating a symbol table or a fragment thereof is dedicated to a mirror (section 3.4).

Symbol table are composed of symbols, i.e. unique identifiers that carry additional metadata. Identity of symbols is based on referential equality, and names take no part in it, which allows to model shadowing, overloading, etc. In order to find out whether a given reference points to a given definition or whether two references mean the same, one compares their associated symbols. This only works within the same universe - correlation of artifacts between different universes is undefined.

Much like trees, symbols are publicly exposed via a combination of advanced Scala features (subsection 3.1.1), so, in order to keep things simple, here we will discuss their internal implementations. The code below represents a simplified extract from `Symbols.scala`, the source file that defines symbols of the Scala compiler. Some private members have been renamed for simplicity.

```
abstract class Symbol(owner0: Symbol, pos0: Position, name0: Name) {
  def owner: Symbol = { ... }
  def owner_=(owner: Symbol): Unit = { ... }

  def pos: Position = { ... }
  def setPos(pos: Position): this.type = { ... }

  def name: Name = { ... }
  def name_=(n: Name): Unit = { ... }

  def info: Type = { ... }
  def setInfo(info: Type): this.type = { ... }

  ...
}

abstract class TypeSymbol(owner0: Symbol, pos0: Position, ...)
extends Symbol(owner0, pos0, name0) with TypeSymbolApi {
  ...
}

class ClassSymbol(owner0: Symbol, pos0: Position, name0: TypeName)
extends TypeSymbol(owner0, pos0, name0) with ClassSymbolApi {
  ...
}
```

The symbol table. Symbols are organized in a tree based on the notion of ownership (represented `Tree.owner`). For example, a method parameter is owned by the containing method, which itself is owned by a containing class and so on. This tree grows from `_root_`, the root package. This means that in fact a symbol table is tree, but the name "symbol table" has become pervasive among the Scala compiler developers and the metaprogramming community, so we are going to use it anyway.

Every symbol that potentially owns other symbols, e.g. a package that can contain classes or a class that can contain methods, has an associated signature (represented by `Symbol.info`) that, among other things, contains a mutable collection of its children. The children collection is mutable, because implementing an immutable symbol graph with owner/child loops would be impractical. Such collections are also usually lazy so that the compiler does not have to load all its dependencies into its symbol table at once.

```
1  scala> :power
2  Power mode enabled. :phase is at typer.
3  import scala.tools.nsc._, intp.global._, definitions._
4  Try :help or completions for vals._ and power._
5
6  scala> class Test { def plus(x: Int, y: Int) = x + y }
7  defined class Test
8
9  scala> val test = symbolOf[Test]
10 test: TypeSymbol = class Test
11
12 scala> test.info
13 res0: Type =
14 AnyRef {
15   def <init>(): Test
16   def plus(x: Int,y: Int): Int
17 }
18
19 scala> test.info.decls.toList
20 res1: List[Symbol] = List(constructor Test, method plus)
21
22 scala> val plus = res0(1)
23 plus: Symbol = method plus
24
25 scala> plus.info
26 res2: Type = (x: Int, y: Int)Int
27
28 scala> plus.owner
29 res3: Symbol = class Test
30
31 scala> res1 == test
32 res4: Boolean = true
33
34 scala> plus.owner.owner
35 res5: Symbol = object $iw
```

The REPL printout above illustrates the structure of the symbol table. First, we obtain a symbol that corresponds to the freshly defined class `Test` using the `symbolOf` API (lines 9-10). This is the starting point of the exploration.

Then, we load the signature of `test` and observe that it contains a list of symbols that represents the definitions in class `Test` (lines 12-20). After getting to one of those symbols (lines 22-23), we observe that its signature also contains a list of children symbols (lines 25-26).

We conclude the exploration by getting back to `test` via `Symbol.owner` (lines 28-29), making sure that we end up where we started (lines 31-32). Naturally, `test` also has an owner (lines 34-35), which in this case is a synthetic object that REPL creates to wrap the code being executed.

Conceptually, the outlined notion of a symbol table is quite simple, but compiler internals complicate the situation. First, the symbol table is not just a tree (with respect to `Symbol.owner`), but is actually a forest. This happens because it is possible to use the scala.reflect API to create symbols that do not have an owner.

Moreover, there exist symbols that are not included in the children collection of their owners. For example, symbols of local variables are owned by their enclosing methods, but signatures of those methods only include type and term parameters, not local variables.

Despite these limitations, symbols are the only official way to traverse the definition structure of the program in scala.reflect. Most of the time, it works well, but some scenarios encounter significant problems. For example, because of the fact that signatures do not contain local variables, it is impossible to go through all definitions in the program using the symbol table alone.

Correlation with abstract syntax trees. As we can see, scala.reflect offers two different views of Scala programs. One view consists of a collection of abstract syntax trees that represent the program and all its dependencies (we will call it the tree view). Another view represents a subset of the first view where symbols and their signatures capture a subset of the structure of definitions (we will call it the symbol view).

If we run `scalac` with another private flag, `-Yshow-syms`, we will see how both views of the program evolve phase after phase. The printout below has been manually edited for readability reasons.

```
$ scalac -Xprint:parser -Xprint:typer -uniqid -Yshow-syms Test.scala
[[syntax trees at end of                     parser]] // Test.scala
// see earlier printouts in section 3.2

[[symbol layout at end of parser]]
package scala#24 (final)

[[syntax trees at end of                       typer]] // Test.scala
// see earlier printouts in section 3.2
```

```
[[symbol layout at end of typer]]
constructor Object#3112
object Test#7854
object Test#7855
    constructor Test#7942
    value <local Test>#7947
    value head#7945 (<stable> <accessor>)
    value head#7946 (private <local>)
    value xs#7943 (<stable> <accessor>)
    value xs#7944 (private <local>)
package <empty>#4 (final <static>)
package scala#25 (final)
    class Int#1850 (final abstract)
    method apply#20019 (case <synthetic>)
    method head#19914
    method println#7621
    object Predef#2088
    object Tuple2#2649 (<synthetic>)
    package immutable#5861 (final)
        class List#8198 (sealed abstract)
        method apply#9078 (override)
        object List#8199
    type AnyRef#2765
package scala#24 (final)
```

We can see that immediately after `parser`, the tree view includes an unattributed AST of the only source file of the program. The symbol view only contains a single symbol for the `scala` package, because that symbol was used by the parser to perform the tuple desugaring.

As the compilation progresses, the symbol view gets populated by the symbols that were discovered while processing the ASTs of the program. For example, after `typer`, the symbol view is much richer, now including the definitions introduced in the program and the definitions referenced from the program.

From these two observations, we can conclude that the tree and symbol views can be inconsistent with each other. After `parser`, the tree view is significantly richer than the symbol view. After `typer`, however, the symbol view becomes more populated, because it loads definitions of dependencies from their class files.

This inconsistency is intentional. In the JVM world, ASTs are not persisted in class files. Therefore, without using additional mechanisms to bundle sources together with binaries, getting access to ASTs of the classpath is impossible. As a result, the developers of the Scala compiler decided to introduce the symbol table as a complementary view of the program structure.

For metaprogrammers who use scala.reflect to generate new code, this presents an significant complication, because they oftentimes need to make sure that the synthetic trees that they produce are consistent with the symbol table of the compiler. Unfortunately, the compiler does not have any mechanisms built in to help out with that, so detecting and fixing inconsistencies requires deep knowledge of compiler internals. Documenting this knowledge is outside the scope of this dissertation, but we refer curious readers to [14] for more information.

3.2.4 Types

Types are probably the simplest among the foundational data structures of scala.reflect. Unlike trees, they are much more modest in number (there is barely two dozen different types in the Scala compiler). Unlike symbols, they are not arranged in any complicated graph structure.

Below we can see a simplified excerpt of `Types.scala`, the source file that defines types of the Scala compiler. All APIs shown in the listing are exposed in scala.reflect.

```
abstract class Type {
  def termSymbol: Symbol = { ... }
  def typeSymbol: Symbol = { ... }
  def <:<(that: Type): Boolean = { ... }
  def =:=(that: Type): Boolean = { ... }
  ...
}

case class SingleType(pre: Type, sym: Symbol) extends ...
case class TypeRef(pre: Type, sym: Symbol, args: List[Type]) extends ...
```

We observe that, unlike trees and symbols, types are immutable. Trees have mutable attributes, symbols have mutable signatures, but the only mutable fields that types have are private caches. Types are created much more commonly than trees and symbols, so they are aggressively cached and, consequently, can not be mutable.

There are uncanny parallels than can be drawn between how trees and types are modelled in scala.reflect. Both of them implement elements of the language model, both of them represent bindings with symbols, both of them sometimes need to be traversed recursively.

Nonetheless, trees and types are implemented via different case class hierarchies. The rationale behind this decision is the fact that not all types can be represented with syntax (e.g. symbol signatures or intermediate type variables created by type inference). As a result, this leads to code duplication in various pieces of the metaprogramming infrastructure that deal with trees and types.

In the language model, types serve three major roles: 1) specifying types of ASTs, 2) specifying signatures of symbols, 3) modelling type system concepts. Let us see how each of these roles is carried out.

Specifying types of ASTs. As illustrated by our running example, all attributed trees have types associated with them. According to subsection 3.2.2, one can use `Tree.tpe` to access the type of a given tree.

In the REPL printout below, we explore this aspect of the scala.reflect API. In order to conveniently work with trees, we use the `q"..."` notation that creates ASTs from Scala-like code snippets (subsection 3.3.2). In order to attribute trees, we use the `typed` method available in the power mode of the REPL.

```
1  scala> :power
2  Power mode enabled. :phase is at typer.
3  import scala.tools.nsc._, intp.global._, definitions._
4  Try :help or completions for vals._ and power._
5
6  scala> val list = q"List(42)"
7  list: Tree = List(42)
8
9  scala> list.tpe
10 res0: Type = null
11
12 scala> val tlist = typed(list)
13 tlist: Tree = immutable.this.List.apply[Int](42)
14
15 scala> tlist.tpe
16 res1: Type = List[Int]
17
18 scala> showRaw(tlist.tpe, printIds = true)
19 res2: String = TypeRef(
20   ThisType(scala.collection.immutable#5861),
21   scala.collection.immutable.List#15177,
22   List(TypeRef(ThisType(scala#25), scala.Int#1850, List())))
```

On lines 6-7, we create a tree that corresponds to `List(42)`. By default, trees created by quasiquotes are unattributed, so `list.tpe` returns null (lines 9-10). Typechecking produces an attributed tree (lines 12-13) which has a correct type associated with it (line 14-15).

Using the `showRaw` API, we can inspect the structure of the resulting `TypeRef`, observing that it has a reference to the class symbol `List` and a type argument which is itself a `TypeRef`. The first part of a `TypeRef` is important for modelling references to nested types, but in our case, where both `List` and `Int` are top-level types, it trivially points to the owner of the underlying symbol.

Specifying signatures of symbols. Most symbols in scala.reflect have signatures that are stored in `Symbol.info`. These signatures are also modelled with types. In the REPL printout below, we can see a few examples, where class signatures are represented with `ClassInfoType` and method signatures are represented with `MethodType`.

```
scala> class Test { def plus(x: Int, y: Int) = x + y }
defined class Test

scala> val test = symbolOf[Test]
test: TypeSymbol = class Test

scala> showRaw(test.info)
res0: String = ClassInfoType(
  List(TypeRef(ThisType(scala#25), TypeName("AnyRef")#2765, List())),
  Scope(termNames.CONSTRUCTOR#218701, TermName("plus")#218702),
  Test#218699)

scala> val plus = test.info.member(TermName("plus"))
plus: Symbol = method plus

scala> showRaw(plus.info)
res1: String = MethodType(
  List(TermName("x")#218704, TermName("y")#218705),
  TypeRef(ThisType(scala#25), scala.Int#1850, List()))
```

Modelling type system concepts. Types are also used by the compiler internally to perform typechecking, implicit search, type inference, etc. Therefore, type encapsulate common notions of the Scala type system. For example, `Type.<:<` represents a subtype check, `Type.members` returns a list of members with respect to linearization, `Type.widen` converts from singleton types to their underlying types, and so on.

3.3 Notations

The language model described in section 3.2 forms the backbone of scala.reflect, but working directly with its data structures is not always very convenient. For instance, as we have seen above, it is not uncommon for relatively simple code snippets to explode into deeply nested abstract syntax trees. That is why scala.reflect includes several DSL-like facilities to simplify manipulation of the underlying data structures.

A very popular approach to notations is quoting, in which language syntax put in a special context is used as a DSL that produces a corresponding element of the language model. Quasiquoting provides an additional capability to enrich quotes with holes, i.e. locations where other artifacts can be inserted. All notations described in this section are based on the notion of quasiquoting.

3.3.1 Reify

In addition to `Tree`, which does not have a static type, scala.reflect supports `Expr[T]`, which represents a statically-typed wrapper over `Tree`.

Below we can see a simplified definition of `Expr`. There we can see `Expr.tree` that exposes the wrapped tree, as well as two magic methods `Expr.splice` and `Expr.value`. The former will be used later in this chapter, and the latter is important for advanced aspects of our macro system (subsection 4.3.1).

```
trait Expr[+T] {
  def tree: Tree
  def splice: T
  val value: T
  ...
}
```

Exprs can be created either manually, by wrapping abstract syntax trees and accompanying type tags (subsection 3.3.3) in a statically-typed shell, or automatically, by calling the `reify` method provided by scala.reflect. The manual approach is tedious and can subvert static guarantees, so it is discouraged.

`reify` is a magic method that takes a well-typed expression of type `T` and produces an equivalent `Expr[T]`. Reify was inspired by on-demand tree reification via `Code[T]` from section 2.3 and was initially using the same infrastructure that used to implement `Code` in the previous versions of Scala.

Syntactic aspect. The main purpose of reify is reproducing the syntactic structure of a given expression for runtime introspection. Of course, this works within the boundaries supported by the language model. For example, reify performs desugarings and does not save comments and formatting.

```
1  scala> import scala.reflect.runtime.universe._
2  import scala.reflect.runtime.universe._
3
4  scala> val list42 = reify(List(42) /* list of forty-two */)
5  list42: Expr[List[Int]] = Expr[List[Int]](List.apply(42))
6
7  scala> list42.tree
8  res0: Tree = List.apply(42)
9
10 scala> showRaw(list42.tree)
11 res2: String = Apply(
12   Select(Ident(scala.collection.immutable.List), TermName("apply")),
13   List(Literal(Constant(42))))
```

In order to experiment with the functionality provided by reify and exprs, we import a runtime universe on lines 1-2, which makes it possible to make use of the entire spectrum of scala.reflect APIs.

Afterwards, we create an expr that corresponds to `List(42)`, a part of our running example from section 3.2. We can see that much like the internal representation used by the Scala compiler after `typer`, the expr looks desugared (line 8), which is unsurprising given that the underlying technology the same.

Composition of exprs, i.e. composition of underlying trees, is achieved via `Expr.splice`. Whenever, an expr is spliced into an argument of a reify call, the call to `splice` is dropped and its expr is inserted into the result. Since `Expr.splice` is statically typed, it is impossible to create syntactically or semantically invalid trees with reify unless the programmer intentionally subverts static types of exprs using `asInstanceOf` or passes a nonsensical tree in a manually created expr.

```
scala> reify {
     |   val xs = list42.splice
     |   val head = xs.head
     |   println((xs, head))
     | }
res1: Expr[Unit] = Expr[Unit]({
  val xs = List.apply(42);
  val head = xs.head;
  Predef.println(Tuple2.apply(xs, head))
})
```

Semantic aspect. Reify not only saves syntactic information about the tree representing its argument, but also remembers some semantic information. Apart from the type of the entire expression, reify does not retain types, but it does retain symbols of free variables. Note how in the listing below reify has the symbols for `Int`, `x` and `List`, but not for `y`, which is defined locally.

```
scala> {
     |   val x = 42
     |   reify {
     |     val y: Int = x
     |     List(y)
     |   }
     | }
res2: Expr[List[Int]] =
Expr[List[Int]]({
  val y: Int = x;
  List.apply(y)
})
```

```
scala> showRaw(res3.tree, printIds = true, printTypes = true)
res3: String = Block(List(
  ValDef(
    Modifiers(), TermName("y"), Ident(scala.Int#391),
    Ident(TermName("x")#10175))),
  Apply(
    Select(
      Ident(scala.collection.immutable.List#2340),
      TermName("apply")),
  List(Ident(TermName("y")))))
```

In order to understand how this works, let us consider reify as a serializer for attributed syntax trees of scala.reflect. This intuition explains how reify operates, and why it is so selective about the attributes that it stores.

As explained in section 2.2, `scalac` saves the top-level part of its symbol table in JVM class files produced during compilation of Scala programs. Afterwards, the runtime universe loads the saved part of the symbol table at runtime (subsection 3.4.4), and reified trees can look up the correct symbols.

Below we can see a simplified fragment of code generated for `reify(List(42))`. In that snippet, `u` and `m` stand for a universe and a mirror used to instantiate the corresponding expr. Note how the compiler emits a call to `Mirror.staticModule` (section 3.5) using a fully-qualified name in order to look up a symbol for `List` in the symbol table of the runtime universe.

```
u.Apply(
  u.Select(
    u.Ident(m.staticModule("scala.collection.immutable.List")),
    u.TermName("apply")),
  List(u.Literal(u.Constant(42))))
```

Unfortunately, even though `scalac` saves the information about top-level definitions and their members, it does not save local definitions in order to reduce class file size. This makes it impossible to reliably refer to local symbols in a runtime universe, which explains why reify does not persist references to such symbols.

However, the compiler does a special trick for local symbols that are free in the reified expression. As a best effort solution, reify creates a special kind of symbol that only remembers a few simple facts about the free variable.

Here is a simplified fragment of code generated for `reify(List(x))`, where `x` is an integer free variable defined locally. Note how `newFreeTerm` takes both "x" and x, capturing both the compile-time and runtime aspects of the free variable. As a result, users of the expr can obtain both the type and the value of `x`.

```
val free$x1 = u.internal.reificationSupport.newFreeTerm("x", x, ...)
free$x1.setInfo(m.staticClass("scala.Int").asType.toType)
u.Apply(
  u.Select(
    u.Ident(m.staticModule("scala.collection.immutable.List")),
    u.TermName("apply")),
  List(u.Ident(free$x1)))
```

This reasoning also explains why reify does not store types of reified subtrees. These types may reference local symbols defined inside the argument of reify, and those symbols are out of the question.

Notation. As a notation for syntax trees, reify works pretty well for simple snippets of code, but unfortunately it does not scale well to the metaprogramming style prevalent in the Scala community.

Practical evaluation of reify as the notation powering Scala macros (chapter 4) has found several significant limitations. Concretely, reify:

- Imposes additional typing discipline. Exprs are more heavyweight to pass around than plain syntax trees, because they are statically typed. As a result, metaprograms that use reify are more verbose than their counterparts that work with trees.
- Can only express well-typed snippets of code. As the size of a metaprogram grows, it becomes increasingly important to modularize - both the metaprogram and the trees that it manipulates. Reify prevents modularization, because it makes it impossible to pull apart definitions and references to these definitions into separate quotes.
- Provides limited composability. While string composition is too permissive, allowing too many locations to insert code snippets into, reify allows too few. Among others, it is impossible to splice definitions if they are supposed to be referenced, it is impossible to splice dynamically computed references, it is impossible to splice lists of trees, etc.
- Can only support construction. Method calls can not be used in pattern position in Scala, which means that reify can only be used to construct, but not to deconstruct syntax trees.

To put it in a nutshell, we initially planned that reify will be scala.reflect's notation of choice for abstract syntax trees. However, based on user feedback, we realized that reify is insufficient for practical applications of scala.reflect. As a result, shortly after the initial release of scala.reflect, we started working on a new notation to replace reify. Results of our work can be found in subsection 3.3.2.

3.3.2 Quasiquotes

Quasiquotes are custom string interpolators [114] that construct and deconstruct abstract syntax trees according to string templates. In comparison with reify (subsection 3.3.1), quasiquotes are much more flexible and can also be used in pattern matches. As a result, quasiquotes are now the go-to notation for tree manipulations in scala.reflect.

As part of his master studies, Denys Shabalin implemented a prototype of quasiquotes and then shipped this implementation in Scala 2.11. Below we provide a brief overview of Denys's contributions, leaving the full story to [103] and [104].

Syntactic aspect. Quasiquotes feature a family of interpolators that cover five syntactic categories: statements, types, cases, patterns and enumerators. Some syntactic elements (e.g. import clauses, import selectors and modifiers) do not have dedicated trees, so they can not be expressed with quasiquotes.

```
1  scala> import scala.reflect.runtime.universe._
2  import scala.reflect.runtime.universe._
3
4  scala> q"List(42)"
5  res0: Tree = List(42)
6
7  scala> tq"List[Int]"
8  res1: Tree = List[Int]
9
10 scala> cq"List(x) => x"
11 res2: CaseDef = case List((x @ _)) => x
12
13 scala> pq"List(x)"
14 res3: Tree = List((x @ _))
15
16 scala> fq"x <- List(42)"
17 res4: Tree = `<-`((x @ _), List(42))
```

Composition of trees is expressed via built-in functionality of string interpolation. It is possible to insert both single trees (unquoting, line 7 below) and multiple trees (unquote splicing, line 4 below) into bigger trees.

```
1  scala> val args = List(q"42")
2  args: List[Literal] = List(42)
3
4  scala> val list42 = q"List(..$args)"
5  list42: Tree = List(42)
6
7  scala> q"println($list42)"
8  res5: Tree = println(List(42))
```

```
9
10 scala> q"class $list42"
11 <console>:38: error: Name expected but Tree found
12        q"class $list42"
13                  ^
14
15 scala> q"val weird: $list42 = ???"
16 res7: ValDef = val weird: List(42) = $qmark$qmark$qmark
```

Since quasiquotes are custom string interpolators, they can restrict the locations that allow unquoting and splicing according to the Scala grammar. This validation happens at compile time. For example, an attempt to insert a tree into a location that expects a name on line 10 results in a compilation error on lines 11-13.

Since quasiquotes are just a notation for syntax trees, unlike reify which comes with a dedicated datatype, safety guarantees are limited to what is provided by the underlying AST. For instance, in scala.reflect it is possible for a result of composition of two trees to be nonsensical according to Scala grammar (lines 15-16), so scala.reflect quasiquotes have the same property.

String interpolation also supports pattern matching, so quasiquotes can be used to deconstruct syntax trees. This is one the main improvements over reify (subsection 3.3.1), which provided no way to improve pattern matching experience. This was one of the main reasons why the introduction of quasiquotes was enthusiastically welcomed by the Scala community.

```
1 scala> val list42 = q"List(42)"
2 list42: Tree = List(42)
3
4 scala> val q"$fn(..$args)" = list42
5 fn: Tree = List
6 args: List[Tree] = List(42)
```

Unquoting and unquote splicing are type-directed. By providing instances of `Liftable` and `Unliftable` typeclasses for given types, users can insert and extract values of those types, even if they are not syntax trees. Lifting and unlifting enable a convenient notation to convert between simple data structures (lists, maps, etc) and trees. Scala.reflect provides instances of `Liftable` and `Unliftable` for primitives and popular standard library types such as tuples, collections and others.

```
1 scala> implicit def liftInt[T <: Int]: Liftable[T] =
2      | Liftable {
3      |   v => Literal(Constant(v))
4      | }
5 liftInt: [T <: Int]=> Liftable[T]
6
```

```
scala> q"${42}"
res8: Tree = 42

scala> implicit def unliftInt: Unliftable[Int] =
     | Unliftable {
     |   case Literal(Constant(v: Int)) => v
     | }
unliftInt: Unliftable[Int]

scala> val q"${fortyTwo: Int}" = res1
fortyTwo: Int = 42
```

Semantic aspect. Quasiquotes deliberately allow ill-typed code snippets. As a result, it is unclear how to attach semantic information to trees created with quasiquotes, because quoted code does not have semantic information in the first place. We have made several attempts at this conundrum.

We tried to typecheck speculatively and then, in case of success, attach semantic information to the result. This ran into two difficulties. First, because of desugarings, scala.reflect makes it really hard to correlate untyped and typed trees, which meant that reliably attaching semantic information after a successful typecheck was very difficult. Secondly, this design creates ample opportunities for confusion. Minor changes or typos in quoted snippets may lead to major impact on the result.

We tried to perform name resolution without doing full-blown typechecking and then save semantic information about established bindings. Unfortunately, it turned out that in Scala name resolution can not be performed without typechecking. Scala allows wildcard imports from arbitrary terms (e.g. as explained in subsection 3.1.1, the omnipresent `import scala.reflect.runtime.universe._` is an import from a regular value), so name resolution needs the result of typechecking such terms in order to proceed.

As a result, at the time of writing, trees created with quasiquotes do not contain any semantic information. Recent research into semantic quasiquotes by Lionel Parreaux [92] may provide ways to remedy the situation, but at the moment such functionality is not available in the standard distribution of the language. Nonetheless, based on our experience with reify (subsection 3.3.1), we consider the resulting gain in flexibility to be worth giving up static guarantees.

From the discussion above, we can see that quasiquotes have been a major improvement over reify in the areas of modularity, flexibility and pattern matching. The only downside of quasiquotes is the lack of semantic information, but that has not prevented quasiquotes from becoming the AST notation of choice in scala.reflect.

3.3.3 TypeTags

`TypeTag` is a magic typeclass whose instances wrap a scala.reflect `Type` representing their type parameter. The main use case for type tags is carrying representations of type parameters to metaprograms executed at compile time (subsection 4.3.2).

Much like exprs (subsection 3.3.1), type tags can be created either manually, by wrapping an underlying type in a statically-typed shell, or automatically, by relying on automatic derivation of `TypeTag` instances provided by scala.reflect. The manual approach is tedious and can subvert static guarantees, so it is discouraged.

When an instance of `TypeTag` is required, and there is none in scope, the Scala compiler generates one on the fly. In this section, we will call this process materialization, and later in the dissertation, we will see how it generalizes to arbitrary type classes (section 6.1).

```
scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> def tag[T](implicit t: TypeTag[T]) = t
tag: [T](implicit t: TypeTag[T])TypeTag[T]

scala> tag[List[Int]]
res0: TypeTag[List[Int]] = TypeTag[scala.List[Int]]

scala> res0.tpe
res1: Type = scala.List[Int]
```

Type tags give rise to a convenient notation to express scala.reflect types. The `typeOf` oneliner function requests a type tag and them immediately unwraps it, returning an underlying type.

```
scala> def typeOf[T](implicit tag: TypeTag[T]) = tag.tpe
typeOf: [T](implicit tag: TypeTag[T])Type

scala> typeOf[List[Int]]
res2: Type = scala.List[Int]
```

Scala.reflect provides two kinds of type tags: `WeakTypeTag` and `TypeTag`. The former can reify all types, whereas the latter only reifies types that do not contain references to untagged type parameters and abstract type members and produces a static error otherwise. This distinction is helpful to make sure that chains of generic methods correctly propagate tags across the call graph.

```
scala> def generic[T: TypeTag](x: T) = implicitly[TypeTag[T]].tpe
generic: [T](x: T)(implicit evidence$1: TypeTag[T])Type
```

```
scala> def goodForwarder[T: TypeTag](x: T) = generic(x)
goodForwarder: [T](x: T)(implicit evidence$1: TypeTag[T])Type

scala> goodForwarder(List(42))
res7: Type = List[Int]

scala> def badForwarder[T](x: T) = generic(x)
<console>:39: error: No TypeTag available for T
       def badForwarder[T](x: T) = generic(x)
                                           ^
```

Unlike with exprs (subsection 3.3.1) which require a magic function to express composition, composition of type tags happens automatically inside the implementation of type tag materialization.

When materializing a type tag, the compiler traverses its structural parts and performs recursive lookup of `TypeTag` instances for them. This means that if there is already a type tag in a scope, it will be automatically picked up by any materializations that involve the associated type. Such composition is guaranteed to be well-typed if the metaprogrammer does not lie about static types of type tags using `asInstanceOf` or manual type tag creation.

```
scala> def listTag[T](implicit t: TypeTag[T]) = tag[List[T]]
listTag: [T](implicit t: TypeTag[T])TypeTag[List[T]]

scala> listTag[Int]
res3: TypeTag[List[Int]] = TypeTag[scala.List[Int]]

scala> res2.tpe
res4: Type = scala.List[Int]
```

Type tags preserve semantic information using a similar strategy to reify, with free variables persisted via symbols. Unfortunately, unlike reify that can avoid persisting local symbols, type tags do not have this luxury, because types consist of symbols (subsection 3.2.4). While simple types that refer to global definitions work like reify, more complicated types that involve refinements and existentials must preserve local symbols because of the way how they are encoded in compiler internals.

As a result, the implementation of type tags has to serialize an approximation of the relevant section of the symbol table, leading to partially correct results or even in spurious compilation failures. This is a big conceptual flaw of the current implementation that can not be fixed without a significant upgrade to how the Scala compiler serializes its symbol table, and that is highly unlikely given how risky this change will be in a production compiler.

As a notation for types, type tags are similar to reify. Since they work only for statically-typed code, their composability is limited and so is their ability to construct types whose shape is not known in advance.

Moreover, analogously to reify, type tags can not be used to take types apart. While the former shortcoming manifests itself much more rarely than the problem with reify (the structure of types is typically much simpler than the structure of trees, so composition is needed more rarely), the latter is a serious problem.

To put it in a nutshell, type tags are a useful tool, since they can often simplify code that manipulates types. Unfortunately, their area of applicability is limited, because they do not work well with compound and existential types, and they do not support pattern matching. Since scala.reflect types must contain semantic information, improving on type tags is very challenging, and our improvement entailed designing a new metaprogramming framework (chapter 8) that represents types with their syntax (section 8.2), and therefore can use quasiquotes to manipulate types (section 8.3).

3.3.4 Hygiene

The notion of hygiene has been widely popularized by the Lisp community. A code generator is called hygienic if it avoids name clashes between regular and generated code.

As numerous experience reports show, hygiene is of great importance to code generation, because name clashes are often non-obvious and lack of hygiene might manifest itself in subtle ways. However, sometimes it may be practically useful to violate hygiene and establish bindings across reflection artifacts, so facilities to work around hygiene may be beneficial.

Hygiene is especially useful for quoted notations, because code in quotes looks similar to regular code. This similarity suggests an intuition about lexical scoping that may not hold, because quotes may be deconstructed, may be inserted into other quotes, etc.

In this chapter, we adapt the notion of hygiene to fit the context of a metaprogramming framework whose functionality goes beyond code generation.

We say that a particular facility for composition of code snippets of a particular kind is hygienic if it prevents the following two kinds of name capture without explicit effort from a metaprogrammer: (A) when a definition in an enclosing snippet binds a reference in an inserted snippet, (B) when a definition in an inserted snippet binds a reference in an enclosing snippet.

We say that a particular kind of code snippets is hygienic if all possible facilities for their composition are hygienic.

Statically-typed notations. The simplest way to avoid bindings between different code snippets is to establish bindings in advance, upon snippet creation. This can be achieved when code snippets are well-typed, which is guaranteed for reify (subsection 3.3.1) and type tags (subsection 3.3.3).

We observe that exprs created by reify are hygienic. Exprs can only be created and composed via reify, and during that composition neither (A) nor (B) are possible. Bindings to free variables can not get corrupted, because they are saved in symbols during reification. Bindings to local variables can not get corrupted, because: 1) reify can not take exprs apart, 2) splices are expressions not statements, so they can not introduce new variables into the scope where they are inserted.

We observe that type tags created by materialization are hygienic. Type tags can only be created and composed via materialization, and that makes both (A) and (B) impossible. Since materialization saves symbols for all bindings, bindings can not get corrupted.

Hygiene in exprs and type tags can be subverted, because they can manually created from naked trees and types. This style of metaprogramming lacks static guarantees provided by reify and materialization, so it is typically discouraged.

Untyped notations. Quasiquotes (subsection 3.3.2) allow ill-typed code snippets by design, so for them establishing bindings in advance is impossible. As a result, at the moment quasiquotes are unhygienic.

Bringing hygiene to quasiquotes is an important element of future work. In this area, it is conventional to look for inspiration in hygienic macro systems of Lisp-family programming languages [69, 70, 40, 24, 51, 50].

In his dissertation [101], Denys Shabalin applied the hygiene algorithm based on renamings [51] to a simple Scala-like language. Scaling these results to full Scala seems to be a promising direction for future research.

Another interesting direction is to consider the hygiene algorithm based on sets of scopes [50] that has been recently developed as a successor for [51]. This new algorithm has already been successfully used [31] in Sweet.js [32]. Tim Disney, the author of Sweet.js, reports that the implementation of hygiene based on sets of scopes is "mostly understandable" and faster.

3.4 Environments

In scala.reflect, environments are represented with universes (subsection 3.1.1). For every supported environment, there is a special subclass of `scala.reflect.api.Universe` that provides an implementation of the abstract language model for this particular environment.

`Universe` is a massive class which at the moment of writing inherits from twenty traits. Split over these traits are several hundred public APIs, so implementing a universe is a very challenging task.

```
abstract class Universe extends Symbols
                           with Types
                           with FlagSets
                           with Scopes
                           with Names
                           with Trees
                           with Constants
                           with Annotations
                           with Positions
                           with Exprs
                           with TypeTags
                           with ImplicitTags
                           with StandardDefinitions
                           with StandardNames
                           with StandardLiftables
                           with Mirrors
                           with Printers
                           with Liftables
                           with Quasiquotes
                           with Internals
```

While a universe provides an infrastructure for the language model and implements operations that work with elements of the model, it does not know how to populate a symbol table (subsection 3.2.3) that corresponds to a particular environment. This is the job for a dedicated entity called `Mirror`.

Below we can find a simplified definition of `Mirror` from scala.reflect that captures the essence of mirrors in the scala.reflect framework. There we can see that a mirror links to an enclosing universe and provides a symbol that corresponds to the root package of Scala programs (`_root_`). From there, using symbol signatures, we can reach all top-level definitions and all their members. Unfortunately, because of the way how compiler internals implement symbol tables, mirrors cannot help with looking up local symbols (subsection 3.2.3).

```
abstract class Mirror {
  val universe: Universe
  val RootPackage: universe.ModuleSymbol
  ...
}
```

Every universe is accompanied with one or more mirrors. There is always a `rootMirror` that reflects the foundational definitions from the standard library, as well as zero or more additional mirrors. Additional mirrors are necessary if an environment can define subenvironments that correspond to different classpaths. In that case, such mirrors are obtained in an implementation-specific way.

```
trait Mirrors {
  self: Universe =>

  type Mirror >: Null <: scala.reflect.api.Mirror
  val rootMirror: Mirror

  ...
}
```

In this chapter, we will discuss how the metaprogramming API provided by `Universe` and `Mirror` is implemented for different environments. We will find out the peculiarities of how particular environments implement the unified API and will outline platform-dependent extensions.

3.4.1 Scala compiler

Metaprograms that run against the Scala compiler are executed in the inversion-of-control style, i.e. instead of a programmer instantiating a universe and running their metaprograms against that universe, a programmer registers their metaprograms to executed inside the compiler.

The Scala compiler exposes three kinds of extension points: 1) compiler plugins that can register custom phases to run after normal phases (section 2.4), 2) typer plugins that can register callbacks invoked at specific points of the internal implementation of the typer phase (outside of the scope of the dissertation), 3) macros that run when the typechecker encounters usages of specially defined methods, types, annotations, etc. (chapter 4 and chapter 5). At the time of writing, only macros are written against scala.reflect, whereas other extensions - compiler and typer plugins - utilize the entirety of compiler internals.

The Scala compiler, i.e. `scala.tools.nsc.Global`, is a scala.reflect universe. This is unsurprising, since scala.reflect has been created by extracting and refining the functionality of compiler internals (subsection 3.1.1).

There is just one mirror, i.e. a root mirror, associated with the Scala compiler. It reflects the combination of the classpath of the compilation and the sourcepath constituting the underlying compilation run.

Since language models of scala.reflect and compiler internals are so similar, execution of scala.reflect metaprograms inside the Scala compiler is trivial. As long as metaprograms conform to the expectations of compiler internals, e.g. utilize correct desugarings (section 3.2) and keep the symbol table in sync with generated code (subsection 3.2.3), everything works well. Since the Scala compiler can oftentimes be very picky, familiarity with compiler internals is required for effective use of scala.reflect.

3.4.2 Dotty

Dotty [42] is a research platform for new language concepts and compiler technologies for Scala, designed to eventually succeed the Scala compiler. Dotty has not yet shipped, but it can already compile the standard library and is rapidly approaching an initial release.

The focus of Dotty is mainly on simplification. It removes extraneous syntax (e.g. no XML literals), and tries to boil down Scala's types into a smaller set of more fundamental constructors. The theory behind these constructors is researched in DOT [2], a calculus for dependent object types.

Dotty features a brand new implementation of Scala, including an independent frontend and a compilation pipeline that is significantly different from the Scala compiler.

Therefore, Dotty needs an integration layer to adapt its internal functionality to the API required by `Universe`. Unfortunately, developing such a layer seems infeasible. Huge API surface, overcomplicated language model, undocumented invariants - this all makes implementing `Universe` look like an insurmountable task.

As a result, Dotty does not support and does not plan to support scala.reflect. Instead, we created scala.meta, a new metaprogramming framework designed to enable third-party implementations (chapter 8) and are relying on it to make it possible for macros and other important metaprograms to run on Dotty.

3.4.3 IDEs

There are three major IDEs in the Scala community: IntelliJ [65], Eclipse [99] and ENSIME [44].

Eclipse and ENSIME internally use the presentation compiler, i.e. an instance of a slightly modified Scala compiler that runs in a daemon mode. Most of the information from subsection 3.4.1 applies here, except for the fact that the presentation compiler is

particularly capricious. Because of its continuous nature and the necessity to handle incomplete programs, the presentation compiler has additional undocumented expectations on how metaprograms should communicate with it. As a result, some macros that run correctly in the Scala compiler, can terminate with errors when executed in the presentation compiler.

IntelliJ features its own frontend for Scala, developed independently from the Scala compiler. Therefore, much like Dotty, IntelliJ struggles with running scala.reflect metaprograms. After an unsuccessful attempt to remedy the situation, we decided to forgo support for scala.reflect in IntelliJ and instead focus on support for scala.meta (section 8.4).

3.4.4 JVM

`scala.reflect.runtime.universe.JavaUniverse` is a scala.reflect universe that uses JVM reflection (section 2.1) to implement the scala.reflect API. Unlike the Scala compiler, which uses the compilation classpath to populate its symbol table, a runtime universe fills in its symbol table using JVM class loaders.

There is a singleton instance of `JavaUniverse` called `scala.reflect.runtime.universe`. The root mirror of this universe is automatically created from the class loader that contains the standard library, whereas other mirrors can be created on demand by passing a class loader to `JavaUniverse.runtimeMirror`.

This creates an interesting arrangement in which there exists a singleton universe with a singleton symbol table that is shared between multiple mirrors. Every mirror has its own copy of package symbols including root packages (signatures of these symbols contain information about definitions residing in packages, so we can not share these symbols), whereas other top-level symbols are shared if they correspond to the same class. This ties symbol identity to class identity established by class loaders.

The implementation the runtime universe shares a significant chunk of code with the Scala compiler. When we were extracting the scala.reflect API from compiler internals (subsection 3.1.1), we modularized a sizeable part of the Scala compiler that handles classpath management and other common functionality (loading of Scala signatures, type system abstractions, etc) into a separate module, and we used that module to build the runtime universe.

The differences between the Scala compiler and the runtime universe are caused by the decision to use class loaders instead of classpaths.

First, unlike classpaths, class loaders do not provide access to classfiles, so we had to replace the logic of class file parsing implemented in the compiler with the logic that does the same based on the `java.lang.Class` API provided by the JVM. This worked,

because in both cases the majority of metadata is encoded in binary Scala signatures (section 2.2), and decoding Scala signatures does not require any special APIs.

Secondly, unlike classpaths, class loaders do not allow eager enumeration of package contents. This makes it impossible to reliably list members of language model elements that stand for packages. In practice, this is less of a problem that it may seem (because it is still possible to resolve fully-qualified names), but nevertheless it remains a glaring hole in introspective capabilities of the runtime universe.

The main implementation challenge came from thread safety. Historically, the Scala compiler has never been thread-safe, because key elements of its language model, e.g. symbol signatures (subsection 3.2.3), were always mutable. However, this has not been a problem, because compile-time metaprograms are run in an inversion-of-control style, with the compiler launching them in singlethreaded mode.

Runtime metaprograms can be launched in multithreaded mode, so for runtime universes the lack of thread safety has become an issue. We made several attempts at fixing data races, introducing thread-local variables, using lock-free programming and fine-grained locks - all without much progress. The only approach that ended up being practical was a global lock that the runtime universe has to acquire whenever it performs a potentially side-effecting operation. This is clearly unsatisfying.

To put it in a nutshell, we have been able to reuse the codebase of the Scala compiler to deliver a universe that can execute scala.reflect metaprograms at runtime. Unfortunately, the limitations of the underlying technologies (inability to enumerate definitions in class loaders and thread-unsafety of the Scala compiler) have negatively affected usability of the runtime universe.

3.4.5 JavaScript

JVM is the main target platform for Scala, but it is not the only one. In the last several years, Scala.js [34] has gained a lot of traction.

Even though Scala.js is impressively compatible with Scala, faithfully mapping pretty advanced features like method overloading and mixin composition, its support for runtime reflection is very limited. The primary reason for that is the presence of a whole-program optimizer that features global reachability and dead code elimination, aggressive inlining, class hierarchy analysis, stack allocation, and other compile-time optimizations. As it works, the Scala.js optimizer can significantly change the shape of the program, removing methods and entire classes if they end up being unused.

The only pieces of JVM reflection (section 2.1) supported in Scala.js are a few methods that are needed to implement core functionality of the standard library. There exists

a Scala.js linker plugin [33] that provides additional functionality like the ability to instantiate classes and objects by fully-qualified names, but its targets need to be specified explicitly, otherwise the functionality will be unavailable.

This makes it impractical to implement a Scala.js universe for scala.reflect. Instead, the focus of the Scala.js community was to phase out usage of idioms that require runtime reflection and instead do compile-time metaprogramming (chapter 4 and chapter 5).

3.4.6 Native

Scala Native [105] is an emerging LLVM-based backend for the Scala compiler that transforms Scala sources into native code. Its internal architecture is quite similar to Scala.js, with heavy focus on whole-program optimizations and minimal reflection facilities. This makes it impractical to implement a Scala Native universe for scala.reflect.

3.5 Operations

In this section, we perform a practical exploration of the operations supported by the scala.reflect API. In the REPL session below, we will see examples of using the most common functionality of the framework.

We start the exploration with a wildcard import from the runtime universe. This is the easiest way to get acquainted with scala.reflect, building intuitions that can be reused when doing compile-time metaprogramming with macros (chapter 4 and chapter 5).

```
scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> val mirror = runtimeMirror(getClass.getClassLoader)
mirror: Mirror = ...
```

In addition to the functionality provided by `Universe`, we will also be using several extensions implemented in a standalone module called `ToolBox`. Toolboxes are only available for a runtime universe, but their APIs are also available in `Context` that is provided to compile-time metaprograms (section 4.5).

```
scala> import scala.tools.reflect.ToolBox
import scala.tools.reflect.ToolBox

scala> val tb = mirror.mkToolBox()
tb: ToolBox[universe.type] = ...
```

Trees. There are three ways to obtain trees: 1) parse a code snippet expressed as a string, 2) use quasiquotes (subsection 3.3.2), 3) receive a tree from the environment.

```
scala> val someString = "List(42)"
someString: String = List(42)

scala> tb.parse(someString)
res0: Tree = List(42)

scala> q"List(42)"
res1: Tree = List(42)

scala> tq"List[Int]"
res2: Tree = List[Int]
```

Trees can be prettyprinted with: 1) `toString`, which produces a string representation of the tree in a Scala-like notation, 2) `show`, which is similar to `toString`, but provides several flags to control printing of underlying symbols, types, etc, 3) `showRaw`, which exposes the case class structure of the underlying tree and supports flags similar to `show`.

None of these facilities can generate compilable Scala code, because of the desugarings that the Scala compiler performs on abstract syntax trees (subsection 3.2.2).

Note how in the example below the Scala compiler internally models the nullary primary constructor of class `C` with a synthetic method called `<init>`. Desugarings like this significantly complicate prettyprinting. This has led to the development of 4) the `showCode` prettyprinter that does its best to strip off some of the most invasive desugarings.

```
scala> show(q"class C")
res3: String =
class C extends scala.AnyRef {
  def <init>() = {
    super.<init>();
    ()
  }
}

scala> showRaw(q"class C")
res4: String = ClassDef(
  Modifiers(), TypeName("C"), List(),
  Template(
    List(Select(Ident(scala), TypeName("AnyRef"))),
    noSelfType,
    List(
      DefDef(
        Modifiers(), termNames.CONSTRUCTOR,
        List(), List(List()), TypeTree(),
        Block(List(pendingSuperCall), Literal(Constant(()))))))
```

```
scala> showCode(q"class C")
res5: String = class C
```

As explained in section 3.2, abstract syntax trees can be unattributed or attributed. Unattributed trees are obtained from quasiquotes and parsing. Attributed trees are received from the environment (e.g. arguments of def macros are typechecked as per section 4.4) or created by calling `typecheck` on unattributed trees.

```
scala> val list42 = q"List(42)"
list42: Tree = List(42)

scala> list42.symbol
res5: Symbol = <none>

scala> list42.tpe
res6: Type = null

scala> val tlist42 = tb.typecheck(list42)
tlist42: Tree = immutable.this.List.apply[Int](42)

scala> tlist42.symbol
res7: Symbol = method apply

scala> tlist42.tpe
res8: Type = List[Int]
```

It is possible to destructure attributed trees and mix the obtained parts with unattributed trees. This oftentimes happens when a macro author takes apart macro arguments and throws them together with synthetic code. Expertise in compiler internals is required to ensure that the Scala compiler treats these partially attributed trees correctly [14].

Symbols. There are three ways to obtain symbols: 1) load them from a mirror using helper methods like `staticClass` and `symbolOf`, 2) traverse the symbol table via `Symbol.info`, 3) extract them from attributed trees.

```
scala> mirror.staticClass("scala.collection.immutable.List")
res9: ClassSymbol = class List

scala> val list = mirror.symbolOf[List[_]]
list: TypeSymbol = type List

scala> val map = list.info.member(TermName("map")).asMethod
map: MethodSymbol = method map

scala> tlist42.symbol
res9: Symbol = method apply
```

Traversing the symbol table is not limited to just the owner/child relation. Using methods like `ClassSymbol.baseClasses` and `Symbol.overrides`, it is possible to explore other relations between symbols. For example, in the printout below, we can see that `List.map` overrides eponymous methods in `TraversableLike`, `GenTraversableLike` and `FilterMonadic`.

```
scala> map.overrides
res10: List[Symbol] = List(method map, method map, method map)

scala> res9.map(_.owner)
res11: List[Symbol] = List(
  trait TraversableLike,
  trait GenTraversableLike,
  trait FilterMonadic)
```

Symbols can answer a lot of questions about the corresponding definitions. The most common questions involve: 1) testing for binary properties, 2) obtaining signatures or parts of signatures.

```
scala> map.isImplicit
res12: Boolean = false

scala> map.isPrivate
res13: Boolean = false

scala> map.info
res14: Type = [B, That](f: A => B)(implicit bf:
scala.collection.generic.CanBuildFrom[List[A],B,That])That

scala> map.typeParams
res15: List[Symbol] = List(type B, type That)

scala> map.paramLists
res16: List[List[Symbol]] = List(List(value f), List(value bf))
```

Types. There are three ways to obtain types: 1) use helper methods like `Mirror.typeOf`, 2) extract then from attributed trees, 3) receive types from the environment (e.g. a macro impl can access type arguments of the macro application that is currently being expanded). The last approach is environment-specific, so here we will illustrate only the first two.

```
scala> val listint = typeOf[List[Int]]
listint: Type = scala.List[Int]

scala> tlist42.tpe
res17: Type = List[Int]
```

Types can be used to enumerate members. Scala.reflect exposes `Type.decls` to list members defined in that exact type and `Type.members` to enumerate all membes including inherited ones. These methods have singular equivalents that take a `TermName` or a `TypeName` indicating the name of a particular member.

```
scala> listint.members
res18: MemberScope = Scopes(method writeReplace, method toStream,
method stringPrefix, method foldRight, method reverse,
method foreach, method span, method dropWhile, ...)

scala> val map = listint.member(TermName("map"))
map: Symbol = method map
```

An important quirk of scala.reflect is that fact that results of `Type.members` and the like forget about the types they were obtained from. This happens because symbols are singletons, so scala.reflect returns the same `map` regardless of whether it has been obtained from `List[Int]`, `List[String]` or by traversing the members of `List`.

```
scala> typeOf[List[Int]].member(TermName("map"))
res19: reflect.runtime.universe.Symbol = method map

scala> typeOf[List[String]].member(TermName("map"))
res20: reflect.runtime.universe.Symbol = method map

scala> res41 == res42
res21: Boolean = true
```

This leads to a confusing situation when the result of `tpe.member(...).info` forgets about the type arguments provided by `tpe`. A dedicated scala.reflect API, `Symbol.infoIn`, works around this inconvenience of the language model.

```
scala> map.info
res22: Type = [B, That](f: A => B)(implicit bf:
scala.collection.generic.CanBuildFrom[List[A],B,That])That

scala> map.infoIn(typeOf[List[Int]])
res23: Type = [B, That](f: Int => B)(implicit bf:
scala.collection.generic.CanBuildFrom[List[Int],B,That])That
```

Finally, types can be used to explore the type system of Scala. For example, `Type.<:<` checks subtyping, `lub` and `glb` compute least upper and greatest lower bounds, etc.

```
scala> typeOf[List[Int]] <:< typeOf[List[Any]]
res24: Boolean = true

scala> lub(List(typeOf[List[Int]], typeOf[List[Any]]))
res25: Type = scala.List[Any]
```

3.6 Conclusion

Scala.reflect was a breakthrough. When we released it in Scala 2.10, we addressed three long-standing issues with metaprogramming in Scala: 1) absence of a documented way to reliably introspect program structure at runtime (section 2.1 and section 2.2), 2) incompleteness of information provided by `Code` and `Manifest` (section 2.3), 3) high barrier to entry to compile-time metaprogramming (section 2.4).

By taking Scala compiler internals as a baseline and trimming them down to a public API using the design described by Gilles Dubochet [35] (subsection 3.1.1), we implemented a unified metaprogramming API that is available both at compile time and at runtime.

Thanks to this powerful API, we have been able to deliver new ways to metaprogram both at compile time and at runtime. Even though runtime metaprogramming has major usability problems because of the limitations of the underlying platforms (subsection 3.1.4 and subsection 3.4.4), compile-time metaprogramming via macros (chapter 4 and chapter 5) became quite popular in the community.

Unfortunately, this powerful API came at a price. Since the Scala compiler was not written with programmability in mind, using its internals in a public API created several long-standing issues.

First, we had to subject metaprogrammers to occasionally bizarre idioms of the compiler codebase. What was fine for a highly specialized group of compiler developers, ended up being obscure to other Scala programmers, and we had to spend a lot of time providing guidance to early adopters.

We have seen examples of this problem in subsection 3.1.2, where we had to work around an idiosynchrasy of compiler internals in order to check immutability of type members and type parameters. Another example is the fact that during parsing and typechecking compiler internals throw away semantically-insignificant information about programs. Formatting, comments and even entire code snippets disappear as code moves through the compilation pipeline (section 3.2).

Secondly, the architecture of scala.reflect, dictated by the desire to minimize the impact on the compiler, required its users to understand and respect its unconventional ways of organizing code as described in subsection 3.1.1 and subsection 3.1.5.

We suffered from that as well, having to patch several advanced language features just to make the architecture work. Scala.reflect was one of the first active users of dependent method types, also released in Scala 2.10, so we spent a significant amount of time discovering and fixing bugs there. Another experience worth mentioning is a series of improvements to scaladoc, the Scala documentation tool, that could recognize the way how scala.reflect API encodes data structures and emit acceptable documentation for

them - a significant project whose sole motivation was scala.reflect. Also, despite our efforts, some language features like `isInstanceOf` and companionship still do not work correctly with scala.reflect.

Thirdly, an API derived from compiler internals has proven to be very unfriendly to other tools. Hundreds of methods on the API surface and poorly documented invariants expected from the data structures made it virtually impossible for third parties to implement scala.reflect. Even at the time of writing, scala.reflect metaprograms still can not be executed outside the Scala compiler (chapter 7).

In chapter 8, we describe scala.meta - our next attempt at a metaprogramming toolkit for Scala that we are building based on our experiences with scala.reflect. Taking note of the most successful parts of scala.reflect and being aware of the perils of publishing compiler internals, we have embarked on a journey to develop a better metaprogramming experience for the Scala ecosystem.

Scala macros Part II

4 Def macros

Def macros are methods whose applications are expanded at compile time. During expansion, the metaprogram associated with the macro transforms the abstract syntax tree representing the macro application into another abstract syntax tree. Such metaprograms operate with a context, which exposes the code to be expanded and a mirror that reflects the program to be compiled.

This functionality is available in Scala as an experimental language feature since v2.10, building on top of functionality provided by scala.reflect (chapter 3).

In this chapter, we discuss the design of def macros. We start with a quick tour of the macro system (section 4.1, section 4.2), describe the details of the underlying language features (section 4.3), discuss the macro expansion pipeline (section 4.4) and outline the available APIs (section 4.5). We pay special attention to feature interaction, because only in synergy with other features def macros achieve their full potential (section 4.6).

Subsequent chapters talk about further exploration of compile-time metaprogramming in Scala. Several extensions that apply this notion to other aspects of the language are discussed in chapter 5. The most important use cases of Scala macros are explained in chapter 6. Issues with accommodating macros in existing developer tools are outlined in chapter 7. Based on our experience with def macros and their extensions, in chapter 8 and chapter 9, we propose a new metaprogramming framework and a new macro system based that take the best from their predecessors and address pertinent design issues.

Def macros were born from my personal passion, but they would not have been possible without the advice and help of Martin Odersky and joint work with Denys Shabalin, Jan Christopher Vogt, Stefan Zeiger, Adriaan Moors, Paul Phillips and other open-source contributors. Our early adopters - Jason Zaugg, Miles Sabin, Travis Brown and many other brave souls - have also greatly influenced the design.

4.1 Motivation

Def macros were inspired by macros in Nemerle [111], which are themselves in part inspired by Template Haskell [108] and Scheme [41]. This provided a fruitful ground of ideas that we then cultivated to achieve idiomatic look-and-feel and avoid unfortunate feature interactions. As a result, we ended up with the following design goals.

1) Minimize the impact on the language surface. At the time when we were designing def macros, the language had already been around for almost a decade. Therefore, it was a good idea to limit the number of new features to an absolute minimum in order to keep the complexity of the language under control. Making macros look like regular methods was very fitting into this theme.

2) Avoid language fragmentation. In order to prevent def macros from splitting the language into macro-based dialects, we decided that macro arguments and macro expansions must be well-typed. This made it impossible for macro writers to change language syntax and allowed macro users to reason about macros using type signatures.

3) Allow running arbitrary logic during compilation. While there are systems that limit metaprograms to a non-Turing-complete language or to a set of moderated operations of a Turing-complete language, we have opted to use the least restrictive design to increase adoption and expressivity.

4) Provide access to both syntactic and semantic information about the program. While there are systems that only allow syntactic transformations of underlying programs, access to semantics allowed macros to make use of the Scala type system, which was always at the heart of idiomatic Scala experience.

A notable omission from this list of design goals is tool support, which was an honest oversight. We were aware that def macros were partially or fully incompatible with some developer tools (chapter 7), but considered this to be less important that the value that macros can provide. As it turned out, incompatibility with tools was a severe problem that in some scenarios outright prevented adoption. Moreover, it ended up being so challenging that, even at present time, there is no comprehensive tool support for macros. We believe that tool support should have been a crucial design goal from day one.

Another unfortunate omission is hygiene. This was a conscious decision dictated by the need to minimize the conceptual and implementational cost of def macros. Back then, we thought that reify (subsection 3.3.1) was going to be the most frequently used facility to manipulate code snippets. Since reify is hygienic, we did not feel that a dedicated mechanism for hygiene was worth the extra complexity. In reality, it was reify that was not worth it, as it could not handle most of the code manipulation tasks. With reify being phased out, macro writers have to manually take care of hygiene (subsection 3.3.4).

4.2 Intuition

In this section, we will walk through writing and using a def macro that implements a subset of functionality expected of language-integrated queries. Without going into much detail about the underlying mechanisms, we will observe the components of our macro system on a high level.

Language-integrated query (LINQ) is a technique that achieves smooth integration of database queries with programming languages. A common approach to LINQ, popularized by .NET Framework 3.5, consists in representing datasources by collection-like library types and then allowing user to write queries as series of calls to these types using familiar higher-order methods like `map`, `filter` and others.

We will now develop a sketch of a database access library built in the spirit of LINQ. In this sketch, we will intentionally forgo the practicalities of building a LINQ library (e.g. mapping from classes to datasources, design of internal ASTs, error reporting, etc.) in order to clearly illustrate the role played by macros.

Below we define a trait `Query[T]` that encapsulates a query returning a collection of objects of type `T`. Next to it, we define its children `Table` and `Select` that represent particular types of queries. `Table` stands for a datasource that knows about the underlying type (in this sketch, we use type tags (subsection 3.3.3) for this purpose). `Select` models a restricted subset of SQL SELECT statements via a simple `Node` AST.

```
trait Query[T]
case class Table[T: TypeTag]() extends Query[T]
case class Select[T, U](q: Query[T], fn: Node[U]) extends Query[U]

trait Node[T]
case class Ref[T](name: String) extends Node[T]

object Database {
  def execute[T](q: Query[T]): List[T] = { ... }
}
```

In this model, a SQL query string `"SELECT name FROM users"` is represented as `Select(Table[User](), Ref[String]("name"))`. Queries like the one just mentioned can be run via `Database.execute` that translates them into SQL, sends the SQL to the database, receives the response and finally translates it to data objects. In this chapter, we will assume such implementation as a given.

The key aspect of a LINQ facility is a convenient notation for queries. Arguably, none of the aforementioned ways to write queries can be called convenient. SQL, as any string-based representation, is prone to syntax errors, type errors and injections. Explicit instantiation of `Query` objects is very verbose, and still does not address all type errors.

In this sketch, we define a LINQ API in the form of methods that mirror the collection API from the standard library. We would like our users to be able to encode queries in intuitively looking, statically typed calls to this API, e.g. `users.map(u => u.name)`, and then have our library translate these calls into calls to `Query` constructors.

```
object Query {
  implicit class QueryApi[T](q: Query[T]) {
    def map[U](fn: T => U): Query[U] = { ... }
  }
}

case class User(name: String)
val users = Table[User]()
users.map(u => u.name)
// translated to: Select(users, Ref[String]("name"))
```

Among other ways, the desired effect can be achieved with compile-time metaprogramming. A metaprogram that runs at compile time can detect all calls to `Query.map` and rewrite them to invocations of `Select` with parameters of `map` transformed to corresponding instances of `Node`. This is exactly what we are going to do in our sketch.

```
import scala.language.experimental.macros

object Query {
  implicit class QueryApi[T](q: Query[T]) {
    def map[U](fn: T => U): Query[U] = macro QueryMacros.map
  }
}
```

`QueryApi.map` is called a macro def. Since macros are experimental, in order to define a macro def, it is required to either have `import scala.language.experimental.macros` in the lexical scope of the definition or to enable the corresponding setting in compiler flags. This is only necessary to define a macro def, not to use it.

Macro defs look like normal methods in the sense that they can have term parameters, type parameters and return types. Just like regular methods, macro defs can be declared either inside or outside of classes, can be monomorphic or polymorphic, and can participate in type inference and implicit search (an example of the latter we have already seen in subsection 3.1.3). Refer to subsection 4.6.1 for a detailed account of differences between macro defs and regular defs.

Bodies of macro defs have an unusual syntax. The body of a macro def starts with the conditional keyword `macro` and is followed by a possibly qualified identifier that refers to a macro impl, an associated metaprogram run by the compiler when it encounters corresponding macro applications. Macro impls are defined as shown below.

```
import scala.reflect.macros.blackbox.Context

object QueryMacros {
  def map(c: Context)(fn: c.Tree): c.Tree = {
    import c.universe._
    ...
  }
}
```

Macro impls take a compiler context that represents the entry point into the macro API. The macro API consists of a general-purpose metaprogramming toolkit provided by scala.reflect (chapter 3) and several specialized facilities exclusive to macro expansion (section 4.5). A typical first line of a macro impl is `import c.universe._` that makes the entire scala.reflect API available to the metaprogrammer.

In addition to the compiler context, for every term parameter of a macro def, its macro impl takes a term parameter that carries a representation of the corresponding argument of the macro application. As described in section 4.3, macro impls can also get ahold of representation of type arguments, but this functionality is unnecessary for this example.

A macro impl returns an abstract syntax tree, and this AST replaces the original macro application in the compilation pipeline. This is how we are going to perform the LINQ translation of calls to `Query.map`.

```
1  import scala.reflect.macros.blackbox.Context
2
3  object QueryMacros {
4    def map(c: Context)(fn: c.Tree): c.Tree = {
5      import c.universe._
6
7      // c.prefix looks like:
8      // Query.QueryApi[<T>](<prefix>)
9      val q"$_.$_[$_]($prefix)" = c.prefix
10
11     val node: Tree = fn match {
12       case q"($param) => $body" =>
13         body match {
14           case q"$qual.$_" if qual.symbol == param.symbol =>
15             val field = body.symbol.name.decodedName.toString
16             q"Ref[${body.tpe}]($field)"
17         }
18     }
19
20     q"Select($prefix, $node)"
21   }
22 }
```

In the listing above, we handle several challenges of macro writing. First, we get ahold of the prefix of the application, i.e. the `users` part of `users.map(u => u.name)`. Unlike macro arguments, prefixes are not mapped onto parameters of macro impls, so we need to use the dedicated `c.prefix` API (section 4.5) on line 9.

The next challenge is to extract the prefix from the macro application. Since `Query.map` is an extension method, the actual prefix is going to be `QueryApi(users)`, not `users`, therefore we need to apply some non-trivial effort.

One way of getting to the query is to access the corresponding field of the implicit class, doing something like `QueryApi(users).q`. Unfortunately, this is out of the question, because `q` is private, and we cannot make it public without adding an extension method called `q` to `Query`.

Another way of achieving the desired result is to take apart the abstract syntax tree representing `QueryApi(users)`, extracting `users` as the argument of the application. It looks like quasiquotes (subsection 3.3.2), which are supposed to provide a convenient WYSIWYG interface to deconstructing Scala code, are going to be a perfect fit.

Unfortunately for macro writers, the Scala compiler heavily desugars code during compilation, so even the modest `QueryApi(users)` will get to the macro impl in the form of `Query.QueryApi[User](users)`. Therefore the naive `q"$_($query)"` quasiquote is not going to work, and we need to apply additional effort. With a bit of knowledge about compiler internals, we take care of this on line 9.

The final challenge is code generation. The pattern match on lines 11-18 transforms the user-provided lambda expression into an equivalent `Node`. Since our sketch only supports `Ref`, we only support simple lambdas that select a field from a parameter. Finally, on line 20, we produce the macro expansion that has the desired shape.

Note how the ability of def macros to access types dramatically improves user experience in comparison with purely syntactic translation. First, even before `Query.map` gets to expand, the compiler typechecks its argument, making sure that queries are well-typed. Secondly, we have the way to reliably check the shape of the supported lambda. The symbol comparison on line 14 makes sure that the qualifier of field selection refers precisely to the parameter of the lambda and not to something else accidentally having the same name. Finally, we use information about the type of the body on line 16 in order to figure our the mandatory type parameter of `Ref`.

Also note another piece of knowledge about compiler internals that was essential to robust operation of the macro. On line 15, we can not simply call `symbol.name.toString`, because the Scala compiler internally mangles non-alphanumeric names. If the parameter of the lambda has such a name, a simple `toString` will produce unsatisfying results, which is why we have to call `Name.decodedName` first.

Before we conclude, let us highlight a very common metaprogramming mistake that we have just made in `QueryMacros.map`. Def macros are unhygienic, which means that they do not prevent inadvertent name capture, i.e. scenarios like the following.

```
val Select = "hijacked!"
users.map(u => u.name)

// error: too many arguments for
// method apply: (index: Int)Char in class StringOps
//    users.map(u => u.name)
//              ^
```

If the macro user accidentally defines a term called `Select` or `Ref`, our macro is going to stop working with what looks like a nonsensical error message. Because principled tool support was not among the design goals of our macro system, a macro user getting this error is mostly helpless apart from trying to use internal compiler options that print all macro expansions and then going through the resulting wall of text (section 7.2).

One reliable approach to prevent hygiene errors is to use fully-qualified names for external references and to generate unique names for local variables. A somewhat more concise, but potentially much more laborious approach is to explicitly assign symbols (subsection 3.2.3) to references and definitions emitted in macro expansions. This approach often requires extensive knowledge of compiler internals, so it is less frequent in the wild. Common to both of these approaches is that they require explicit attention from metaprogrammers and failures to apply them typically go unnoticed. This is the price to pay for not complicating the compiler with a dedicated mechanism that supports hygiene.

In this section, we wrote a simple macro that implements a sketch of a LINQ transformation. Because applications of def macros are indistinguishable from regular method applications, we were able to transparently expose this transformation to the users. What typically requires dedicated language support in languages like C# or F#, we achieved in a standalone library without any changes to the compiler. Moreover, thanks to def macros being integrated with the type system, our LINQ facility is statically typed.

Unfortunately, even in this simple macro we encountered situations where knowledge of compiler internals (the desugaring of the `QueryApi` application, the fact that non-alphanumeric names are encoded) was essential. We also ran into problems with tool support and hygiene, which demonstrates how important they are to a macro system. These are all common criticisms of def macros, but we believe that they do not invalidate the design per se, just the part that relies on scala.reflect as its metaprogramming API. In chapter 9, we will outline our plan to develop a better macro system based on scala.meta which will be devoid of many of these problems.

4.3 Macro defs and macro impls

Def macros are split into two parts: 1) macro defs that provide type signatures for arguments and expansions, 2) macro impls that provide metaprograms to run against representations of term and type arguments of macro applications. These two parts are linked together via a macro impl reference, i.e. the language construct that constitutes the body of a macro def.

In the listing below, we can see an abridged code example from section 4.2. `QueryApi.map` on line 6 is a macro def. Its body contains a macro impl reference that refers to a macro impl `QueryMacros.map` on line 11.

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.blackbox.Context
3
4  object Query {
5    implicit class QueryApi[T](q: Query[T]) {
6      def map[U](fn: T => U): Query[U] = macro QueryMacros.map
7    }
8  }
9
10 object QueryMacros {
11   def map(c: Context)(fn: c.Tree): c.Tree = {
12     ...
13   }
14 }
```

Macro defs look like regular methods except for their special bodies. They have regular signatures, they can be defined in regular locations, and they can have most of the regular modifiers. Refer to subsection 4.6.1 for a detailed discussion of how macro defs can be defined and what restrictions such definitions have in comparison to regular methods.

Macro impls are regular methods that can contain arbitrary Scala code and can call into arbitrary code. There is a restriction that requires macro impls to be public, static and non-overloaded, but it is introduced for implementation convenience - to make it easier for the macro engine to call into macro impls.

Macro defs and macro impls must be compatible with each other according to the rules specified in subsection 4.3.1 and subsection 4.3.2. In a nutshell, for every term parameter of a macro def, its macro impl must have a corresponding parameter of type `Tree` or `Expr` from the scala.reflect API (subsection 3.2.2). Also, for every type argument of a macro impl reference, its macro impl must have a corresponding type parameter as well as an optional implicit parameter of type `TypeTag` (subsection 3.2.4). Finally, return types have to correspond as well, much like term parameters.

The listing below gives some examples of how compatibility works. In the provided code, all pairs of eponymous macro defs and macro impls are compatible.

```
object Query {
  implicit class QueryApi[T](q: Query[T]) {
    def map1[U](fn: T => U): Query[U] = macro QueryMacros.map1
    def map2[U](fn: T => U): Query[U] = macro QueryMacros.map2[T]
    def map3[U](fn: T => U): Query[U] = macro QueryMacros.map3[T, U]
    def map4[U](fn: T => U): Query[U] = macro QueryMacros.map4[T, U]
  }
}

object QueryMacros {
  def map1
    (c: Context)(fn: c.Tree): c.Tree = ???
  def map2[T: c.WeakTypeTag]
    (c: Context)(fn: c.Tree): c.Tree = ???
  def map3[T: c.WeakTypeTag, U: c.WeakTypeTag]
    (c: Context)(fn: c.Tree): c.Tree = ???
  def map4[T, U]
    (c: Context)(fn: c.Expr[T => U]): c.Expr[Query[U]] = ???
}
```

4.3.1 Term parameters

The first parameter of a macro impl must be a context that provides access to macro APIs. A context is a wrapper over the compile-time scala.reflect universe. In addition to the standard scala.reflect API, contexts also expose some additional functionality that only makes sense in macros (e.g. `Context.prefix` or `Context.abort`) and is therefore not included in the general-purpose part of scala.reflect (section 4.5).

Afterwards, for every parameter of a macro def, a macro impl must have a parameter that will hold an abstract syntax tree of a corresponding argument of a macro application. These parameters can be either exprs or trees. During macro expansion, the macro engine will automatically destructure the macro application and will create the corresponding arguments for the macro impl.

The ability for macro impls to work with exprs is legacy functionality inherited from the initial release of def macros. Back then, the only supported notation for abstract syntax trees was reify (subsection 3.3.1), and it could only work with statically typed trees, i.e. exprs. Therefore, we pretty much had to make macro impls take exprs as parameters and return exprs for consistency, because otherwise macro writers would not be able to use reify. These days, the overwhelming majority of metaprogrammers uses quasiquotes (subsection 3.3.2) that can work with plain trees. Therefore, we made macro

impl signatures more lightweight allowing trees as parameter types and return types, but kept exprs for backward compatibility reasons.

The prefix of the macro application, i.e. the equivalent of `this` for def macros, is not represented as an explicit parameter of a macro def, so we also do not represent it with a parameter of a macro impl. Instead, there is a dedicated macro API - `Context.prefix` - that provides the corresponding expr. An equivalent tree can be obtained by unwrapping that expr using `Expr.tree`.

In order to avoid confusion, we require strict correspondence between the parameters of macro defs and macro impls. For every parameter of a macro def, there must be a parameter with the same name in a macro impl (except for compiler-generated parameters synthesized for some language features, in which case the macro impl is free to choose any name). Additionally, the types of corresponding parameters of macro defs and macro impls must correspond to each other.

The concept of parameter type correspondence is an important part of the macro typechecking algorithm. Since macro impls can take exprs, we need to make sure that parameter types of macro defs correspond to static types of those exprs. For example, if a macro impl takes a parameter `x: c.Expr[Int]`, then the macro def should have a corresponding parameter `x: Int`.

However, implementing this concept turned out to be tricky. Intuitively, it seems sufficient to unwrap the `Expr` in macro impl parameter types and compare the result with the corresponding macro def parameter type, but it is not so simple. The scopes of macro defs and corresponding macro impls are different, so simple type equality does not work.

As a result, our parameter type correspondence algorithm not only checks that the structure of the given types is the same after unwrapping the macro impl parameter type, but also has three special rules to accommodate potential differences in scoping:

1) Type arguments of a macro impl reference correspond to the type parameters of a macro impl. This makes it possible for macro impls to refer to types that are local to the scope of their correspondent macro defs.

2) `this` type of the enclosing class of a macro def corresponds to `c.PrefixType` of the macro impl, where `c` is the context parameter. Even though usages of this rule can be expressed in terms of the previous rule, it makes referring to types defined within an enclosing class - a situation that arises when using cake pattern - much easier.

3) For a macro def parameter `pd` and a macro impl parameter `pi`, `pd.U` corresponds to `pi.value.U`. Whenever `pd` is `T`, `pi` is supposed to be `Expr[T]`, therefore a path-dependent type `pd.U` refers to the same definition as `pi.value.U` (subsection 3.3.1). This makes it possible for macro impls to correspond to macro defs that have dependent method types.

In the listing below, we provide a sketch of how one can could have reimplemented the standard `reify` from the scala.reflect framework. The code showcases a simplified `MyUniverse` cake (line 10) as well as its `MyExprs` slice that introduces `MyExpr[T]` (line 4). Inside `MyUniverse`, we define `reify` as a macro def (line 13) and then try to write an expr-based signature for its macro impl (line 17).

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.blackbox.Context
3
4  trait MyExprs {
5    self: MyUniverse =>
6
7    trait MyExpr[T] { ... }
8  }
9
10 trait MyUniverse extends MyExprs
11                     with ... {
12
13   def reify[T](expr: T): MyExpr[T] = macro MyUniverseMacros.reify[T]
14 }
15
16 object MyUniverseMacros {
17   def reify[T]
18     (c: Context{ type PrefixType = MyUniverse })
19     (expr: c.Expr[T]):
20     c.Expr[c.prefix.value.MyExpr[T]] = { ... }
21 }
```

The monster of a signature on lines 17-20 takes care of the correspondence between the macro def and the macro impl. The main challenge is expressing the path dependence of the result type of `reify` on the type of the prefix of the macro application. In order to achieve that, we capture the type of the prefix on line 18 (rule 2) and then use the type of the prefix in `c.prefix.value` on line 19 (rule 3).

As we can see, the desire to statically type parameters of macro impls leads to significant complications. In theory, exprs and reify provide additional guarantees about metaprograms. In practice, however, the benefit of these additional guarantees is by far overshadowed by the complexity of the correspondence rules and the insufficiency of the functionality provided by reify (subsection 3.3.1).

Therefore, the overwhelming majority of macro impls nowadays are written using trees. This trivializes the term parameter correspondence rules, simplifying lives of metaprogrammers and lowering barriers to entry.

4.3.2 Type parameters

In addition to getting access to term arguments of macro applications, macro impls can also get ahold of representations of types associated with macro applications. Typically, metaprogrammers are interested in type arguments of macro applications, but there are also ways to obtain other types such as type arguments of enclosing classes and traits.

For every type argument of a macro impl reference, a macro impl must have a type parameter. During macro expansion, such a type parameter stands for its corresponding type argument of the macro impl reference, in which the usages of macro def type parameters, enclosing class type parameters, etc are replaced with their instantiations in the corresponding macro application.

Whenever a metaprogrammer is interested in inspecting the type underlying a given type parameter, they declare an implicit type tag (subsection 3.3.3) for that type parameter. During macro expansion, the compiler will automatically compute relevant types and pass them into the macro impl wrapped in a type tag.

If in our running example of a LINQ macro (section 4.2), we wanted to explore the types `T` and `U` associated with `Query.map`, we would declare macro impl as follows.

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.blackbox.Context
3
4  object Query {
5    implicit class QueryApi[T](q: Query[T]) {
6      def map[U](fn: T => U): Query[U] = macro QueryMacros.map[T, U]
7    }
8  }
9
10 object QueryMacros {
11   def map[T, U](c: Context)(fn: c.Tree)
12     (implicit T: c.WeakTypeTag[T], U: c.WeakTypeTag[U]): c.Tree = {
13
14     val typeOfT = T.tpe
15     val typeOfU = U.tpe
16     // do something with typeOfT and typeOfU
17     // ...
18   }
19 }
```

Now, during an example macro expansion of `users.map(u => u.name)`, in the body of the macro impl, `typeOfT` will be a type that represents `User`, and `typeOfU` will represent `String`. Note how the macro engine is capable of figuring out the representation of `T` from the type of `users` which is `Query[T]`.

In order to make the notation more concise, in practice metaprogrammers typically declare implicit parameters via context bounds, e.g. `def map[T: c.WeakTypeTag, U : c.WeakTypeTag](...) = ...`. In that case, implicit parameters can be accessed via `implicitly[c.WeakTypeTag[...]]` or via `weakTypeOf[...]`, a helper function defined in scala.reflect that obtains an implicit type tags and unwraps it.

Type tags may seem like quite a roundabout way of accessing types, and indeed they are. However, much like exprs, type tags are here for backward compatibility reasons. The thing is that reify, because of its statically typed nature, cannot work with plain types and requires statically typed type tags. Back in the day, when reify was supposed to become the main way of manipulating code in macros, we decided to use type tags, not types in the signatures of macro impls.

Unlike with exprs, we have not implemented a simpler way of obtaining types in macro impls. However, this is planned for the future macro system described in chapter 9.

4.3.3 Separate compilation

For implementation convenience, we require that macro applications and their corresponding macro impls are defined in different compilation runs. This is the consequence of us not having a reliable technology to interpret or JIT-compile abstract syntax trees.

At the time of writing, the only way for the compiler to invoke a macro impl is via JVM reflection of precompiled bytecode. If a macro application is compiled together with its underlying macro impl, there is no bytecode for that macro impl yet, which, for the current macro engine, makes macro expansion impossible.

Note that this restriction does not mean that macro defs and their applications must be compiled separately. If a macro impl comes from a different compilation run, then users can utilize corresponding macro defs in the same compilation run where they are defined. Consequently, our macro system can define of macro-generating macros.

Nonetheless, the separate compilation restriction visibly affected the Scala ecosystem. A common situation when this limitation proves seriously inconvenient is when authors of a project want to define domain-specific helper macros to be used in their project. In order to do that, they are forced to split the project into two parts: 1) macro definitions and their transitive dependencies from the original project, 2) the rest of the original project that depends on the first part and uses macros defined there.

Luckily, this limitation does not hurt other macro-based workflows. There is no problem for regular users to depend on third-party libraries that use macros, because third-party libraries are by definition precompiled. Also, there is no need to create separate projects just to test macros, because all build systems in the Scala ecosystem already

compile projects and their tests separately. Finally, every command in the Scala REPL is processed in a separate compilation run, which makes it possible to develop macros in an interactive fashion.

We definitely want to lift the separate compilation restriction, but the technology is not there yet. Hopefully, as scala.meta matures, we will be able to solve this problem once and for all, removing this limitation of our macro system.

4.3.4 Split or merge?

The split between macro defs and macro impls was quite a controversial decision during the early design phase. Let us revisit the example from section 4.2 once again taking note of the syntactic shell of the metaprogram.

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.blackbox.Context
3
4  object Query {
5    implicit class QueryApi[T](q: Query[T]) {
6      def map[U](fn: T => U): Query[U] = macro QueryMacros.map
7    }
8  }
9
10 object QueryMacros {
11   def map(c: Context)(fn: c.Tree): c.Tree = {
12     import c.universe._
13     ...
14   }
15 }
```

A clearly more concise design decision would be to unify the currently disparate macro defs and macro impls, producing something syntactically similar to the code snippet provided in the listing below.

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.blackbox.Context
3
4  object Query {
5    implicit class QueryApi[T](q: Query[T]) {
6      def map[U](fn: T => U): Query[U] = macro {
7        import c.universe._
8        ...
9      }
10   }
11 }
```

Our reasons to go for the split were as follows. First, def macros were first of their kind in production Scala. While previous approaches to compile-time metaprogramming featured advanced techniques like type-level computations and compiler plugins, macros were meant to be widely accessible. Therefore, there were concerns that the unusual behavior of lexical scoping in macro bodies (e.g. the fact that on lines 7-8 `fn` is a tree, not a function) will be too confusing for the users.

Secondly, all macro APIs are exposed via a context. Since the context is not explicitly written down in the more concise design, macro writers would have to conjure it from somewhere. Of course, hardcoding the context to be called `c` like on line 7 is unacceptable. One possibility would be to magically add the context to implicit scope, but that again was deemed to be potentially confusing.

Several years later, Scala users have become pretty familiar with macros, and now we think that the time is right to finally cut the boilerplate. In the new macro system that we are working on, macro defs and macro impls are no longer separated, resulting in a more lightweight look-and-feel (chapter 9).

In the meanwhile, in order to avoid the boilerplate of writing `import c.universe._` again and again when defining multiple macros, metaprogrammers can define macro impls in macro bundles, a feature that is available in the standard distribution since v2.11. Below we can see our running LINQ example that has been expanded to include additional functionality that supported `filter` and modified to host macro impls in the `QueryMacros` bundle.

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.blackbox.Context
3
4  object Query {
5    implicit class QueryApi[T](q: Query[T]) {
6      def map[U](fn: T => U): Query[U] = macro QueryMacros.map
7
8      def filter(fn: T => Boolean): Query[T] = macro QueryMacros.filter
9    }
10 }
11
12 class QueryMacros(c: Context) {
13   import c.universe._
14
15   val q"$_.$_[$_]($prefix)" = c.prefix
16
17   def map(fn: Tree): Tree = {
18     q"Select($prefix, ${toNode(fn)})"
19   }
20
```

```
21    def filter(fn: Tree): Tree = {
22      q"Filter($prefix, ${toNode(fn)})"
23    }
24
25    private def toNode(fn: Tree): Tree = {
26      ...
27    }
28  }
```

In the listing above, we can observe that macro bundles allow to extract a significant portion of boilerplate caused by the split of macro defs and macro impls. Moreover, note how easily we have been able to define and use helpers. As explained in subsection 3.1.5, this is quite involved in the world without macro bundles.

To put it in a nutshell, the decision to split macro defs and macro impls was not an easy one. Prior to introduction of def macros, Scala users did not have much experience with similar functionality, and we decided to be on the safe side, introducing a more verbose, but more straightforward syntax. In practice, users were quite quick to get familiar with the new concepts, and we ended up under pressure to make the syntax more lightweight. We have partially addressed the concerns via macro bundles, and now plan to fix the problem for good in the new macro system described in chapter 9.

4.4 Macro expansion

Macro expansion is part of typechecking. Originally, this was just an implementation detail of the initial prototype of macros, but later we have discovered macros that must expand in typechecker.

Def macros are split into two groups based on how they interact with typechecking. Whitebox macros not only perform tree rewriting, but can also influence implicit search and type inference as described below. Blackbox macros only do tree rewriting, so the typechecker can view them as black boxes, which is how they got their name.

The property of a macro being whitebox or blackbox is called boxity and is expressed in the signature of its macro impl as described in section 4.5. Boxity is recognized by the Scala compiler since v2.11. In Scala 2.10, all macros behave like whitebox macros.

Expansion of whitebox and blackbox macros works very similarly. Unless stated otherwise, the description of the expansion pipeline applies to both kinds of def macros, regardless of their boxity. We will explicitly highlight the situations when boxity changes how things work inside the typechecker.

4.4.1 Expansion pipeline

The typechecker performs macro expansion when it encounters applications of macro defs. This can happen both when these applications are written manually by the programmer, and when they are synthesized by the compiler, e.g. inserted as implicit arguments or implicit conversions.

Sometimes, the compiler does speculative typechecking, e.g. to figure out whether a certain desugaring is applicable or not. Such situations also trigger macro expansion, with the notable exception of implicit search. When a macro application is typechecked in the capacity of an implicit candidate, its treatment depends on its boxity as described in subsection 4.4.2.

When the typechecker encounters a given method application, it starts an involved implementation-defined process that consists in typechecking the method reference, optional overload resolution, typechecking the arguments, optional inference of implicit arguments and optional inference of type arguments. Macro expansion usually happens only after all these steps are finished so that the macro impl can work with a consistent and comprehensive representation of the macro application.

Blackbox macros always work like this. Whitebox macros can also be expanded when type inference gets stuck and cannot infer some or all of the type arguments for the macro application as explained in subsection 4.4.3.

As a result of the decision to typecheck the method reference and its arguments before macro expansion, expansions work from inside out. This means that enclosing macro applications always expand after enclosed macro applications that may be present in their prefixes or arguments. Consequently, macro expansions always see their prefixes and arguments fully expanded.

During macro expansion, a macro application is destructured and the macro engine obtains term and type arguments for the corresponding macro impl according to the rules described in subsection 4.3.1 and subsection 4.3.2. Together with the context that wraps a compile-time mirror and provides some additional functionality as described in section 4.5, these arguments are passed to the macro impl.

A full name of the macro impl is obtained from the macro def and is then used to dynamically load the enclosing class using JVM reflection facilities (the Scala compiler executes on the JVM, so we also use the JVM to run macro impls). After the class is loaded, the compiler invokes the method that contains the compiled bytecode of the macro impl. Despite being one of the original design goals, unrestricted code execution can have adverse effects, e.g. slowdowns or freezes of the compiler, so this is something that we may improve in the future.

All macro expansions in the same compilation run share the same JVM class loader, which means that they share global mutable state. Controlling the scope of side effects while simultaneously providing a way for macros to communicate with each other is another important element of our future work.

A macro impl invocation can finish in three possible ways: 1) normal return that carries a macro expansion, 2) an intentional abort via `c.abort` as described in section 4.5, 3) an unhandled exception. In the first case, expansion continues. Otherwise, a compilation error is emitted and expansion terminates.

Upon successful completion of a macro impl invocation, the resulting macro expansion undergoes a typecheck against the return type of the macro def to ensure that it is safe to integrate into the program being compiled. This typecheck can lead to recursive macro expansions, which can overflow compiler stack if the recursion goes too deep or does not terminate. This is an obvious robustness hole that we would like to address in the future.

Typecheck errors in the expansion become full-fledged compilation errors. This means that macro users can sometimes receive errors in terms of generated code, which may lead to usability problems. This is discussed in more detail in section 7.2.

Finally, a macro expansion that has successfully passed the typecheck replaces the macro application in the compiler AST. Further typechecking and subsequent compiler phases will work on the macro expansion instead of the original macro application.

If the macro expansion has a type that is more specific than the type of the original macro application, it creates a tricky situation. On the one hand, there is a temptation to use the more specific type, e.g. for return type inference of an enclosing method or for type argument inference of an enclosing method call. On the other hand, such macro expansions go beyond what the macro def signature advertises, which may create problems with code comprehension and IDE support (while macro applications that have the advertised type can be treated as regular method applications by the IDEs, macro application that refine the advertised type require dedicated IDE support for def macros).

This situation was the original motivation for the split between whitebox and blackbox macros. We decided that for whitebox macros the typechecker is allowed to make use of more specific type of macro expansions. However, blackbox expansions must have the exact type that is advertised by their corresponding macro def. Internally, this is implemented by automatically upcasting macro expansions to advertised return types.

To put it in a nutshell, macro expansion in our design is a complicated process tightly coupled with typechecking. On the one hand, this significantly complicates reasoning about macro expansion. On the other hand, this enables unique techniques, and at the time of writing one of such techniques is a cornerstone of the open-source Scala community (subsection 6.1.4).

4.4.2 Effects on implicit search

Implicit search is a subsystem of the typechecker that is activated when a method application lacks implicit arguments or when an implicit conversion is required to coerce a term to its expected type.

When starting implicit search, the typechecker creates a list of implicit candidates, i.e. implicit vals and defs that are available in scope. Next, these candidates are speculatively typechecked in an implementation-defined order to validate the fact that they fit the parameters of the search. Finally, the remaining candidates are compared with each other according to the implicit ranking algorithm, and the best one is selected as the search result. If no candidates are applicable, or there are multiple applicable candidates with the same rank, implicit search returns an error.

Since implicit search involves typechecking, it can be affected by macro expansion. In Scala 2.10, we allowed macro expansion during validation of implicit candidates. Therefore, implicit macros were able to dynamically influence implicit search. For example, an implicit macro could decide that it is unfit for a certain implicit search and call `abort` (section 4.5), terminating macro expansion with an error and therefore removing itself from the list of implicit candidates for this particular implicit search.

We have found that such macro-based shenanigans significantly complicate implicit search, which was already pretty complicated prior to introduction of macros. Understanding the scope of available implicits, keeping track of nested implicit searches and backtracks - that was already hard without the necessity to take macro expansions into account.

Therefore, since Scala 2.11, only whitebox macros are expanded during implicit search. Blackbox macros participate in implicit search only with their signatures, just like regular methods, and their expansion happens only after implicit search selects them.

4.4.3 Effects on type inference

When an application of a polymorphic method is missing type arguments - regardless of whether this method is a regular def or a macro def - the typechecker tries to infer the missing arguments. We refer curious readers to [94] for details about Scala type inference, and here we provide just a brief overview.

During type inference, the typechecker collects constraints on missing type arguments from bounds of type parameters, from types of term arguments, and even from results of implicit search (type inference works together with implicit search because Scala supports an analogue of functional dependencies [59]). One can view these constraints as a system of inequalities where unknown type arguments are represented as type variables and order is imposed by the subtyping relation.

After collecting constraints, the typechecker starts a step-by-step process that, on each step, tries to apply a certain transformation to inequalities, creating an equivalent, yet supposedly simpler system of inequalities. The goal of type inference is to transform the original inequalities to equalities that represent a unique solution of the original system.

Most of the time, type inference succeeds. In that case, missing type arguments are inferred to the types represented by the solution.

However, sometimes type inference fails. For example, when a type parameter `T` is phantom, i.e. unused in the term parameters of the method, its only entry in the system of inequalities will be `L <: T <: U`, where `L` and `U` are its lower and upper bound respectively. If `L != U`, this inequality does not have a unique solution, and that means a failure of type inference.

When type inference fails, i.e. when it is unable to take any more transformation steps and its working state still contains some inequalities, the typechecker breaks the stalemate. It takes all yet uninferred type arguments, i.e. those whose variables are still represented by inequalities, and forcibly minimizes them, i.e. equates them to their lower bounds. This produces a result where some type arguments are inferred precisely, and some are replaced with seemingly arbitrary types. For instance, unconstrained type parameters are inferred to `Nothing`, which is a common source of confusion for Scala beginners.

Ever since def macros were introduced, we have been wondering how to use them to allow library authors to customize type inference. After many failed attempts, we found a solution.

If type inference for a macro application gets stuck, the typechecker snapshots its current system of inequalities and produces a partial solution. This solution includes all type arguments that have already been inferred as well as synthetic types that stand for yet uninferred type arguments and are bounded according to the corresponding inequalities. Afterwards, the typechecker performs macro expansion using the partial solution as type arguments for the macro application.

Since macro applications that get this special treatment from the typechecker cannot be viewed as black boxes, partial type inference is only enabled for whitebox macros. Blackbox macros, much like regular methods, have their uninferred type arguments forcibly minimized as described above.

Despite looking quite exotic, this trick plays a key role in the technique of implicit materialization. In subsection 6.1.4, we will see how whitebox implicit macros using this technique can make the typechecker infer unusually precise types, enabling advanced type-level programming.

4.5 Macro APIs

Macro APIs are encapsulated in the context parameter of macro impls (section 4.3). Since Scala 2.11, macro contexts can be declared as `scala.reflect.macros.blackbox.Context` or as `scala.reflect.macros.whitebox.Context`. In Scala 2.10, there is only `scala.reflect.macros.Context` that is equivalent to the whitebox context.

The distinction between the context types is used to indicate boxity of the macro impl (section 4.4). There is also an API difference between blackbox and whitebox contexts, but it consists of compiler internals that are outside of the scope of this dissertation.

The main part of the macro APIs comes from `Context.universe` and `Context.mirror` that expose a scala.reflect universe and mirror corresponding to the current compilation run (chapter 3). However, there are also select bits of functionality that are unique to macros and will be covered below.

First, there are `Context.macroApplication` and `Context.prefix`. The former captures the macro application being expanded, and the latter contains the prefix of the macro application. As we have seen in section 4.2, these APIs complement the obvious way of obtaining arguments of macro applications via parameters of macro impls.

Secondly, contexts support emission of diagnostic messages. It is possible to prematurely terminate macro expansion with a custom error via `Context.abort`, and it is also possible to emit diagnostics of various severities and continue execution (however the latter functionality is much less popular in the community, probably because of the complexity associated with gracefully continuing execution after something went wrong). All such APIs take a position, so that the compiler can correlate a message with a location in the code. This is a powerful way of producing domain-specific error messages.

Thirdly, there are possibilities to tap into compiler internals. Our macro APIs have evolved organically, so now they feature advanced and dangerous functionality, e.g. low-level symbol table manipulation routines that can be used to work around the limitations of the scala.reflect language model. When we will start from scratch in chapter 9, it is likely that none of these APIs will exist, but at the moment we are forced to retain them for backward compatibility.

4.6 Feature interaction

Def macros were designed to become part of an already mature language with an almost ten-year history. A very important consideration was their interaction with existing features of Scala. Uncovering these interactions was akin to a gold rush, with our team and early adopters digging everywhere and every now and then uncovering nuggets of novel techniques. Refer to chapter 6 to learn more about the most successful ideas.

4.6.1 Macro defs

One of the first things that we did when working on def macros was going through existing flavors of methods. For every such case, we tried to change the corresponding regular method into a macro def and considered what is going to happen.

Metadata. Most definitions in Scala, excluding only packages and package objects, allow user-defined annotations that can be read at compile time and/or runtime. Macro defs are no exception. Thanks to their ability to attach custom compile-time metadata to definitions, annotations have been used to share information in situations that involve interplay between macros (section 6.2, [106]).

Modularity. Scala has a rich set of features that allows to manage scope and accessibility of its definitions. All these features work as usual with macro defs. Much like with regular defs, it is possible to define macro defs both in local scope and in member scope of an enclosing definition. If necessary, macro defs can also be private and protected.

Inheritance. One of the key characteristics of regular defs is dynamic dispatch. Scala supports the usual mix of features that support dynamic dispatch: subclassing, overriding, abstractness and finality. Additionally, there is a less mainstream notion of mixin composition and the associated concepts of linearization and abstract override.

Def macros expand at compile time, which means that dynamic dispatch is out of the question, and so are some of the features that make sense only for dynamically dispatched methods. In particular, this means that macro defs cannot be abstract (there is no syntactic possibility for this anyway) and cannot override abstract methods.

Interaction of def macros and overriding has been controversial for quite a while. On the one hand, one can imagine situations when it may seem desirable for a macro def to override a regular method. For instance, authors of a collection library may want to provide an optimized implementation of `Range.foreach` that overrides the inherited iterator-based implementation of `Seq.foreach` with a much simpler loop over the range boundaries. Towards that end, one may declare `Range.foreach` as a macro def that overrides `Seq.foreach` which is a regular def.

On the other hand, while overriding may solve the problem in simple cases when the type of the object is known statically, it does not help if the exact type is only known at runtime. This means that such optimizations are going to be unreliable.

In order to let the community figure out the story of macro-based optimizations, we allowed macros defs to override both regular defs and other macros defs. Unfortunately, this capability does not have much adoption, so we plan to prohibit the new version of macros to participate in overriding (subsection 9.3.1).

Implicits. Depending on the signature, a regular implicit def can serve either as a default value for accordingly-typed implicit parameters or as a conversion between otherwise incompatible types. Both these roles of implicit defs can benefit from compile-time metaprogramming, so we allowed macros defs to be implicit, too. The technique of implicit materialization enabled by implicit macros represents one of the most important use case of def macros at the time of writing (section 6.1).

Constructors. At the time of writing, it is impossible to define a constructor as a def macro. Nonetheless, it is conceivable to want to do so for consistency, because object instantiation is one of the few language constructs that cannot be enriched by macros.

Some time ago, we have proposed a Scala compiler patch that implements support for declaring secondary constructors as macros in the context of [124]. Such macros would expand into calls to other constructors of the underlying class or just regular code. The patch was rejected during code review, because the potential for changing the meaning of `new` was deemed undesirable.

Signatures. Macro defs have exactly the same signature elements as regular defs - name, optional type parameters, optional term parameters, optional return type.

There are no theoretical limitations on the parameters of macro defs (even though declaring a macro def parameter to be by-name does not make any difference in behavior as per subsection 4.6.2, it still makes sense). However, in practice, we disallow macro defs that have default parameters. We did not have time to implement this functionality for the initial release of def macros, and a follow-up patch that introduced it got rejected during code review because of a disagreement that involves compiler internals.

There is, however, a restriction on return type inference. Regular defs can have their return type inferred from the type of their body. Naturally, since macro defs have an unusual body, the usual algorithm is no longer applicable for inferring their return types.

In the initial version of def macros that required macro impls to take and return exprs, we used to infer macro def return types from return types of their corresponding macro impls. For instance, if a macro impl returns `Expr[Query[U]]`, then we could infer the return type of the corresponding macro def as `Query[U]`.

Now when we allow macro impls to return plain trees, return type inference for macro defs is no longer possible in general case. Therefore, we marked this feature as deprecated and do not plan to support it in our future macro system (chapter 9).

Overloading. Like regular defs, macro defs can be overloaded - both with regular defs and other macro defs. This works well, because overload resolution happens before macro expansion as described in section 4.4.

Additionally, since no bytecode is emitted for macro defs (due to def macros only working at compile time), there is no restriction that macro defs must have their erased signature different from erased signatures of other methods.

Types. When it comes to nesting, Scala does not have many limitations. Terms can be nested in types, types can be nested in terms, and a similar story is true for definitions. As a result, some advanced types, namely compound types and existential types, can contain definitions. While existentials can only define abstract vals and abstract types, compound types may include any kind of abstract members.

During a spontaneous discussion with community members, we discovered that the internal compiler structure that represents definitions of compound types can also hold macro defs. By the virtue of using scala.reflect, macros can tap into compiler internals and emit unconventional compound types that contain macro defs. Users of such strange compound types can call those macro defs and trigger macro expansion as if these macro defs were declared in regular classes. This possibility sounds very obscure, but we have been able to successfully use it as a key component in emulation of type providers (section 6.2).

From the discussion above, we can see that macro defs are a relatively seamless extension to regular defs. The differences are: 1) restrictions on overriding, because macro defs do not exist at runtime, 2) almost non-existent return type inference, because macro defs have unusual bodies, 3) inability to be secondary constructors or have default parameters, because macro defs did not initially support this functionality, and our follow-up patches were rejected.

4.6.2 Macro applications

The same analysis that we applied to macro defs can be applied to macro applications. If we go through all flavors of method applications and contexts where method applications can be used, trying to replace usages of regular methods with usages of macros, we obtain the following results.

Fully-specified applications. If a macro application has all type arguments and all term argument lists specified according to the signature of the macro def, then the compiler expands it by calling the corresponding macro impl with these arguments as described in subsection 4.4.1.

Missing type arguments. When a macro application does not have type arguments, and the corresponding macro def does, type inference kicks in before macro expansion happens. Refer to subsection 4.4.3 for a detailed explanation of how type inference works for macro applications.

Partial applications. Regular defs can be partially applied, i.e. can have their applications contain less term argument lists than their corresponding defs. Macro defs also support partial application, but with some restrictions.

Missing implicit arguments lists are discussed below. Missing empty argument lists are appended automatically, exactly like for regular defs. Other cases of partial application involve missing arguments that cannot be inferred. For regular defs, this is handled by eta expansion which converts a partial application into a function object that can be provided with remaining arguments at runtime. However, since def macros expand at compile time, eta expansion for them is prohibited.

Missing implicit arguments. Before macro expansion happens, the compiler makes sure that the macro application has its implicit arguments figured out. If implicit arguments were not specified explicitly, the typechecker launches implicit search to infer them. There are no restriction on where these implicit arguments come from - both regular vals, regular defs and macro defs are allowed.

Missing default arguments. Unlike regular defs, macro defs cannot have default parameters (subsection 4.6.1), so macro applications cannot have default arguments.

Named arguments. Since named and default arguments are implemented by the same subsystem of the typechecker, the fact that default arguments are unsupported also outlaws named arguments.

Vararg arguments. Macro applications can have zero or more arguments corresponding to the same vararg parameter of the macro def. In that case, according to subsection 4.3.1, the macro impl must also have a vararg parameter. The macro engine wraps every vararg argument individually, and then passes the collection of these arguments to the vararg parameter of the macro impl.

By-name arguments. We allow by-name parameters for macro defs, and treat their corresponding arguments as if the by-name modifier was missing. It is the responsibility of the macro writer to respect the evaluation strategy in the macro expansion.

Structural types. When the prefix is a method application has a structural type, and the method is declared only in the refinement of that type, such an application cannot be compiled in a conventional way on the JVM. In this situation, the Scala compiler emits bytecode that uses JVM reflection to dynamically lookup and invoke the required method at runtime. Since such bytecode typically leads to a significant performance degradation, the compiler requires such applications to be enabled by a special built-in import or a dedicated compiler flag. To the contrast, macro applications are expanded at compile time, so invoking macros on structural types is allowed without any special requirements.

Desugarings. An interesting peculiarity of Scala is that a significant number of its language features, e.g. assignment, pattern matching, for comprehension, string interpolation and others, are oftentimes desugared into method applications. As a result, these features can be transparently enriched by macros. This simple idea has many important implications as described in chapter 6.

As we can see, macro applications almost seamlessly integrate with the existing infrastructure of method applications. The differences are: 1) special treatment of whitebox macros by the type inference algorithm, because macro expansion can manipulate type inference, 2) almost non-existent partial application, because macro applications expand at compile time, 3) lack of support for named and default arguments, because macro defs did not initially support this functionality, and our follow-up patches were rejected.

4.6.3 Java interop

An important feature of Scala is bidirectional interoperability with Java. The main target platform of Scala is the JVM, and the Scala compiler emits bytecode that is very close to the what the Java compiler would emit for idioms that have correspondence in Java.

As a result, Scala programs can easily use libraries written in Java (call methods, extend classes and interfaces, etc - as if they were written in Scala). Java programs can also use libraries written in Scala (of course, Scala features like implicit inference will not be available, so the corresponding Java code is more verbose). This is very important practically, because there are popular libraries written in Scala (Akka, Play, Spark, etc) that are also used by Java programmers.

Def macros are one of the rare exceptions to the Java compatibility guideline. Since they operate in terms of a Scala language model, they cannot be realistically supported in a Java compiler. As a result, def macros cannot be used in Java programs.

4.7 Conclusion

Def macros have revolutionized metaprogramming in Scala. Prior to the introduction of macros in Scala 2.10, compile-time metaprogramming with out of the box functionality was either about limited and baroque type-level programming or about comprehensive but unstable and hard to distribute compiler plugins. Def macros have been able to improve upon both, featuring a powerful reflection API and a way to transparently package and distribute compile-time metaprograms.

Even before their public release, def macros have found passionate adopters among the authors of popular libraries. Shortly afterwards, they have widely spread in the Scala community, creating new language idioms and enabling novel use cases (chapter 6).

Def macros owe much of their success to similarity with regular methods (section 4.6). Outside of the context of macros, many existing Scala features are desugared to method calls – either to calls to methods with special names like apply and unapply, or to methods with special meaning like implicits. Def macros make it possible to retain similar user interface and semantics for these existing Scala features, while also gaining code generation and compile-time programmability powers provided by macros.

Another important factor that contributed to the popularity of def macros is the ease of distribution. Most of the functionality enabled by macros was within reach of compiler plugins, albeit in a more verbose way. However, enabling a compiler plugin requires a custom compiler flag, while enabling a def macro does not require any manual configuration. As a result, macro users do not even need to know that they are customizing their compiler, which represents a crucial improvement in user experience.

Scala.reflect (chapter 3) has played a paramount role in making def macros possible. A rich metaprogramming API was essential for usefulness of def macros, but something like that would typically require significant time to design and implement. In our case, such an API was readily available in the form of scala.reflect.

Ironically, scala.reflect also plays a paramount role in the majority of complaints about macros. Overcomplicated API, bizarre coding idioms, lack of hygiene, as well as subpar tool support - these are all direct consequences of using scala.reflect as the underlying metaprogramming framework.

Learning from our experience with def macros, we have set out to build a better macro system that will feature a concise language interface and will be powered by a simple yet comprehensive API supported by third-party developer tools. Central to this new macro system is a new metaprogramming framework called scala.meta (chapter 8), which is not part of the standard distribution, but already shows promise as a potential successor to scala.reflect. In chapter 9, we outline the proposed design, and explain how it improves upon def macros.

5 Extensions

In Scala, a language with rich syntax and static types, program transformations distinguish syntactic categories of terms, patterns, types, definitions and several others.

With def macros desribed in chapter 4, we brought compile-time metaprogramming to the term level. Usefulness of def macros illustrated in chapter 6 has encouraged us to continue experiments with metaprogramming in other linguistic contexts.

In this chapter, we will discuss the results of our most successful experiments. We start with pattern macros in section 5.1, proceed with type macros in section 5.2 and conclude with macro annotations in section 5.3. Not all these experiments ended up in standard distribution like def macros, but all of them influenced the community and the design of future versions of Scala.

Chronologically, these experiments took place before the inception of scala.meta (chapter 8), so in this chapter we will write metaprograms in terms of scala.reflect (chapter 3). Nevertheless, macro flavors that will be discussed here do not depend on a particular metaprogramming framework, and can be reimplemented on top of scala.meta.

5.1 Pattern macros

Pattern macros are macros that expand patterns to patterns, much like def macros (chapter 4) that expand terms to terms. An approximation of pattern macros can be implemented using def macros since Scala 2.11.

In this section, we provide a motivating example that can benefit from pattern macros (subsection 5.1.1) and discuss how to achieve macro expansion in pattern position using def macros (subsection 5.1.2).

5.1.1 Motivation

The listing below defines a console application that tries to parse its first argument as a Scala version using a regular expression. Depending on the result of the attempt, the application either prints out the structural parts of the Scala version (its epoch, major and minor components) or reports an error.

```
1  case class Regex(pattern: String) {
2    def unapplySeq(scrutinee: String): Option[Seq[String]] = {
3      import java.util.regex._
4      val p = Pattern.compile(pattern)
5      val m = p.matcher(scrutinee)
6      if (m.matches) Some(1.to(m.groupCount).map(m.group))
7      else None
8    }
9  }
10
11 object Test {
12   def main(args: Array[String]): Unit = {
13     val ScalaVersion = Regex("""^(\d{1})\.(\d{1,2})\.(.+)$""")
14     args(0) match {
15       case ScalaVersion(epoch, major, minor) =>
16         println((epoch, major, minor))
17       case _ =>
18         println("not a valid Scala version!")
19     }
20   }
21 }
```

In order to parse the input, the application creates a regex on line 13 and uses it as a pattern on line 15 to extract structural parts of a Scala version. In case you are wondering about the particular regex used for this purpose, its last group cannot be a number, because it has to accommodate versions like `1.4.0+3`, `2.9.0-1` and `2.9.1-1-RC1`.

`ScalaVersion` can be used as a pattern because its type defines a magic method `unapplySeq` that returns `Option[Seq[String]]`. The value returned from `unapplySeq` can be either `Some(Seq(...))`, which indicates a successful match and wraps its results, or `None`, which indicates a failed match. If we knew the number of results statically, we could define a magic method `unapply`, whose signature would then wrap a statically-sized collection of results.

More concretely, the Scala compiler will lower the pattern match on lines 14-19 into the following low-level code, which is adapted from a diagnostic compiler printout. In the printout below, note the two synthetic variables `x1` and `o7` introduced by the pattern matcher, as well as several named labels (`case5`, `case6` and `matchEnd4`) that do not have surface syntax, but are present in the intermediate representation of the compiler.

```
1  val x1: String = args.apply(0);
2  case5(){
3    val o7: Option[Seq[String]] = ScalaVersion.unapplySeq(x1);
4    if (o7.isEmpty.unary_!)
5      if (o7.get.!=(null).&&(o7.get.lengthCompare(3).==(0)))
6        {
7          val epoch: String = o7.get.apply(0);
8          val major: String = o7.get.apply(1);
9          val minor: String = o7.get.apply(2);
10         matchEnd4(println((epoch, major, minor)))
11       }
12     else
13       case6()
14   else
15     case6()
16 };
17 case6(){
18   matchEnd4(println("not a valid Scala version!"))
19 };
20 matchEnd4(x: Unit){
21   x
22 }
```

To put it in a nutshell, defining magic methods `unapply` and `unapplySeq`, which are collectively called extractors, allows Scala programmers to customize the default behavior of pattern matching.

Unfortunately, extractors sometimes lack precision. For example, in the compiler printout above, `epoch`, `major` and `minor` are all strings, even though we know that both `major` and `minor` can be represented as integers. Matching these groups into variables of type `Int` may be beneficial for user experience, and that is what we will be trying to achieve in this section.

5.1.2 Pattern expansion via def macros

Without introducing new language features and using only def macros, we will now write an macro-based extractor that computes precise types for groups captured by `Regex`.

Before we proceed with developing a def macro that performs pattern expansion, let us note that control flow in patterns is inverted in comparison with control flow in terms. In method calls, we pass in arguments and get back the output that represents the entire call. In extractor patterns, we pass in a scrutinee that represents the entire pattern and get back a collection of outputs that represent subpatterns. For example, even though the pattern says `ScalaVersion(epoch, major, minor)`, its lowering is `ScalaVersion.unapplySeq(x1)`, and parts are extracted from the result of the call.

Therefore, the macro extractor that we are about to write will not have access to the subpatterns of the `ScalaVersion` pattern, but will instead see a dummy like `x1` that represents the scrutinee.

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.whitebox.Context
3
4  class Regex[T](pattern: T) {
5    def unapply(scrutinee: String): Any = macro RegexMacros.unapply
6  }
7
8  object Regex {
9    def apply(pattern: String): Any = macro RegexMacros.apply
10 }
11
12 class RegexMacros(val c: Context) {
13   import c.universe._
14   import definitions._
15
16   def apply(pattern: c.Tree): c.Tree = {
17     pattern match {
18       case Literal(constant @ Constant(pattern: String)) =>
19         val T = c.internal.constantType(constant)
20         q"new Regex[$T]($pattern)"
21       case _ =>
22         c.abort(c.enclosingPosition, "only literal patterns supported")
23     }
24   }
25   ...
```

In order to communicate the regex between the term that creates a pattern to the pattern that uses it, we parameterize `Regex` with a constant type.

Constant types are a special flavor of singleton types that are inhabited by a single value defined by the underlying constant. Constant types are typically used as phantom type parameters that carry metadata to be used for compile-time metaprogramming, e.g. in type-level computations or macros.

Even though there is no surface syntax for constant types, these types are internally supported by the Scala compiler. This means that macros based on scala.reflect can create values that involve constant types (line 19), and then, by the virtue of type inference, these types can propagate through the program.

```
scala> val ScalaVersion = Regex("""^(\d{1})\.(\d{1,2})\.(.+)$""")
ScalaVersion: Regex[String("^(\\d{1})\\.(\\d{1,2})\\.(.+)$")] = ...
```

Once regexes have precise types that carry patterns used to create them, we can finally write the `unapply` macro that will compute precise types for extracted groups.

```
25    ...
26    def unapply(scrutinee: c.Tree): c.Tree = {
27      val pattern = c.prefix.tree.tpe.widen match {
28        case TypeRef(_, _, List(ConstantType(Constant(s: String)))) => s
29        case _ => c.abort(c.enclosingPosition, "unsupported regex")
30      }
31
32      val types = {
33        ...
34      }
35
36      val resultType = {
37        if (types.isEmpty) typeOf[Boolean]
38        else {
39          val packedResults = {
40            if (types.length == 1) types(0)
41            else appliedType(TupleClass(types.length), types)
42          }
43          appliedType(typeOf[Option[_]], packedResults)
44        }
45      }
46
47      val success = {
48        if (types.isEmpty) q"true"
49        else {
50          val rawResults = 1.to(types.size).map(i => q"m.group($i)")
51          val results = rawResults.zip(types).map{ case (res, tpe) =>
52            if (tpe =:= typeOf[Int]) q"$res.toInt"
53            else if (tpe =:= typeOf[String]) q"$res"
54            else sys.error("unsupported pattern type: " + tpe)
55          }
56          q"Some((..$results))"
57        }
58      }
59
60      val fail = {
61        if (types.isEmpty) q"false"
62        else q"None"
63      }
64
65      q"""
66        new {
67          def unapply(scrutinee: String): $resultType = {
68            import java.util.regex._
```

```
69            val p = Pattern.compile($pattern)
70            val m = p.matcher(scrutinee)
71            if (m.matches) $success else $fail
72          }
73        }.unapply($scrutinee)
74      """
75    }
76  }
```

First, we obtain the underlying regex from the phantom type argument of the pattern (lines 27-30). In order to do that, we get the prefix of the macro application, which represents the `ScalaVersion` part of `ScalaVersion.unapply(x1)`, and then take apart its type. The necessity to call `Type.widen` on the type of the prefix is one of the pieces of arcane knowledge about compiler internals.

Afterwards, we compute the types of the groups defined by the regex (lines 32-34). For brevity, we do not include the implementation of this logic in the listing.

Having the types at hand, we can calculate the signature of the `unapply` method. The shape of the return type of `unapply` varies depending on the number of extracted values. Zero values are represented by `Boolean` that indicates whether there was a match or not. One value is represented by `Option[T]` that wraps the extracted value. Multiple values are represented by `Option[(T1, T2, ...)]` that wraps an appropriately-sized tuple packing the extracted values. This logic is captured on lines 36-45.

This brings our macro, just like most other pattern macros, into the whitebox category. It is an error to use a blackbox macro in pattern position.

Lines 47-63 generate success and failure cases for the matcher according to the `unapply` contract explained above. On lines 52-53, we implement the main purpose of the macro - conversion of matched groups to precise types.

Finally, we assemble the expansion that exposes match results with types computed in the macro (lines 65-74). Note how we expand `ScalaVersion.unapply(x1)` into `<new matcher>.unapply(x1)` instead of simply emitting the match results. Just like the trick with `Type.widen` on line 27, this is yet another internal implementation detail.

For simplicity, we have not gone into practicalities that would have to be handled in a real-world implementation. First, both `apply` and `unapply` are unhygienic. In order for them to be robust, all references to global definitions must be fully qualified and all local definitions must have unique names. Secondly, the implementation is inefficient, because it creates a tuple that the pattern matcher takes apart shortly afterwards. This can be taken care of with name-based pattern matching. Finally, it would be useful to hide the constructor of `Regex`. Regexes created outside of `Regex.apply` are unlikely to be parameterized with a precise constant type, so they will not be usable in `Regex.unapply`.

The diagnostic compiler printout provided below (adapted from real compiler output) demonstrates that our prototype achieves its goal, producing patterns that have precise types computed at compile time.

```
1  val x1: String = args.apply(0);
2  case5(){
3    val o7: Option[(Int, Int, String)] = {
4      ...
5    };
6    if (o7.isEmpty.unary_!)
7      if (o7.get.!=(null).&&(o7.get.lengthCompare(3).==(0)))
8        {
9          val epoch: Int = o7.get._1;
10         val major: Int = o7.get._2;
11         val minor: String = o7.get._3;
12         matchEnd4(println((epoch, major, minor)))
13       }
14     else
15       case6()
16   else
17     case6()
18 };
19 case6(){
20   matchEnd4(println("not a valid Scala version!"))
21 };
22 matchEnd4(x: Unit){
23   x
24 }
```

5.1.3 Conclusion

Using only def macros, we have been able to perform macro expansion in pattern position, obtaining programmatic compile-time control over pattern matching.

The `unapply` macro (subsection 5.1.2) is a great example of how def macros interact with other language features (here, with extractors and constant types) in order to bring compile-time metaprogramming to unexpected domains.

An unpleasant limitation of this approach to customizing pattern matching is the inability to inspect and change subpatterns, which is inconsistent with the functionality provided by def macros. Strictly speaking, we do not expand patterns to patterns, but instead expand extractor calls to extractor calls as part of the implementation-specific desugaring process performed for patterns by the Scala compiler. Since the macro-enabled part of the desugaring of extractor patterns does not involve subpatterns, macros like `unapply` cannot obtain or influence them.

One way to fix this limitation would be to implement pattern macros as a separate compile-time metaproramming facility distinct from def macros. Such pattern macros would receive subpatterns as arguments and then produce a resulting pattern upon macro expansion.

Unfortunately, this design requires non-trivial changes in the macro engine. First, much like control flow, typechecking of patterns is also inverted in comparison with typechecking of terms. Unlike method arguments which are typed before the enclosing method call, subpatterns are typed after the enclosing extractor call. This means that subpatterns, as seen inside the hypothetical pattern macros, would be untyped, which is a noticeable departure from def macros. Secondly, Scala's pattern matcher works in a very specific way that is hard to customize, so implementing pattern macros would require a major engineering effort.

Faced with significant complications, we decided to temporarily suspend the investigation of a dedicated design for pattern macros. This remaining an interesting challenge for future work.

5.2 Type macros

Type macros are macros that expand types to types, much like def macros (chapter 4) that expand terms to terms. Unlike pattern macros, type macros cannot be emulated by existing language features, so we implemented them as a dedicated feature in an experimental fork of Scala 2.10 called "Macro Paradise v1" [11].

In this chapter, we outline an important use case that can benefit from type macros (subsection 5.2.1) and demonstrate how to achieve it (subsection 5.2.2).

5.2.1 Motivation

Type providers [116] originate from the F# community and provide a mechanism for compile-time generation of statically-typed wrappers for datasources.

Along with LINQ, type providers have proven to be one of the foundational facilities to simplify data access, so one of our goals with macros has been to provide analogous functionality for the Scala ecosystem.

After tackling LINQ (section 4.2), we moved our focus to type providers and found that macro expansion on the type level seems to be a good fit. Expanding type applications to types in order to generate top-level definitions was the main motivation for developing type macros.

5.2.2 Type expansion

Below we define a type macro `H2Db` that generates case classes representing tables in an H2 database and provides CRUD functionality for the underlying tables.

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.Context
3
4  object TypeProviders {
5    type H2Db(url: String) = macro TypeProviderMacros.h2db
6  }
7
8  object TypeProviderMacros {
9    def h2db(c: Context)(url: c.Tree): c.Tree = { ... }
10 }
11
12 object Test {
13   def main(args: Array[String]): Unit = {
14     import TypeProviders._
15     object Db extends H2Db("coffees")
16
17     val brazilian = Db.Coffees.insert("Brazilian", 99.0)
18     Db.Coffees.update(brazilian.copy(price = 100.0))
19     println(Db.Coffees.all)
20   }
21 }
```

Type macros are a hybrid of def macros and type members. On the one hand, type macros are split into macro types and macro impls, with macro types being very similar to macro defs and macro impls being the same as regular macro impls (section 4.3).

Note how the `H2Db` macro type has a term parameter unlike a regular type and like a macro def (line 5), and how it is applied to a term argument exactly like a macro def (line 15). The only two differences between macro types and macro defs is that macro types cannot have return types, because they do not expand into terms, and that macro types cannot be implicit, because `implicit` does not make sense for types.

On the other hand, type macros belong to the namespace of types and, as such, they can only be used in type positions and can only override other types and type macros.

In the listing above, macro expansion happens when the compiler encounters an application of a macro type on line 15. As a result, analogously to def macros, that application is typechecked, the macro impl is looked up and called, and so on.

The only difference with the def macro expansion pipeline (section 4.4) is the way how macro expansions are typechecked. Since type macros expand into types, their macro

expansions are typechecked as types. Moreover, even though we validate def macro expansions against the return type of their macro def, we do not do that for type macros. Since kinds in Scala are only capable of checking arities of type constructors and their parameters, which does not seem particularly useful for type macro expansions, we decided to to skip the validation step for type macros.

After `H2Db("coffees")` expands, the compiler replaces it with its expansion, and for all intents and purposes the compiler considers that `Db` subclasses the type produced by the macro expansion. In our running example, this means that users of `Db` can refer to synthetic definitions like `Coffees` that have been generated by the type macro.

```
1  def h2db(c: Context)(url: c.Tree): c.Tree = {
2    import c.universe._
3    import definitions._
4    import java.sql._
5    import JdbcHelpers._
6
7    val connString = url match {
8      case Literal(Constant(url: String)) =>
9        ...
10     case _ =>
11       c.abort(c.enclosingPosition, "only literal urls supported")
12   }
13
14   val tableDefs = {
15     Class.forName("org.h2.Driver")
16     val conn = DriverManager.getConnection(connString, "sa", "")
17     try {
18       val tablesq = conn.list("show tables")
19       val tables = tablesq.map(row => row("table_name").to[String])
20
21       tables.flatMap(tbl => {
22         val schemaq = conn.list(show columns from " + tbl)
23         val schema = schemaq.map(row => {
24           val columnName = row("column_name").to[String]
25           val scalaName = TermName(columnName.toLowerCase)
26           val scalaType = {
27             val tpe = row("type").to[String]
28             if (tpe.startsWith("INTEGER")) typeOf[Int]
29             else if (tpe.startsWith("VARCHAR")) typeOf[String]
30             else if (tpe.startsWith("DOUBLE")) typeOf[Double]
31             else c.abort(c.enclosingPosition, "unsupported type: " + tpe)
32           }
33           scalaName -> scalaType
34         })
35
```

```
36          val dtoName = TypeName(tbl.toLowerCase.capitalize.dropRight(1))
37          val dtoFields = schema.map{ case (n, t) => q"val $n: $t" }
38          val dtoClass = q"case class $dtoName(..$dtoFields)"
39
40          val tableName = TermName(tbl.toLowerCase.capitalize)
41          val insertParams = dtoFields.filter(f => f.name.toString != "id")
42          val tableObject = q"""
43            object $tableName {
44              def all: List[$dtoName] = { ... }
45              def insert(..$insertParams): $dtoName = { ... }
46              def update(dto: $dtoName) = { ... }
47              def remove(dto: $dtoName) = { ... }
48            }
49          """
50
51          List(dtoClass, tableObject)
52        })
53      } finally {
54        conn.close()
55      }
56    }
57
58    val pkgName = c.enclosingPackage.pid.toString
59    val className = c.freshName(TypeName("Db"))
60    c.introduceTopLevel(pkgName, q"class $className { ..$tableDefs }")
61    Select(c.enclosingPackage.pid, className)
62  }
```

Similarly to previous examples of macros, we forgo practicalities for brevity. First, we omit the code that creates a database connection string from the argument of the type macro (line 9). Moreover, we do not take any measures to improve the performance of the macro by connection pooling, caching expansion results, etc. Additionally, we use a naive approach to configuration, hardcoding database credentials (line 16) and name mapping rules (lines 25, 36, 40 and 41). Finally, we are not concerned with hygiene.

On a very high level, the `h2db` macro impl connects to a database at compile time (line 16), generates a synthetic base class that contains the definitions representing the database (lines 17-55), inserts the synthetic class into the current package via the newly introduced `Context.introduceTopLevel` API (line 60) and then expands into a reference to that synthetic class (line 61).

From the discussion above, we can see that type macros can implement functionality similar to type providers. While our running example does not support lazy and/or erased code generation, similar behavior can be achieved with additional help from def macros as described in section 6.2.

5.2.3 Conclusion

Type macros brought the notion of macro expansion to the type level. Thanks to them, it has become possible to implement eager type providers, making a significant step ahead in the data access story in Scala. Unfortunately, along with new possibilities, type macros also created new complications.

The most controversial addition to the macro system was the idea of non-local expansion. While def macros can only change their applications by rewriting them into something else, type macros, in addition to local rewritings, can also create top-level definitions via `introduceTopLevel`.

`introduceTopLevel` has provided a long-requested functionality of generating definitions that can be used outside macro expansions. However, metaprogrammers have quickly discovered that `introduceTopLevel` is dangerous. Top-level scope is a resource shared between the typechecker and user metaprograms, so mutating it with `introduceTopLevel` can lead to compilation order problems. For example, if one file in a compilation run relies on definitions created by a macro expansion performed in another file, compiling the former before the latter may lead to unexpected compilation errors.

Moreover, our tooling ecosystem turned out to be unexpectedly incompatible with the notion of programmatically generated top-level definitions. One of the most spectacular incompatibilities was associated with sbt, the most popular Scala build tool, which insisted on going into infinite recompilation loops when type macros were involved. Because of the particular way how sbt implemented detection of dependencies, we have not ever been able to patch it to fully support synthetic top-level definitions.

Additionally to the difficulties associated with `introduceTopLevel`, type macros have made a noticeable impact on the complexity budget of the language and the macro engine. In order for type macros to be practically useful, they required additional infrastructure that can, among other things: share state between expansions, perform caching of expansions, enable lazy expansion in order to model huge schemas, and provide access to resources of the project.

On these grounds, our Scala compiler patch that implements support for type macros was rejected, and we tried to salvage the underlying ideas in our future experiments. The most successful of these experiments are macro annotations that provide a principled way of generating top-level definitions (section 5.3).

5.3 Macro annotations

Macro annotations are macros that expand definitions to definitions, much like def macros (chapter 4) that expand terms to terms. We implemented macro annotations in a popular compiler plugin called "Macro Paradise v2" [12] available since Scala 2.10.

In this section, we explain the motivation behind macro annotations (subsection 5.3.1) and illustrate how macro annotations perform definition expansion (subsection 5.3.2). Additionally, we discuss to what degree macro annotations can introspect other definitions in the enclosing program (subsection 5.3.3, subsection 5.3.4).

5.3.1 Motivation

The main motivation for developing macro annotations was support for boilerplate generation that involves top-level definitions.

Even though both def macros and type macros can achieve similar functionality, they are not general enough. Def macros can expose local definitions with the help from structural types, but that approach uses fringe Scala features and has practical limitations (section 6.2). Type macros can make modules inherit from synthetic traits created by `introduceTopLevel`, indirectly introducing top-level definitions into the affected modules (subsection 5.2.2). However, this approach can only create new definitions, but not rewrite or even see existing ones.

Languages of the .NET platform typically support attributes that annotate definitions and provide static metadata for their annottees. Nemerle extends this notion by making it possible for specially defined attributes to transform annotated definitions. In the example below, taken from [110], `Proxy` is a macro attribute that enumerates the list of methods in the interface `IMath` and, for every such method, generates a forwarder that calls this method on `math`.

```
// Remote "Proxy Object"
class MathProxy : IMath
{
  // the stubs implementing IMath by calling
  // math.* are automatically generated
  [Proxy (IMath)]
  math; // object of type Math

  this()
  {
    math = ObtainExternalReferenceToMath ();
  }
}
```

Inspired by Nemerle, we have taken the existing concept of Scala annotations and provided a way for metaprogrammers to define them in a special way in order to obtain definition-transforming macros.

5.3.2 Definition expansion

In the listing below, we can see a definition of the `h2db` macro annotation that is observationally equivalent to the `H2Db` type macro developed in subsection 5.2.2.

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.Context
3  import scala.annotation.StaticAnnotation
4
5  class h2db(url: String) extends StaticAnnotation {
6    def macroTransform(annottees: Any*): Any =
7      macro H2DbMacros.macroTransform
8  }
9
10 object H2DbMacros {
11   def macroTransform(c: Context)(annottees: c.Tree*): c.Tree = {
12     ...
13   }
14 }
15
16 object Test {
17   def main(args: Array[String]): Unit = {
18     @h2db("coffees") object Db
19     val brazilian = Db.Coffees.insert("Brazilian", 99.0)
20     Db.Coffees.update(brazilian.copy(price = 100.0))
21     println(Db.Coffees.all)
22   }
23 }
```

Macro annotations are subclasses of the `StaticAnnotation` trait from the standard library that define a macro with the magic name `macroTransform` (line 5-7). The macro impl that corresponds to that macro def takes a synthetic block of code that wraps the annottees and returns a synthetic block of code that wraps the expansion (line 11-13).

This way of defining macro annotations involves quite a lot of boilerplate, but it was dictated by our desire to retrofit macro annotations onto the existing infrastructure of def macros. The future version of macro annotations (chapter 9) will have a much more lightweight syntax.

Macro annotations can be applied to definitions that allow annotations, which currently includes all Scala definitions except packages and package objects. When applied to

traits and classes, macro expansion involves not just the annottees themselves, but also their companion objects if they exist. Therefore, implementations of macro annotations (e.g. the one on lines 11-13) take a collection of definitions instead of a single definition. Analogously, expansions of traits and classes can return not just the rewritten annottees, but can also create or rewrite their companion objects.

Expansion pipeline for macro annotations reuses the standard expansion pipeline implemented for def macros (subsection 4.4.1), but overrides it in several key points.

First, unlike def macros, which expand during typechecking, macro annotations expand before typechecking. As a result, macro annotations take untyped definitions and expand them into untyped definitions, which gives them freedom to change type signatures of their annottees (e.g. add or remove parameters to annotated methods, add or remove members to annotated classes, etc). Consequently, unlike def macros, macro annotations expand outside-in.

The untyped nature of macro annotations brings additional flexibility, but it also reduces the set of tools available to the metaprogrammer, because untyped trees do not know what their identifiers resolve to, do not know about their types, etc.

Secondly, again unlike def macros, resulting expansions are not typechecked right away, because that would be too early for the compilation pipeline. Eventually, at a later point in time when the pipeline reaches the typechecking step, these expansions get typechecked and possible errors in them are reported to the users.

Finally, there are certain rules that govern the shape of allowed expansions. Top-level definitions can only expand into eponymous top-level definitions, e.g. it is not allowed for `@ann class C` to expand into `class D`. This restriction is in place in order to prevent compilation order issues as explained in subsection 5.3.4. To the contrast, nested definitions (i.e. class/trait/object members or local definitions) can expand into anything, including multiple definitions or nothing at all.

```
1  def macroTransform(c: Context)(annottees: c.Tree*): c.Tree = {
2    import c.universe._
3
4    val connString = c.macroApplication match {
5      case q"new h2db(${url: String}).macroTransform(..$_)" =>
6        ...
7      case _ =>
8        c.abort(c.enclosingPosition, "only literal urls supported")
9    }
10
11   val tableDefs = {
12     // source code taken from subsection 5.2.2
13     ...
```

```
14    }
15
16    val annottees1 = annottees match {
17      case Seq(ModuleDef(mods, name, Template(parents, self, body))) =>
18        val body1 = body ++ tableDefs
19        Seq(ModuleDef(mods, name, Template(parents, self, body1)))
20      case _ =>
21        c.abort(c.enclosingPosition, "unsupported annottee")
22    }
23
24    Block(annottees1, Literal(Constant(())))
25  }
```

In the implementation of the `h2db` macro annotation, we reuse most of the logic that we developed for the `H2Db` type macro (subsection 5.2.2), only changing the shell of the metaprogram that delivers the generated definitions.

The first change in comparison with `H2Db` affects how the macro obtains its configuration. In the case of type macros, url is the parameter of the macro type, so it is trivially available as a parameter of the macro impl. In the case of macro annotations, in order to reuse the existing def macro infrastructure, we model macro expansions via expansions of the corresponding `macroTransform` def macros. Our implementation creates synthetic invocations that look like `new h2db(<url>).macroTransform(<object ...>)` and feeds them into the standard expansion pipeline. As a result, arguments of macro annotations are only available via `Context.prefix` or `Context.macroApplication` as shown on line 5.

The second change in comparison with `H2Db` affects how generated definitions are returned in the macro expansion. We can observe that on lines 16-24. First, `macroTransform` matches the list of annottees, expecting a single `object` on line 17 (objects are internally called `ModuleDefs` in the Scala compiler parlance). After obtaining the annottee, it takes the generated table defs and inserts them into the body of the annottee on line 19. Finally, it wraps the transformed `object` in a block on line 24 according to the convention that macro annotations have to follow.

To put it in a nutshell, macro annotations can generate and/or rewrite top-level definitions in a relatively straightforward way. This successfully fills the compile-time metaprogramming niche uncovered by def macros and type macros.

5.3.3 Enabling semantic APIs

One of the key design decisions behind macro annotations was making their expansions untyped. More precisely, neither annottees nor expansions of macro annotations are typechecked in the macro annotation expansion pipeline.

On the one hand, this increased flexibility of macro annotations and made them useful to design extremely concise DSLs and prototype new language features.

In the listing below, we can see the `@enum` macro annotation sketched by Simon Ochsenreither that provides a concise notation for defining the enumeration representing days of week. Note that if `object Day` was typechecked prior to the expansion of `@enum`, this notation would not work well, because it is unlikely that the identifiers in the body of the object would typecheck.

```
@enum
object Day {
  Monday
  Tuesday
  Wednesday
  Thursday
  Friday
  Saturday
}
```

On the other hand, the decision to be untyped created a complication for macro annotations that want to obtain type information about certain parts of their annottees.

For example, one may want to write a macro annotation `@async` that goes through the list of methods of a given class, trait or object and generates their asynchronous versions that wrap execution in futures. In order to avoid creating asynchronous versions of pre-existing asynchronous methods, such a macro would need a way to determine whether a method returns `Future[T]` or not. Since Scala has renaming imports and return type inference, simply inspecting the syntax of the return type trees is not good enough to achieve robustness.

```
@async
object Server {
  def longRunningOperation1() = { ... }
  def longRunningOperation2() = { ... }
  ...
}
```

In order to support both untyped and typed use cases, we expose the `Context.typecheck` API in macro annotations. On a very high level, this method takes an abstract syntax tree and then typechecks it in the context of the current macro expansion, producing a typed tree that supports semantic APIs.

As a result, metaprogrammers have a choice. If they want to have completely untyped annottees, they do nothing. If they want the annottees partially typechecked, they call `Context.typecheck` on the trees they are curious about.

5.3.4 Ensuring consistent compilation results

`introduceTopLevel` was frequently criticized for its ability to introduce compilation order problems.

With `introduceTopLevel`, it was not uncommon for inexperienced metaprogrammers to use it to define top-level definitions and then to refer to these definitions elsewhere in the program. Naturally, this only works if the macro expansion introducing the definition happens before the typechecker gets to a reference to the definition. In other words, this requires the compiler to typecheck the file with the macro application before the file with the reference to the synthetic definition.

There is no language-integrated way to guarantee that a given file will be typechecked before another file. In order to obtain such a guarantee, programmers have to manually configure their build tools to pass files to the Scala compiler in the correct order. This is a very brittle workflow, so it is strongly discouraged in the community.

`introduceTopLevel` only works well in macros like `H2Db` (subsection 5.2.2), where it creates a definition with an obscure unique name that is only used in the enclosing macro expansion. As a result, even though `introduceTopLevel` can be useful, it is not very good as an API, because it gives metaprogrammers room for error.

In a sense, macro annotations are similar to `introduceTopLevel`, because both can introduce definitions that are visible to the entire compilation run. Therefore, when designing macro annotations, we developed them to completely prevent possibilities for compilation order problems.

Concretely, our goal was to avoid two kinds of situations that can lead to unexpected behavior when developing and using macro annotations.

The first situation involves spuriously invalid references, when a reference in one file incorrectly considers that a definition generated by a macro annotation in another file does not exist. In the example below, class `A` depends on class `BB` generated in a different file by a macro annotation that changes names of its annottees by repeating them twice. We want such a program to either consistently succeed or consistently fail compilation, regardless of the compilation order.

```
// A.scala
class A extends BB

// B.scala
@double class B
```

The second situation involves spuriously valid references, when a reference in one file incorrectly considers that a definition deleted by a macro annotation in another file does

exist. In the example below, class `A` from `A.scala` refers to class `B` from `B.scala`, but that reference is invalid, because class `B` gets deleted by the macro expansion. We want such a program to consistently fail compilation with a legible error, regardless of the compilation order.

```
// A.scala
class A extends B

// B.scala
@double class B
```

Additionally, our design space was limited by the architecture of the compiler. Just like in other projects described in this dissertation, our goal was to ship macro annotations with the Scala compiler, so our implementation had to work within the available compiler framework.

Within the `scalac` frontend, definitions can be in one of the following four states: unattributed, entered, completed and typechecked. After parsing, all definitions are unattributed. Afterwards, as the compilation proceeds, definitions are entered, i.e. assigned newly created symbols which are then added to the internal symbol table of the compiler. Such newly created symbols do not know their signatures yet - they just know how to compute them on demand. Further in the future, when their signatures are required, definitions are completed, i.e. have their signatures computed. Finally, definitions are typechecked, i.e. have their trees attributed by the typechecker.

To put it shortly, the frontend of the Scala compiler consists of three phases: `parser`, `namer` and `typer`, each of which sequentially processes source files in the order they were provided to the compiler. The `parser` phase parses all source files into abstract syntax trees. Then, the `namer` phase enters all top-level definitions in all source files. Finally, the `typer` phase goes through the source files from top to bottom and recursively typechecks their abstract syntax trees. Typechecking may trigger completion of other definitions, which may trigger entering new symbols, and so on. Eventually, `typer` finishes going through the source files, at which point all definitions should be typechecked.

In this architecture, expansion of macro annotations can happen either upon entering or upon completion. During typechecking, the signature of the definition is already finalized, which means that typechecking cannot accommodate general-case macro annotations. Let us explore the available opportunities.

Expand on enter. In this case, the first situation is going to work well. Even though `B` in the first situation is going to expand into `BB`, that expansion happens on enter. Since completion can be triggered only after all definitions in scope have been entered, regardless of the compilation order, completion for `A` will successfully see an entry for `BB` in the symbol table, as desired by the design requirements.

Moreover, in the second situation, by the time when `A` starts completing, `B` will have already been renamed by the enclosing macro annotation, again regardless of the compilation order. Therefore, name resolution for `B` will result in a compilation error, which is the desired behavior.

However, when we expand on enter, we have problems with `typecheck`. Typechecking includes name resolution, therefore our macro annotations can trigger name resolution during expansion, i.e. during enter. As a result, typechecks may see inconsistent state of the symbol table. For example, if in the first situation we compile `B.scala` before `A.scala` and the `@double` macro annotation tries to typecheck something that involves `A`, that typecheck will result in a spurious failure. This happens because expansion of `@double` triggers upon enter on `B`, and by that time `A` will not have been entered yet.

Expand on complete. In this case, the first situation is not going to work consistently. If `A.scala` is compiled before `B.scala`, `A` will complete before `B` completes, which means that by that time `@double` will not have expanded yet and `BB` will not exist, resulting in a compilation failure. If we change the compilation order, then macro expansion happens before completion of `A`, and compilation succeeds.

Regardless of problems with the first situation, the second situation is still going to consistently fail compilation. If `A.scala` is compiled before `B.scala`, completion of `A` is going to typecheck a reference to `B`, which will succeed and then try to complete `B`. This recursive completion will trigger macro expansion that will rename `B` and invalidate the completion and failing the compilation. If `B.scala` is complied before `A.scala`, then typechecking a reference to `B` will fail in the first place, again correctly failing compilation.

Unfortunately, `typecheck` is still problematic. The issue is that even though `typecheck` sees a full snapshot of the symbol table at the moment of expansion, such snapshot may change between macro expansions. As a result, changes in expansion order (which may be caused by changes in compilation order) may cause changes in expansions.

To put it in a nutshell, bringing `typecheck` into the mix presents a very challenging design problem. None of the available expansion schemes work well with allowing macro annotations to typecheck their annottees.

When designing the first version of macro annotations, we were very determined to ship `typecheck`. As a result, we had to work out some compromises, and our current expansion scheme represents a hybrid solution.

For top-level definitions, we expand on complete, ruling out the first situation by prohibiting expansions that add or remove definitions. This restriction fixes the problem with `typecheck`, because it guarantees that the top-level slice of the symbol table does not change during macro expansion.

For nested definitions, we expand on enter, allowing unrestricted macro expansions for maximum flexibility. In order to make `typecheck` independent from expansion order, we prohibit it from seeing definitions on the same level of nesting. In short, we avoid problems with soundness by creating a limitation in `typecheck`.

From the discussion above, we can see that expansions of macro annotations were designed to not depend on compilation order. However, this was not easy to achieve. In order to accommodate semantic APIs without sacrificing consistent expansions, we had to introduce several arbitrarily-looking restrictions.

5.3.5 Revisiting the separate compilation restriction

Since macro annotations reuse the def macro infrastructure, they inherit its separate compilation restriction. Concretely, usages of macro annotations and the underlying macro impls must come from different compilation runs, because otherwise the macro engine will not be able to execute macro impls during macro expansion (subsection 4.3.3).

However, in addition to this restriction, macro annotations need another one. Unlike in the situation with def macros, usages of macro annotations and definitions of macro annotations must be compiled separately. Consequently, macro annotations cannot generate other macro annotations to be expanded in the same compilation run.

This restriction originates from the observation that name resolution is required to expand macro annotations. Indeed, when the Scala compiler sees code like `@ann class A`, it must understand that `@ann` is a macro annotation and for that it needs to resolve `ann`.

Now, note that we can view name resolution as a mini-`typecheck`, which means that it has the same restrictions with respect to the expansion scheme as discussed in subsection 5.3.4. Therefore, since our current scheme involves expand on enter, we cannot reliably resolve names of macro annotations.

Luckily, if we require that macro annotations must be precompiled, the conundrum gets resolved. Expand on enter only affects definitions that come from the same compilation run, so by ignoring such definitions we sidestep the problem.

To put it in a nutshell, the separate compilation restriction imposed on macro annotations is much more harsh than the analogous restriction on def macros. Even if it would be possible to solve the latter by, for example, interpreting macro impls, solving the former will also require a redesign of the macro annotation expansion scheme.

5.3.6 Conclusion

Macro annotations provide a robust way to generate top-level definitions. This capability comes in handy when experimenting with new language features and scrapping the boilerplate associated with hard-to-abstract coding idioms.

In comparison with type macros (section 5.2) that were previously playing a similar role in the Scala macro ecosystem, macro annotations: 1) have a much narrower focus, which makes them more tractable, 2) apply not just to classes, traits and objects, but to arbitrary definitions, which makes them more universal, 3) do not need an additional `introduceTopLevel` API, which makes them more maintainable.

Even though macro annotations have not yet been included into the Scala compiler, they are a popular approach to boilerplate generation. At the moment of writing, the Macro Paradise plugin that implements macro annotations clocks more than twenty thousand downloads per month.

Our biggest technical challenge came from the desire to both allow unrestricted expansions of definitions and enable semantic APIs during these expansions. In our experiments with expansion schemes for macro annotations within the current compiler architecture, no scheme could simultaneously provide: a) independence of macro expansions from compilation order, b) arbitrary shapes of definitions-to-definitions expansions, c) ability to call semantic APIs.

In the current implementation of macro annotations, we decided that it is important to expose semantic APIs, but for that we had to sacrifice simplicity of expansions. Therefore, in our new macro system (chapter 9) we are planning to restrict the functionality of macro annotations to syntactic APIs, turning semantic APIs into an exclusive feature of def macros.

6 Case studies

Macros (chapter 4 and chapter 5) have tapped into the latent demand for language-integrated compile-time metaprogramming in the Scala community. Even before macros were released as part of the standard distribution, authors of some of the top open-source libraries started experimenting with them, discovering new idioms and using them in their projects. Nowadays, macros have established themselves as a tool of choice for many scenarios involving code generation, program analysis and domain-specific languages.

In this chapter, we document some of the use cases where Scala macros have made a significant impact, either enabling new idioms or providing non-trivial improvements to existing approaches in Scala.

We start with materialization (section 6.1), a novel technique that enables programmable derivation of typeclass instances. In section 6.2, we discuss how Scala macros can be used to achieve functionality similar to F# type providers [116]. In section 6.3, we continue with how macros influence type-level programming in Scala. Finally, the chapter concludes with the impact of Scala macros on the DSL story, both for internal (section 6.4) and external (section 6.5) domain-specific languages.

Many of the ideas and techniques described below have been born and refined in collaboration with the members of the Scala community: Aggelos Biboudis, Travis Brown, Paul Butcher, Olivier Chafik, Aloïs Cochard, Mathias Doenitz, Dominik Gruntz, Philipp Haller, Li Haoyi, Mark Harrah, Lars Hupel, Vojin Jovanovic, Grzegorz Kossakowski, Evgeny Kotelnikov, Roland Kuhn, Heather Miller, Andrew Markii, Adriaan Moors, Alexander Myltsev, George Leontiev, Alexander Nemish, Simon Ochsenreither, Martin Odersky, Erik Osheim, Dmitry Petrashko, Paul Phillips, Michael Pilquist, Hubert Plociniczak, Aleksandar Prokopec, Johannes Rudolph, Miles Sabin, Tobias Schlatter, Leonard Schneider, Denys Shabalin, Amir Shaikhha, Nick Stanchenko, Nicolas Stucki, Tom Switzer, Eric Torreborre, Vlad Ureche, Bill Venners, Jan Christopher Vogt, Pascal Voitot, Adam Warski, Jason Zaugg, Stefan Zeiger and many others.

6.1 Materialization

Materialization is a technique that uses implicit def macros (chapter 4) to generate arguments for methods that take implicit parameters. Since implicit parameters can be used to encode typeclasses [89], materialization can be used to programmatically derive typeclass instances, enabling datatype-generic programming [84]. Another common use case for implicit parameters is encoding type-level computations in a manner similar to functional dependencies [59], and macros can contribute to this area as well.

Initial experiments with materialization were attempted by Miles Sabin even before the official release of def macros [96], but the technology for that has arrived only with Scala 2.10.2. Among the pioneers of materialization were Miles Sabin [97], Lars Hupel [63], and the initial scala/pickling team consisting of Heather Miller, Philipp Haller and Eugene Burmako [81]. Applications of materialization to data-type generic programming are further described in [83].

6.1.1 Essence of materialization

The example below defines the `Showable` typeclass, 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`.

```
trait Showable[T] {
  def show(x: T): String
}

def show[T](x: T)(implicit ev: Showable[T]) = ev.show(x)
```

After being declared like that, `show` can be called with only the target provided, and the Scala compiler will try to infer the corresponding typeclass 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.

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

One of the well-known problems with typeclasses, 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 rarely possible to implement them concisely, so the programmer is forced to repeat themselves over and over again, like in the listing provided below.

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

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

If we are targeting the JVM, such repetition can usually be abstracted away with runtime reflection (subsection 3.1.4), but runtime reflection oftentimes is either too imprecise because of erasure or too slow because of the overhead it imposes.

With def macros (chapter 4) it becomes possible to completely eliminate the boilerplate without compromising platform-independence, static precision and performance.

```
trait Showable[T] {
  def show(x: T): String
}

object Showable {
  implicit def materialize[T]: Showable[T] = macro ...
}
```

Instead of declaring multiple instances, the programmer defines a single `materialize` macro in the companion object of the `Showable` typeclass.

According to Scala implicit resolution rules, members of companion objects are used for fallback implicit lookups, which means that in cases when the programmer does not provide an explicit instance of `Showable` and when one cannot be found in scope, the materializer will be called. Upon being invoked, the materializer can acquire a representation of `T` as described in subsection 4.3.2 and generate the appropriate instance of the `Showable` typeclass. This represents the essence of materialization.

6.1.2 Integration with vanilla implicits

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 as shown in the listing below.

```
case class Point(x: Int, y: Int)

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

show(List(Point(40, 2))) // prints: List(Point(40, 2))
```

In this case, the resulting typeclass instance for `Showable[List[Point]]` will be composed from an autogenerated instance for `Point` and a call to the manually written `listShowable`. Importantly, since implicits declared in companion objects have the lowest priority of all definitions in the implicit scope, user-defined typeclass instances will always have priority over the default materializer.

Integration with vanilla implicits is typical to materialization libraries. Usually, such libraries feature a low-priority catch-all materializer along with several higher-priority manually defined instances that cover corner cases, i.e. primitives or types special to the domain (lists, other collections, etc).

6.1.3 Tying the knot

An important aspect of materialization is dealing with recursion. Usually, materializers enumerate the fields of the type, for which the typeclass instance is generated, and then recursively process these fields.

```
case class Employee(name: String, company: Company)
case class Company(name: String)

val employee: Employee = { ... }
show(employee)
```

For example, materialization of the `Showable[Employee]` instance that will be produced by `Showable.materialize` may look as follows:

```
show(employee)(
  new Showable[Employee] {
    def show(x: Employee) = (
      "Employee(" + show(x.name) + ", " + show(x.company) + ")"
    )
  }
)
```

Due to the fact that expansions of def macros are immediately typechecked, materialization will cause recursive implicit lookups of `Showable[String]` and `Showable[Company]`. Of those, the former will most probably be defined manually by the library author as explained in subsection 6.1.2, whereas the latter will again be handled by the materialization macro.

Since types in Scala can be mutually recursive, it is important to ensure that the materialization infrastructure is resilient to cycles in the materialization graph both at compile time, i.e. when generating instances, and at runtime, i.e. when executing the generated instances.

In order to prevent infinite recursion at compile time, one needs to "tie the knot", shortcircuiting implicit searches of already generated instances to themselves. Here is one way of approaching this:

```
show(employee)({
  implicit object ShowableEmployee$1 extends Showable[Employee] {
    def show(x: Employee) = (
      "Employee(" + show(x.name) + ", " + show(x.company) + ")"
    )
  }
  ShowableEmployee$1
})
```

In the example above, instead of generating an anonymous instance of the `Showable` trait, the macro emits a block that defines a freshly-named object implementing `Showable` and immediately returns that object. Since the defined object is implicit, it will satisfy potential future implicit searches of `Showable[Employee]`, avoiding triggering `materialize` again and again.

In order to prevent infinite recursion at runtime, materialization infrastructure typically needs to be enriched with a cache that keeps track of already processed objects. For example, in the case of prettyprinting, such cache could detect objects that have already been prettyprinted and emit some shortcut representation for them. Refer to [83] for a more detailed analysis of dealing with cycles in runtime object graphs.

6.1.4 Fundep materialization

Fundep materialization is an extension of materialization (subsection 6.1.1), in which generated typeclass instances are used to express functional dependencies between type parameters, guiding type inference in a domain-specific fashion. Compiler support for fundep materialization was initially added during the development cycle of Scala 2.11.0 and was later backported to Scala 2.10.5.

For a motivating example, let us consider a simplified excerpt from Shapeless, a popular generic programming library for Scala [98], developed by Miles Sabin and contributors. Among its foundational data structures, Shapeless defines HLists, inspired by an eponymous Haskell library [68].

```
sealed trait HList
final case class ::[+H, +T](head: H, tail: T) extends HList {
  def ::[H](head: H) = new ::(head, this)
}
sealed trait HNil extends HList {
  def ::[H](head: H) = new ::(head, this)
}
final case object HNil extends HNil

type HPerson = String :: String :: HNil
val hperson: HPerson = "John" :: "Smith" :: HNil
```

In the listing above, we can see that hlists, much like regular lists, are modelled as a disjunction of a cons and a nil. Unlike regular lists, hlists can contain elements of different types and keep track of these types.

Thanks to their regular structure, hlists lend themselves well to being used in type-level computations. For example, here is how one can define a serialization facility that works with arbitrary hlists.

```
def serialize[T](x: T)(implicit ev: Serializer[T]): Payload =
  ev.serialize(x)

trait Serializer[T] {
  def serialize(x: T): Payload
}

object Serializer {
  implicit val string: Serializer[String] =
    new Serializer[String] { def serialize(x: String) = ... }

  implicit val hnil: Serializer[HNil] =
    new Serializer[HNil] { def serialize(x: HNil) = Payload.empty }

  implicit def hlist[H, T <: HList]
    (implicit shead: Serializer[H],
              stail: Serializer[T]): Serializer[H :: T] =
    new Serializer[H :: T] {
      def serialize(x: H :: T) =
        shead.serialize(x.head) + stail.serialize(x.tail)
    }
}
```

Much like regular lists can be processed using pattern matching, type-level lists represented by hlists can be processed using pattern matching encoded in implicit search rules. For example, if one calls `serialize(hperson)`, implicit search will recursively go through the components of `HPerson` and produce `hlist(string, hlist(string, hnil))`.

Unfortunately for this elegant serialization solution, real-world data in Scala is usually represented with different kind of data structures, e.g. case classes, which do not have the regular type-level structure of hlists.

In order to provide infrastructure to combat this incompatibility, Shapeless provides the `Generic` typeclass that embodies a bidirectional conversion between regular datatypes and hlists. Below we can see its simplified version.

```
trait Generic[T, R] {
  def to(t: T): R
  def from(r: R): T
}

case class Person(first: String, last: String)
type HPerson = String :: String :: HNil

implicit val personGeneric = new Generic[Person, HPerson] {
  def to(t: Person) = t.first :: t.last :: HNil
  def from(r: HPerson) = Person(r.head, r.tail.head)
}
```

Now, if we define instances of `Generic` for all case classes that we want to serialize, we can have a generic serializer that takes care of all those case classes at once. Note that this serializer only uses `Generic.to`, but if we were to implement a deserializer, that one would also need `Generic.from`.

```
implicit def serializer[T, R]
  (implicit generic: Generic[T, R],
            hlistSerializer: Serializer[R]): Serializer[T] = {

    new Serializer[T] {
      def serialize(x: T) = {
        val repr = generic.to(x)
        hlistSerializer.serialize(repr)
      }
    }
}

val person = Person("John", "Smith")
serialize(person)
```

It is important to note how the typechecker handles a call to `serializer` which arises when the typechecker performs implicit search for `serialize(person)`.

Initially, type inference only knows that `T = Person`, because the parent implicit search is looking for `Serializer[Person]`. Afterwards, when the typechecker starts looking up implicit arguments for the call to `serializer`, type inference starts learning additional information about the other, yet uninferred types.

When looking up an implicit value for `generic` of type `Generic[Person, ?]`, where ? signifies an uninferred type, the typechecker finds only one implicit, i.e. `personGeneric`, that satisfies this signature. From the type of this implicit, which is `Generic[Person, HPerson]`, type inference concludes that `R = HPerson`.

Afterwards, when searching for an implicit value for `hlistSerializer`, the typechecker uses the information obtained during the previous round of implicit search, and looks for an implicit of type `Serializer[HPerson]`. As discussed above, this will successfully terminate with `hlist(string, hlist(string, hnil))`.

To sum it up, by strategically defining instances of `Generic` to be non-overlapping, programmers can express the functional dependency from `T` to `R`, and then the typechecker can make use of it during type inference.

Fundep materialization allows programmers to avoid manually setting up implicits that express the functional dependency, allowing to dynamically generate the corresponding typeclass instances. This way, the type-level computation encoded by the functional dependency can be performed programmatically.

```
import scala.language.experimental.macros
import scala.reflect.macros.whitebox.Context

object Generic {
  implicit def materialize[T, R]: Generic[T, R] = {
    macro GenericMacros.materialize[T]
  }
}

class GenericMacros(c: Context) {
  import c.universe._

  def materialize[T](implicit tag: WeakTypeTag[T]): Tree = {
    val T = tag.tpe
    val R = computeRepr(T)
    q"new Generic[$T, $R] { ... }"
  }
}
```

When the typechecker searches for an implicit value of type `Generic[Person, ?]` and is unable to find one in lexical scope, it will look through the implicits defined in companion objects of the fragments of the type, including `Generic`.

As outlined in subsection 4.4.3, when not all type arguments of `Generic.materialize` can be inferred (and in this case, we can only infer `T = Person`, the macro engine will expand the materializer anyway. Inside the corresponding macro impl, we can introspect the list of fields of the inferred type argument and use it to compute its hlist-based analogue.

The fact that the materializer expands into value of type `Generic[Person, HPerson]` allow the type inferencer to proceed further, concluding that `R = HPerson`, which, as explained above, it can use in further type-level computations.

As we can see from the discussion above, fundep materialization provides a framework to encode type-level computations that cannot otherwise be expressed in the Scala type system without requiring significant amounts of boilerplate. This technique has been successfully used for even more complex domains in Shapeless and other libraries in the Scala ecosystem.

6.1.5 Conclusion

Materialization is probably the most distinctive feature of Scala macros. Transparently enriching the mechanism of implicits, which are capable of expressing both term-level and type-level computations, materializers are very versatile.

There are two major use cases for materialization. First, materializers can programmatically derive instances of typeclasses, which enables datatype-generic programming. Secondly, materializers can encapsulate type-level computations expressed in terms of the scala.reflect API and hook them into the Scala compiler.

One downside of materialization is that repeated invocations of materializers on the same type parameter result in repeatedly generated expansions. Modulo the potential presence of scope-specific implicit values that can potentially change the direction of recursive materializations, these expansions will most likely duplicate each other. This peculiarity of materialization can adversely affect compilation performance, resulting code size and runtime performance. Addressing this shortcoming is an important component of our future work.

6.2 Type providers

Type providers [116] are a strongly-typed type-bridging mechanism that enables information-rich programming in F#. A type provider is a compile-time facility that is capable of generating definitions and their implementations based on static parameters describing datasources.

In the example below taken from [116], 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 }
```

An important feature of type providers is that they generate datasource representations lazily, providing types and their members only when explicitly requested by the compiler. This becomes crucial when creating strongly-typed wrappers for entities of the datasource schema is either redundant (from performance and/or reflection standpoints) or infeasible (authors of [116] mention cases where eagerly generated bytecode is too large for the limits of a .NET process).

Moreover, type providers can be configured to have their generated types erased, i.e. replaced by a base class in resulting bytecode. This reduces the size of executable binaries without sacrificing type safety at compile time.

In this section, we describe to what extent Scala macros can be used to emulate the functionality of type providers. Unsurprisingly, the code generation aspect of type providers is easily reproducible with macro annotations (subsection 6.2.1). More interestingly, even def macros can achieve generation of top-level definitions via an unexpected feature interaction (subsection 6.2.2). The most challenging task is to reproduce lazy and erased behavior of type providers (subsection 6.2.3).

6.2.1 Public type providers

Public type providers are called like that because they involve straightforward generation of top-level classes, traits, objects, etc. This is the primary use case for macro annotations (section 5.3), and they support it well. In the listing below, we reproduce the code example from the section about macro annotations (subsection 5.3.2) that defines the `h2db` type provider as a macro annotation.

```
import scala.language.experimental.macros
import scala.reflect.macros.Context
import scala.annotation.StaticAnnotation

class h2db(url: String) extends StaticAnnotation {
  def macroTransform(annottees: Any*): Any =
    macro H2DbMacros.macroTransform
}

object H2DbMacros {
  def macroTransform(c: Context)(annottees: c.Tree*): c.Tree = {
    ...
  }
}

object Test {
  def main(args: Array[String]): Unit = {
    @h2db("coffees") object Db
    val brazilian = Db.Coffees.insert("Brazilian", 99.0)
    Db.Coffees.update(brazilian.copy(price = 100.0))
    println(Db.Coffees.all)
  }
}
```

A typical approach to implementing a public type provider is to write a macro annotation that can be put on an object that will host the generated definitions. Upon becoming annotated, such an object expands according to compile-time logic implemented in the macro annotations. On the example above, such logic connects to a database and generates statically-typed data classes and CRUD helpers for available tables.

6.2.2 Anonymous type providers

Anonymous type providers are macro-based type providers that use anonymous objects and structural types to generate boilerplate and expose it to the outer world. This technique was pioneered by Travis Brown, who noticed a strange interaction between anonymous objects and def macros shortly after macros were officially released [9].

Unlike public type providers (subsection 6.2.1), which use macro annotations and therefore require a third-party compiler plugin, anonymous type providers can be used with the default distribution of Scala since Scala 2.10.

In this approach, boilerplate is generated not as top-level definitions, but as member definitions of local anonymous objects. Concretely, in our running example from subsection 6.2.1, we can replace `@h2db("coffees") object Db` with `val Db = h2db ("coffees")`, where `h2db` is a whitebox macro that expands into an anonymous object.

In order to understand how this works, let us try to write the macro expansion manually in REPL and explore the generated result.

```
scala> val db = new {
     |   case class Coffee(id: Int, name: String, price: Double)
     |   object Coffees {
     |     def insert(name: String, price: Double): Coffee = ???
     |     def update(coffee: Coffee): Unit = ???
     |     def all: List[Coffee] = List(Coffee(1, "Brazilian", 99))
     |   }
     | }
db: AnyRef {
  type Coffee <: Product with Serializable {
    val id: Int;
    val name: String;
    val price: Double;
    def copy(id: Int,name: String,price: Double): this.Coffee;
    def copy$default$1: Int @uncheckedVariance;
    def copy$default$2: String @uncheckedVariance;
    def copy$default$3: Double @uncheckedVariance
  };
  object Coffee
  object Coffees
} = $anon$1@3003a3a3
```

As we can see, the Scala compiler infers a very precise type for the aforementioned anonymous object. All members of the anonymous object get exposed in the resulting structural type. The class representing `Coffee` gets erased to a type member, but nonetheless the type member retains all public members of the original class, even the synthetic `copy$default$N` methods generated to support default parameters in the `copy` method.

The resulting object can be used as if it was written as `object db { ... }`, with all member accesses and method calls statically typechecked. For example, in the printout below, we can see how the structural type statically catches typos, recursively contains structural types for nested objects and even support advanced features like method applications with named and default parameters.

```
scala> db.Coffes
<console>:9: error: value Coffes is not a member of AnyRef { ... }

scala> db.Coffees
warning: there were 1 feature warning(s);
re-run with -feature for details
res0: db.Coffees.type = $anon$1$Coffees$@50325aa1
```

```
scala> res0.all
res1: List[this.Coffee] = List(Coffee(1,Brazilian,99.0))

scala> res1(0).name
res2: String = Brazilian

scala> res1(0).price
res3: Double = 99.0

scala> res1(0).copy(price = 100.0)
res4: db.Coffee = Coffee(1,Brazilian,100.0)
```

An unpleasant downside of this approach is its low-level implementation. On the JVM, structural types are erased to their upper bound (in the case of `db`, that will be `AnyRef`), and method calls on them are performed via runtime reflection, which has suboptimal runtime performance. If we rerun the REPL with `-feature` as prompted by the warning above, we will see the detailed explanation.

```
scala> db.Coffees
<console>:9: warning: reflective access of structural type member
object Coffees should be enabled by making the implicit value
scala.language.reflectiveCalls visible.
This can be achieved by adding the import clause
'import scala.language.reflectiveCalls'
or by setting the compiler option -language:reflectiveCalls.
See the Scala docs for value scala.language.reflectiveCalls
for a discussion why the feature should be explicitly enabled.
              db.Coffees
                 ^
res0: db.Coffees.type = $anon$1$Coffees$@30926bd7
```

Interestingly, this downside can be worked around by even more macros, with the technique that we have called "vampire macros" [8].

If, instead of exposing `Coffee.id`, `Coffee.name`, `Coffee.price` and others as regular vals and defs, we declare them as macros, then the runtime cost of reflective dispatch will no longer be an issue, because accesses to these members will be resolved at compile time during macro expansion.

Of course, this brings another problem. Now, compile-time invocations like `Coffee.name` need to expand into something. A typical approach here is to have classes like `Coffee` wrap a generic property bag, e.g. `Map[String, Any]`, and then have macro expansions access this property bag and downcast the result to the statically-known type. For example, a reference to `coffee.name` could expand into statically-typed `coffee.column("NAME").asInstanceOf[String]`.

This sketch works surprisingly well in practice. Even though structural types do not fully support the entire language (e.g. they cannot refer to certain abstract types and they cannot carry implicit members) and even though def macros have some limitations (section 4.6), it is possible to get quite far with anonymous type providers.

6.2.3 Conclusion

Both def macros (chapter 4) and macro annotations (section 5.3) can generate boilerplate definitions, providing statically-typed wrappers over datasources.

There are trade-offs associated with choosing one approach over another. The approach based on def macros (subsection 6.2.2) is readily available in the standard distribution, but is based on a feature interaction with an exotic feature whose implementation has its corner cases. The approach that relies on macro annotations (subsection 6.2.1) generates regular classes that can be consumed in a straightforward way, but requires a third-party compiler plugin.

Both approaches described above rely on generating dedicated classes for each entity in a datasource schema. While this approach may work well for moderately-sized datasources, [116] reports that there are practical datasources that hit the limits of scalability of such approach. We have not experimented with this yet, but we have a hunch that a combination of the technique used in subsection 6.2.2 with `Dynamic` may very well provide the infrastructure to implement erasure for type providers. This is an interesting element of future work.

6.3 Type-level programming

Type-level programming is a technique that involves writing functions that operate on types and using these functions to achieve precision in type signatures.

While type-level programming has proven to be useful in Scala, being a fundamental feature enabling the design of standard collections [89], 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 [89] 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.

Moreover, 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 strategy to do type-level programming, allowing the programmer to encode type manipulations in macros, written in full-fledged Scala. As we have seen in subsection 6.1.4, fundep materialization provides a practically proven framework that can be used to achieve this.

Type-level computations with macros are more powerful than when written with implicits, because they are not limited by the rules of implicit search and can use the entire surface of the scala.reflect API.

Moreover, 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 that is the epitome of implicit-based programming.

However, type-level computations with implicits sometimes end up more concise than their macro-based analogues. Even though most algorithms are way easier to express with a reflective function, the boilerplate that is required to hook them up into the fundep materialization infrastructure sometimes overweighs the benefits.

For example, let us take a look at the definition of `Generic` from subsection 6.1.4. In order to express a type-level function that goes from case classes to their homogeneous hlist-based representations, we define a dedicated typeclass, an associated materializer and then wrap the result of the type-level function in an auxiliary instance of that typeclass.

```
trait Generic[T, R] {
  def to(t: T): R
  def from(r: R): T
}

object Generic {
  implicit def materialize[T, R]: Generic[T, R] = {
    macro GenericMacros.materialize[T]
  }
}

class GenericMacros(c: Context) {
  import c.universe._

  def materialize[T](implicit tag: WeakTypeTag[T]): Tree = {
    val T = tag.tpe
    val R = computeRepr(T)
    q"new Generic[$T, $R] { ... }"
  }
}
```

If our only goal is to express a type-level computation, that is a lot of boilerplate, since all that we need for that goal is to call `computeRepr(T)`. It would definitely be less ceremony if it was possible to a macro that uses an imaginary syntax similar to what was possible with type macros (section 5.2).

```
type Generic[T] = macro Macros.generic[T]

class Macros(c: Context) {
  import c.universe._

  def generic[T](implicit tag: WeakTypeTag[T]): Tree = {
    val T = tag.tpe
    computeRepr(T)
  }
}
```

The syntax in the listing above inherits the limitations of the initial design of Scala macros, so it is still more verbose than it could be. However, in the new macro system that merges macros and their impls, this snippet would look remarkably concise (see an example of a similar type-level function at the end of subsection 9.3.5).

Unfortunately, by reducing materialization to type-level computation, we have lost the ability to perform type-driven synthesis of terms. Concretely, our serializer from subsection 6.1.4 that was using `Generic.materialize`, also needed its synthesized `to` method, which is no longer available after we turn `Generic` into a type function.

```
implicit def serializer[T, R]
  (implicit generic: Generic[T, R],
            hlistSerializer: Serializer[R]): Serializer[T] = {

    new Serializer[T] {
      def serialize(x: T) = {
        val repr = generic.to(x)
        hlistSerializer.serialize(repr)
      }
    }
}
```

To put it in a nutshell, macro-based approach to type-level computations is more straightforward and more powerful than its implicit-based counterparts, but sometimes it is also significantly more verbose. It seems that the balance between simplicity of macros and declarativeness of implicits has yet to be found.

6.4 Internal DSLs

Macros have been successfully used to build powerful internal domain-specific languages. Thanks to the fact that the code wrapped in a macro application is available as an AST, DSL authors can inspect and rewrite such code according to domain-specific logic.

For example, we have seen how def macros (chapter 4) can provide a foundation to build a LINQ facility that translates calls to an internal data access DSL that mimics standard collection operators into domain-specific query constructors.

```
trait Query[T]
case class Table[T: TypeTag]() extends Query[T]
case class Select[T, U](q: Query[T], fn: Node[U]) extends Query[U]

trait Node[T]
case class Ref[T](name: String) extends Node[T]

object Query {
  implicit class QueryApi[T](q: Query[T]) {
    def map[U](fn: T => U): Query[U] = macro ...
  }
}

case class User(name: String)
val users = Table[User]()
users.map(u => u.name)
// translated to: Select(users, Ref[String]("name"))
```

This is a bread-and-butter technique available to def macros, and it has been successfully used to develop numerous DSLs. Nevertheless, despite of the simplicity of this technique, there are several practical concerns that metaprogrammers have to keep in mind when using it. Below we outline the most common ones, and refer curious readers to Vojin Jovanovic's dissertation [66] for an extensive analysis of internal DSLs in Scala.

Transformation scope. Because of its local nature (section 4.4), macro expansion can see tree representation only for the code that is currently expanding. As a result, if a macro needs to introspect or transform code that lies outside the expansion scope, this may cause problems.

```
def nameMapper(user: User) = u.name
users.map(nameMapper)
```

If we take a look at the code of the `Query.map` macro provided in section 4.2, we will see that expanding the snippet provided above will result in a compilation error. This happens because the translation macro only sees `nameMapper` and does not know anything about its body.

This transformation failure is quite hard to fix in the current implementation of def macros, because there is no API to obtain a tree of an arbitrary method. Even though there is an undocumented way to get ahold of ASTs of all files being compiled, the method may be already precompiled, in which cast its AST will be missing. In scala.meta, we are experimenting with mandatory AST persistence for compiled code that would solve this problem, but this technology has not yet reached sufficient maturity (section 8.4).

To work around this problem, metaprogrammers have been using various techniques to extend the scope of code available to macros. One example is wrapping DSL snippets in an explicit macro block, e.g. requiring users to write `query { ... }`, where `query` is a macro, instead of declaring methods like `Query.map` as macros and hoping that their invocations capture all relevant context.

Error reporting. Some DSLs cannot handle the entire surface of the language. For example, our running example of a LINQ DSL can only support operations that map onto the capabilities of the underlying database. Even though some simple operations like `String.substring` have adequate analogues in SQL, more complicated methods from the standard library are unlikely to be supported.

Type-based DSLs solve this problem by providing a carefully crafted interface that only exposes the supported parts of the language and the standard library. For example, a database access DSL from Slick [75] uses `Rep` types that represent database types and only support allowed operations.

```
class Users(tag: Tag) extends Table[(String, String)](tag, "USR") {
  def name = column[String]("NAME")
  def role = column[String]("ROLE")
  def * = (name, role)
}

val users = TableQuery[Users]
users.filter(u => u.role === "admin").map(u => u.name)
```

In the listing above, all plain types are wrapped, which allows the DSL to capture the query in a fashion similar to symbolic execution.

For example, the result of the `role` method used in `filter` is not `String`, but `Rep[String]`. By the virtue of extension methods, `Rep[String]` defines operations like `===` or `indexOf`, so they can be used in Slick queries. To the contrast, `Rep[String]` does not feature analogues of the standard `String.getBytes` or `String.matches`, so these methods can not be used.

Defining such an interface is a tedious and error-prone task, but for type-based DSLs it is a necessity, because otherwise DSL authors will not be able to access representations of the programs that users write using these domain-specific languages.

To the contrast, macro-based DSLs get program representations for free, massively simplifying the task of creating a language.

However, what macro-based DSLs lack is an out-of-the-box mechanism to restrict user programs to supported shapes. If, in the Slick query, we change `u => u.name` to `u => u.name.getBytes`, we will get a compilation error. Unfortunately, the `Query.map` macro will accept such a lambda expression, because the macro engine only checks that macro arguments are well-typed, and the task of rejecting it will fall on the metaprogrammer.

From the discussion above, we can see that macros provide a powerful foundation for building internal DSLs, but this foundation can benefit from higher-level abstractions that take care of common practicalities of authoring domain-specific languages.

6.5 External DSLs

External domain-specific languages are relevant even in languages like Scala which were designed to be friendly to internal DSLs. Regular expressions, XML, JSON, 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 plain strings, which can be parsed and interpreted, but cannot communicate with the main program. Compile-time metaprogramming provides a way to add depth to external DSLs, making them able to analyze and possibly influence the enclosing program.

In Scala, external DSLs can be embedded into code by the virtue of string interpolation [114], 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 `sc.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. If we turn `json` in the example above into an object that contains `apply` and `unapply` methods, we will be able to use the `json` interpolator both in expressions and patterns.

Turning interpolators into def macros (chapter 4) opens a number of possibilities to DSL authors. 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}"
```

Analogously, it is possible to apply this strategy to interpolations used in pattern matching. For example, a pattern macro (section 5.1) can figure out exact types for patterns in `payload match { case json"{$name: $value}" => ... }` and ensure that domain-specific pattern matching is precisely typed.

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 [104] 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.

The most comprehensive overview of building external DSLs with macros is provided by Denys Shabalin in his `joyquote` tutorial [102] that embeds a subset of the Joy programming language into Scala. The tutorial develops the notions from this section into a practical system that supports unquoting, splicing, lifting and unlifting, as well as discusses interactions between different features of macro-based external DSLs.

To put it in a nutshell, the feature interaction between def macros and string interpolation makes it possible to embed external DSLs in Scala, providing compile-time validation and typechecker integration for snippets of code written in other languages. A peculiar property of such DSLs is that they can be used both in expressions and in patterns.

7 Tool support

Scala macros (chapter 4 and chapter 5) provide powerful abstraction facilities that strain the developer tool ecosystem in several unique ways.

First, macros generate code that is invisible to their users. Since subsequent compilation phases of the compiler do not see the original macro applications and work with macro expansions instead, understanding compile-time and runtime behavior of the original program may become challenging.

Secondly, macros change the type system. Blackbox macros (section 4.4) are relatively benign, because for the purposes of code analysis they can be treated as regular methods (modulo compilation errors produced by ill-typed expansions or macro APIs). However, whitebox macros (section 4.4) and other macro flavors (chapter 5) result in immediately visible changes to how Scala typechecks, making tools that do not support macros noticeably less useful.

This chapter documents tool integration issues that metaprogrammers face when writing Scala macros and explains our approaches to overcoming these issues. Most of these approaches are only available as prototypes that represent ongoing work.

In section 7.1, we outline ways how macros change the usual intuitions about code comprehension. In section 7.2, we further develop this notion by looking at new kinds of compilation errors that can be caused by macros. Changes to the traditional development workflow are described in section 7.3 (incremental compilation), section 7.4 (testing) and section 7.5 (debugging). Finally, in section 7.6, we analyze how macros affect documentation.

Most of the work described below was performed by open-source contributors under my supervision - both by my students at EPFL and by external collaborators from the community. I am thankful to everyone for our fruitful work together.

7.1 Code comprehension

Common to whitebox macros (section 4.4) and macro annotations (section 5.3) is the fact that their signatures cannot be expressed in the Scala type system.

When whitebox macros change implicit search or type inference, or when macro annotations generate top-level definitions, IDEs need to interactively expand these macros, otherwise they will have incorrect information about the program and may display infamous red squiggles.

There are three major IDEs in the Scala community: IntelliJ [65], Eclipse [99] and ENSIME [44]. Both Eclipse and ENSIME run a full-fledged Scala compiler that can expand macros. However, IntelliJ has its own independent frontend, so it needs an adapter that would expose its internal functionality behind a scala.reflect API. Unfortunately, such an adapter was deemed to be impractical because of the complexity of scala.reflect and has never been implemented (subsection 3.4.3). Therefore, at the time of writing, IntelliJ has problems with whitebox macros and macro annotations.

In order to make it possible for IntelliJ to expand macros, we designed scala.meta, a better metaprogramming API (chapter 8), and created a new macro system based on scala.meta (chapter 9). Enabling scala.meta metaprograms to run in IntelliJ is an area of ongoing collaboration with Mikhail Mutcianko from the IntelliJ team (section 8.4).

In the meanwhile, IntelliJ employs two practical workarounds - one that involves communicating with the Scala compiler to obtain macro expansions, and another one that involves writing IDE plugins.

The first workaround consists in continuously running background compilation with internal compiler flags that print macro expansions to console. These expansions are then parsed and integrated into the language model of IntelliJ. While this approach works as a last-ditch effort, it hurts interactivity and may stop working if the underlying program has compilation errors.

The second workaround consists in exposing an API to write IntelliJ plugins that capture the essence of particular macros [21]. In practice, most of the complexity of typical macros lies in performing robust code generation, so the code that is responsible for type system changes constitutes a small part of the macro logic. This means that extracting the essence of popular macros, reimplementing it on top of the infrastructure provided by the IDE and keeping it up to date is quite practical.

7.2 Error reporting

When a macro expansion generates erroneous code, it often happens that compilation errors produced by the compiler make little sense. Since these errors are reported against generated code that is normally invisible to programmers, the results can be quite perplexing. An illustration of this problem can be observed in section 4.2.

To help macro users with this problem, we have implemented several low-cost workarounds and are collaborating with IDE developers to deliver a principled solution. In this section, we document all approaches that our macro system supports.

Expansion printouts. This workaround involves internal compiler flags - `-Ymacro-debug-lite` and `-Ymacro-debug-verbose`, available in Scala since v2.10 - which make the compiler print macro expansions to console. Such printouts rarely help with pinpointing an exact culprit, because macro expansions are typically quite sizeable, but they at least give something to work with.

One practical aspect of this workaround is that it highly benefits from a helpful prettyprinter. Unfortunately, scala.reflect trees involved in macro expansions lack formatting and are typically desugared to a significant degree (section 3.2), so the prettyprinted result may be cryptic even to experienced metaprogrammers. Since the potential consumers of such printouts are macro users, not macro developers, this is especially concerning. We expect that new-style macros based on scala.meta will improve the situation (chapter 9).

Explicit positions. This workaround enlists the help of macro writers. Scala.reflect provides a way to annotate generated trees with positional information (subsection 3.2.2). In the case when a culprit of a typecheck error is a positioned AST, the error message emitted by the compiler will point at the corresponding position.

This technique works well when all that a macro does is carefully enriching existing code with small bits of additional logic. However, it does not help much if a macro generates a lot of synthetic code that cannot be assigned with meaningful positions.

Interactive expansion. Our most promising solution to the problem of error messages is interactive expansion. The idea is to integrate into IDEs and provide user interface for step-by-step macro expansion. This way macro users can see exactly what their macro applications are expanding into (section 7.1), and within these expansions they can pinpoint erroneous trees using the standard on-the-fly typechecking facilities.

Thanks to the efforts of Mikhail Mutcianko, Dmitry Naydanov, Artem Nikiforov, Igor Bogomolov, and Jatin Puri, we have explored two UIs for interactive expansion - in-place expansion via the code folding metaphor and side-by-side expansion via the playground metaphor. Ultimately, both UIs have proven to have their merits.

7.3 Incremental compilation

Sbt [74], the most popular Scala build tool, supports *incremental compilation*. When building Scala programs, sbt attempts to reuse the results of previous builds as much as possible, by only compiling the source files that have been changed since the last build and dependencies thereof.

In order to achieve incremental compilation, sbt runs the Scala compiler in a special mode where the standard compilation pipeline is extended by several additional phases which collect and record dependencies between files. When a programmer makes a change to their program, sbt uses the computed dependencies to understand which files in the project depend on the changed files and therefore need to be recompiled along with them.

When macros were introduced, it became apparent that the existing dependency detection algorithm in sbt has become insufficient. As part of his master studies, Martin Duhem has implemented solutions to most of the problems that we found, and many of his fixes were integrated into production versions of sbt. Below we provide a brief overview of Martin's contributions, leaving the full story to [37] and [38].

Tracking dependencies of macro applications. Sbt dependency collection phase runs after typechecking, which means that earlier versions of sbt did not get to see the original macro applications, just their expansions. As a result, sbt failed to recompile macro expansions if dependencies of macro arguments changed. The fix was fairly simple. Every macro expansion now carries its original, so that sbt could take both into account when computing dependencies.

Tracking dependencies created by introspection. Macro impls have access to powerful introspection capabilities via their `Context` parameter (section 4.5), and almost every act of reflection creates a dependency on the reflected code element. This is completely opaque to sbt, because it only samples the compilation pipeline before and after the expansion.

In order to record information about language elements being introspected, we created a proxy for `Context` that intercepts APIs that return symbols and then proxies those symbols to record situations when someone inspects them (their names, signatures, etc). This did not track all dependencies, because we only recorded accesses that happened inside macro impls, but not accesses performed internally in the compiler, but that fixed a significant chunk of incremental compilation issues.

Tracking dependencies on external files. Macro impls can feature arbitrary logic, including file I/O. Some metaprogrammers use this possibility to generate code from external files, only to discover that sbt does not recompile their code that uses the generated definitions if these external files change.

Upon a feature request, we implemented a private API that expresses that a certain macro expansion depends on a certain group of files. A more principled approach would be to hide file I/O behind a macro API and automatically track calls to such an API, but that remains future work that we may address in the new macro system (chapter 9).

Tracking dependencies of macro impls. Any change in a macro impl (section 4.3) or its transitive call graph should lead to recompilation of macro expansions. Unfortunately, sbt dependency tracking infrastructure is way too simplistic to support this requirement. Now, there is a prototype that upgrades sbt to accommodate runtime dependencies of macro impls, but it remains to be seen whether this prototype can be turned into a production-ready solution.

7.4 Testing

At the time of writing, it is only possible to write integration tests for macro expansions. By instantiating the Scala compiler infrastructure at runtime and passing it snippets of code that contain macro applications, it is possible to expand macros in a isolated environment and inspect expansion results, i.e. to look into generated code, compilation errors, etc.

Unit tests for pieces of logic underlying macro expansions do not work well at the moment. Mocking the scala.reflect API requires implementing several hundred APIs (section 3.4), which is a very time-consuming task. We are not aware of such implementations existing in the Scala ecosystem.

7.5 Debugging

There are two kinds of debugging that involve macros. The first is debugging macro impls. This one is conceptually simple, because macros are written as regular Scala code, so any Scala debugger will work for this task.

The second is debugging generated code, and that is a real challenge that we will be discussing in this section based on the results obtained by Dmitry Naydanov, Mikhail Mutcianko, Artem Nikiforov, Igor Bogomolov, Zhivka Gucevska, Jatin Puri and Uladzimir Abramchuk. Some of these results are documented in [87].

Much like with error reporting, there are workarounds for debugging generated code (e.g. inserting debug prints into generated code), but a full-fledged solution required dedicated effort. A lot of Scala programmers debug their applications using breakpoints, debug watches and other functionality provided by IDEs, and supporting that workflow requires coordination between the macro engine and the IDE.

Firsly, the IDE needs to know the results of macro expansion, because one can not set a breakpoint on what one can not see. Because code changes require rerunning the compiler before debugging can proceed, in this use case there is no pressing demand for interactive expansion (section 7.1), however there still must be a way to communicate macro expansions from the Scala compiler to the IDE.

Secondly, the Scala compiler needs to emit debug information that takes macro expansions into account. The JVM provides a way to establish correspondence between bytecodes and lines in source files. However, this support is not enough for macro expansions, which may transform a single line of code into a sizeable macro expansion that spans many lines when prettyprinted. Therefore, there must be a mechanism which allows the debugger to find out which parts of executable code correspond to which parts of prettyprinted expansions.

One approach is to solve the debugging problem is pretend that files with macro expansions never had macro applications in the first place, and that their corresponding expansions were written there manually. This involves changing the Scala compiler to internally prettyprint expansions, then assign positions to expanded trees according to the results of prettyprinting, and finally shift positions of original trees that do not participate in macro expansion.

In this case, the debugger obtains macro expansions (either by expanding relevant macros internally or by loading expansions saved by the Scala compiler), inserts them into the UI and then uses the standard functionality provided by the JVM to perform stepping, breaking, etc. This has the benefit of a relatively simple implementation, but also involves the massive downside of breaking existing tools, namely macro-unaware debuggers and, more importantly, stack traces.

Another approach is to pretend that macro expansions are written past the end of the file. This is similar to the previous strategy, with the difference that the positions of manually written code remain as they were, which is more friendly to existing tools.

In this case, the job of the debugger becomes more complicated, because it needs to correlate the positions of macro expansions as reported in debug information with positions of macro expansions in the UI. Additionally, this strategy still does not fix stack traces and macro-unaware tools completely, because some stack trace elements may point to generated code, which will have strange line numbers.

Having experimented with multiple approaches to debugging generated code, we now think that the only way to provide smooth user experience is with a fully-customized stack of tools.

Instead of trying to fit macro expansions into the standard debugging infrastructure provided by the JVM, we propose to save relevant information in a custom class file section.

Then, a dedicated debugger will read and respect this information while rendering its UI. Despite the major effort required to adapt existing debuggers to the new infrastructure, this approach has the benefit of avoiding surprises when working with macro-unaware tools, which we believe to be essential for macro users. We are looking forward to experimenting with this approach in the future.

7.6 Documentation

Macro annotations (section 5.3) can generate top-level definitions, therefore for optimal usability they need an API to provide documentation for generated code.

Unfortunately, scala.reflect does not provide facilities to manipulate syntactic trivia (subsection 3.2.2), e.g. to attach documentation comments to generated trees, which means that there is no public way of supporting this use case. We are planning to improve on this with new-style macro annotations, because their underlying API natively supports comments (section 8.2).

However, as it often happens with scala.reflect, there is a way to tap into compiler internals and wrap generated definitions into internal `DocDef` AST nodes that represent definitions with attached documentation. This is yet another situation when effective use of Scala macros requires knowledge of compiler internals.

7.7 Conclusion

Disregarding tool support during the initial design of Scala macros was a mistake. As our practice shows, tool support has profound impact on user experience of compile-time metaprogramming and is not something that can be easily added as an afterthought.

Most of the problems with tool support are caused by scala.reflect (chapter 3), the metaprogramming API that underlies Scala macros. The fact that scala.reflect is based on the internals of the Scala compiler: 1) prevents third-parties from implementing the macro engine, 2) complicates instrumentation of the API for testing and dependency tracking, 3) causes problems with prettyprinting of expansions, which is key to a surprising amount of tool-related scenarios.

In order to implement comprehensive tool support for macros, we developed scala.meta (chapter 8), a new metaprogramming framework that solves all aforementioned problems of scala.reflect. Based on scala.meta, we designed a new macro system (chapter 9) that lends itself well to third-party implementations. Our initial experiments have produced promising results (section 9.6), so we are confident that scala.meta provides a solid foundation for solving the tool support problem.

Part III Beyond unification

8 Scala.meta

Scala.meta is a unified metaprogramming framework created from scratch to be easy to use for humans and easy to implement for developer tools. After testing its ideas on a series of prototypes, we are now working on an implementation that we hope will become the new standard for metaprogramming in Scala. The current milestone release is available as a standalone library for Scala 2.11 [15].

The main innovation of scala.meta is a language model that can serve as a lingua franca understood by the Scala compiler and third-party tools (section 8.2). Metaprograms written against scala.meta can run both at compile time and at runtime (section 8.4).

The most important use case for scala.meta is macros (chapter 4 and chapter 5). The original motivation for creating scala.meta was the need in a metaprogramming API that would provide better user experience for macros, and scala.meta delivered on that, becoming the foundation for a next-generation macro system (chapter 9).

Scala.meta was and still is strongly influenced by scala.reflect (chapter 3). On the one hand, we view scala.reflect as the source of inspiration for things that did and did not work in an API, and that very much guides our design process. On the other hand, we view scala.reflect as a tool that programmers use to get things done. Since we are planning to succeed scala.reflect, we will have to inherit and support its users, which means that we need to be mindful of its important use cases. In the majority of situations, our prototypes achieve feature parity or succeed the capabilities of scala.reflect.

Scala.meta was born from my collaboration with Denys Shabalin. We coauthored the initial design and implementation, and even now, when Denys has shifted his focus to other avenues, we frequently discuss design problems and further evolution of the project. Fragments of design and implementation of scala.meta were developed together with Mikhail Mutcianko, Oleksandr Olgashko, Mathieu Demarne, Adrien Ghosn, Eric Béguet, Johann Egger, Ólafur Páll Geirsson, Guillaume Massé, Vojin Jovanovic, Guillaume Martres and other open-source contributors.

8.1 Intuition

We will organize this section as a comparison with section 3.1, where we started with an architectural overview, then introduced the `isImmutable` metaprogram and explained how it runs in different environments supported by scala.reflect.

In this section, we will briefly go through the architecture of scala.meta (subsection 8.1.1) and then see how to write `isImmutable` in terms of scala.meta (subsection 8.1.2), discussing how to run `isImmutable` at compile time and at runtime (subsection 8.1.3) and how the outlined approaches differ from what we have already been able to achieve with scala.reflect.

8.1.1 High-level architecture

Informed about the major usability downsides of using cake pattern to expose a public API (subsection 3.1.1), scala.meta implements its language model in a set of top-level definitions. All that it takes to use the language model is `import scala.meta._`.

Much like the decision to use cake pattern in scala.reflect, the decision to use top-level definitions in scala.meta also has far-reaching consequences. In addition to bringing a lightweight feel of not forcing users into unconventional idioms (subsection 3.1.5), it has a major impact on how scala.meta APIs are organized.

In scala.reflect, universes encompass both the definitions of the language model and the pieces of state that are required for its operations. For example, when a metaprogram asks a definition about its signature or a type about the list of its members, the enclosing universe consults its symbol table that internally lives in the universe. This happens unbeknownst to the users, because scala.reflect lives within a cake that hides this detail in its internal state.

In scala.meta, we have to be explicit about state, because the cake is gone. In order to accommodate this design requirement, we went through the operations supported by scala.reflect (section 3.5) and split them into groups based on the kind of state these operations work with. As a result, we ended up with three groups of APIs that comprehensively cover the functionality exposed in scala.reflect.

1) Stateless APIs such as manual construction and deconstruction of reflection artifacts. Unlike in scala.reflect, the language model of scala.meta is stateless, so it can be used in arbitrary situations, regardless of whether it is compile time, runtime or any other environment.

```
scala> import scala.meta._
import scala.meta._
```

```
scala> Term.Name("x")
res0: Term.Name = x

scala> val Term.Name(x) = res0
x: String = x
```

2) Syntactic APIs such as parsing, quasiquotes and prettyprinting. These APIs can change behavior depending on a particular version of the language, so we reified these distinctions into a dedicated entity called `Dialect`, and require a dialect in all such APIs. Below we can see a simplified excerpt from scala.meta that illustrates this design.

```
1  package meta {
2    trait Dialect {
3      private[meta] def allowXmlLiterals: Boolean
4      ...
5    }
6
7    package object dialects {
8      implicit object Scala211 extends Dialect { ... }
9      implicit object Dotty extends Dialect { ... }
10     ...
11   }
12
13   ...
14 }
15
16 package object meta {
17   implicit class XtensionParse[T](inputLike: T) {
18     def parse[U](
19       implicit convert: Convert[T, Input],
20       parse: Parse[U],
21       dialect: Dialect): Parsed[U] =
22     {
23       val input = convert.apply(inputLike)
24       parse.apply(input, dialect)
25     }
26   }
27
28   ...
29 }
```

Here, a dialect is an opaque entity that does not have public methods, encapsulating differences between language versions in methods that are only visible to scala.meta. For example, `Dialect.allowXmlLiterals` (line 3) indicates whether a particular language version supports XML literals. Current versions of the compiler have this feature, but future versions based on Dotty are going to drop support for it.

Syntactic operations like `Input.parse` take an implicit dialect and use its internal methods to implement their logic. This particular operation is just a simple proxy that converts its argument to parser input and then feeds the input along with the dialect into a parser encapsulated in `Parse`, but the parser itself makes full use of syntax peculiarities expressed by the dialect.

In order to use a syntactic API, we import an implicit dialect (note the `implicit` modifiers next to implementors of `Dialect` on lines 9-10). After an implicit dialect is available in scope, calls to syntactic operations will automatically use it.

```
scala> import scala.meta.dialects.Scala211
import scala.meta.dialects.Scala211

scala> "<xml />".parse[Term]
res0: Parsed[scala.meta.Term] = <xml />
```

In order to improve user experience, current version of scala.meta features a fallback dialect that is used if no dialect was explicitly imported by the metaprogrammer. This default dialect captures to the version of the compiler that compiled a given call to a syntactic API. Therefore, even if in the listing above we did not import `Scala211`, the call to `parse` would still work, and its result would correspond to the behavior of the particular version of the compiler that underlies the REPL.

3) Semantic APIs such as name resolution, typechecking, enumeration of members of a given type, etc. These operations need an index that keeps track of definitions available in the program and its dependencies. We encapsulated such an index in a dedicated trait called `Mirror` and require a mirror in all semantic APIs. To illustrate this point, here is a simplified excerpt from an experimental branch of scala.meta that demonstrates the definition of `Mirror` and one of the associated semantic operations.

```
1  package meta {
2    trait Mirror {
3      private[meta] def dialect: Dialect
4      private[meta] def defn(ref: Ref): Member
5      ...
6    }
7
8    object Mirror {
9      implicit def mirrorToDialect(mirror: Mirror): Dialect = {
10       mirror.dialect
11     }
12   }
13
14   ...
15 }
16
```

```
17 package object meta {
18   implicit class XtensionTypeRef(tree: Type.Ref) {
19     def defn(implicit m: Mirror): Member = {
20       m.defn(tree)
21     }
22   }
23
24   ...
25 }
```

Much like a dialect, a mirror is also an opaque trait with all its logic concentrated in internal methods. Analogously to syntactic operations, semantic operations take an implicit mirror. Additionally, since a mirror must be aware of its language version, it has a dialect and can be converted to a dialect enabling syntactic APIs (lines 10-12).

Here is an example that creates a mirror from a JVM environment that contains the standard library and then transparently uses this mirror to resolve the identifier `List`, obtaining a scala.meta representation of its definition. In this example, we use quasiquotes (section 8.3), a convenient notation for abstract syntax trees.

```
1 scala> implicit val m = Mirror(".../scala-library-2.11.8.jar")
2 m: Mirror = ...
3
4 scala> q"List".defn
5 res3: Member.Term =
6 object List extends SeqFactory[List] with Serializable { ... }
```

This design of metaprogramming operations requires programs written using scala.meta to explicitly request capabilities that describe their needs. For example, a formatter will probably be okay with a dialect, whereas a linter will likely need a mirror.

Replacing universes with capabilities has been a significant improvement of user experience. First, some metaprograms do not need explicit capabilities, which means that both they and their usages are going to be more concise than in scala.reflect. Secondly, in scala.reflect both writers and callers of metaprograms have to worry about universes, whereas in scala.meta capabilities are typically implicit, so they can be passed around automatically. Finally, capabilities present a much smaller cognitive load, only requiring their users to understand implicits, in contrast to universes that make use of advanced aspects of path-dependent types (subsection 3.1.5).

The only case when capabilities cause issues are universal methods `toString`, `hashCode` and `equals` that are inherited from `Any`, the top type of the Scala type system. These methods have hardcoded signatures that can not accommodate additional implicit parameters, which presents a serious design problem.

Despite the difficulties, we have been able to ensure sensible behavior for stringification. On the one hand, we provide a dedicated API for customizable prettyprinting, which does take a dialect that can be explicitly specified by metaprogrammers. On the other hand, for every object whose prettyprinting depends on a dialect, we remember if a particular dialect was used during its creation (e.g. a dialect to parse code into a tree) and then use it when `toString` is called.

The problem with hashing and equality is more challenging. Maps and sets in both Scala and Java standard libraries use `hashCode` and `equals`, and that makes them unusable for abstract syntax trees whose equality relies on name resolution. Since, unlike with dialects, there is no mirror that could work as a sensible default, we currently give up and require metaprogrammers to use customized collections.

To put it in a nutshell, the architecture of scala.meta provides tangible benefits over scala.reflect. From the point of view of plumbing, metaprograms written with scala.meta are more concise and easier to understand, because the infrastructure makes use of less advanced language mechanisms. The only significant issue that we observed is the necessity for custom maps and sets to accommodate custom hashing and equality for abstract syntax trees.

8.1.2 Revisiting isImmutable

In the listing below, we can see a scala.meta-based implementation of the immutability check of Scala types described in subsection 3.1.2. In order to figure out the differences between scala.reflect and scala.meta, let us compare and contrast it with a scala.reflect implementation provided in the aforementioned subsection.

```
1  import scala.meta._
2
3  def isImmutable(t: Type)(implicit m: Mirror): Boolean = {
4    val cache =
5      scala.collection.mutable.Map[Type, Boolean]().
6      withEquality(_ =:= _)
7
8    def uncached(t: Type): Boolean = {
9      t match {
10       case t"$t ..@$annots" =>
11         cached(t)
12       case t"$t forSome { ..$defns }" =>
13         cached(t)
14       case t"..$parents { ..$defns }" =>
15         parents.exists(cached)
16       case t"$_.type" | t"${_: Lit}" =>
17         cached(t.widen)
```

```
18        case t"${ref: Type.Ref}[...$_]" =>
19          if (ref.defn.isFinal && (ref.defn.tpe =/= t"Array")) {
20            if (t.vars.nonEmpty) return false
21            val fieldTypes = (t.vals ++ t.objects).map(m => m.tpe)
22            fieldTypes.forall(cached)
23          } else {
24            false
25          }
26        case _ =>
27          sys.error("unsupported type: " + t)
28      }
29    }
30
31    def cached(t: Type) = {
32      cache.getOrElseUpdate(t, { cache(t) = true; uncached(t) })
33    }
34
35    cached(t)
36  }
```

As explained in subsection 8.1.1, it is enough to simply `import scala.meta._` on line 1 in order to get access to the entire scala.meta API. In our personal experience of using scala.meta after years of writing scala.reflect metaprograms, this relatively minor aspect is particularly refreshing.

With the cake gone, users of scala.meta have to explicitly request capabilities depending on the needs of their metaprograms. `isImmutable` actively uses semantic APIs, e.g. name resolution on line 19 and member enumeration on lines 20-21, so it requests a mirror on line 3.

Lines 4-6 accommodate the fact that semantic equality that is necessary to correctly compare types does not work out of the box with maps from the standard library. The `withEquality` method is a custom helper that makes maps respect custom equality schemes. Implementation of this helper is omitted for brevity.

The rest of the code on lines 8-35 is similar to the scala.reflect implementation. On a very high level, it also uses memoization on lines 31-33 to avoid infinite recursion, and its main logic on lines 8-29 also goes through different flavors of types, checking them for immutability. This is all part of the problem statement, so it is not a surprise that nothing changes on the high level. Let us focus on the details of language models of scala.reflect and scala.meta.

Scala.reflect uses about a dozen different entities to represent Scala programs. The most important data structures are trees, symbols and types that we have seen in action earlier, but there is a long tail of highly-specialized data structures like modifiers and flags that

often overlap with the main ones. We refer curious readers to chapter 3 for more details about how the language model of scala.reflect works.

To the contrast, scala.meta represents Scala programs with: tokens that describe low-level syntactic details, and abstract syntax trees that describe everything else. This design is remarkably minimalistic and is explained in more detail in section 8.2.

Therefore, places in the scala.reflect version of `isImmutable` that required symbols and types now consistently use abstract syntax trees. As a result, on lines 10, 12, 14 and others, we are able to take apart the given type using a WYSIWYG notation provided by quasiquotes. Moreover, instead of working with scala.reflect symbols, which represent an approximation of definitions with their own dedicated set of operations, we take the given type and, on line 19, go directly to the definition that it refers to using the `Ref.defn` operation.

Other differences between the two implementations are not so major, and are mostly the consequence of scala.meta being optimized for ease of use. For example, one can notice that instead of going through `t.members` like in scala.reflect, we use more specific helpers `t.vars`, `t.vals` and `t.objects` on lines 20-21. Also, we no longer have to process type members and type parameters separately from classes, traits and objects, because scala.meta takes care of quirks like that.

From the discussion above, we can see that scala.meta is simpler than scala.reflect - both from the point of view of high-level architecture and from the point of view of the language model. While these simplifications can not directly influence the inherent complexity of metaprogramming Scala, they lower the barriers to entry and make metaprograms more robust.

8.1.3 Executing isImmutable

Compile-time execution. The original motivation for scala.meta was the need for a metaprogramming API that would power a better macro system. Soon after the official release of scala.reflect macros, we realized that scala.reflect causes a multitude of usability problems including high barrier to entry and subpar tool support (section 3.6) and decided to replace it with something better. Therefore, compile-time execution is the primary use case for scala.meta.

Below we can see the `Immutable.materialize` macro from subsection 3.1.3 rewritten in the new macro system that uses scala.meta. The details of why this macro is useful were explained earlier in this section, and here we will only cover the basics of how new macros work and how they use scala.meta. We invite readers to chapter 9 for more information about the new macro system.

```
1  import scala.meta._
2
3  trait Immutable[T]
4
5  object Immutable {
6    inline implicit def materialize[T]: Immutable[T] = meta {
7      if (isImmutable(T)) q"null"
8      else abort(T.origin, T + " is not immutable")
9    }
10 }
```

From the listing, it becomes clear that the new macro system based on scala.meta is much more concise than the current one based on scala.reflect. Instead of separating signatures and implementations of macros like in subsection 3.1.3, we define macros in one go thanks to meta expressions (lines 5-8).

Cornerstone to new macros is the interaction between the newly introduced mechanisms of `inline` and `meta` that capture the essence of macro expansion. The `inline` modifier on a method indicates to the compiler that applications of that method are to be replaced with the method body having formal parameters substituted with real arguments. `meta` expressions wrap metaprograms written against scala.meta.

Having encountered such an expression, the compiler runs the corresponding metaprogram, wrapping itself in a `Mirror` and passing this mirror to the metaprogram. After the metaprogram returns, its result replaces the original expression. There are some details to how `meta` interacts with lexical scoping, but we will not touch them in this section leaving them for subsection 9.3.3.

As a result, in this new system, macro applications such as `materialize[MyConfig]` expand in two steps. First, the invocation of the macro gets inlined and occurrences of `T` in the macro body are replaced with their actual values, resulting in code resembling `meta { if (isImmutable(t"MyConfig")) ... }`. Afterwards, the meta expression gets evaluated, which results in either a dummy instance of `Immutable` (line 6) or in a compilation error (line 7). The location of the error is specified via an origin, which is a more principled equivalent of scala.reflect's position.

Going back to the architecture of scala.meta described in subsection 8.1.1, we recall that most scala.meta metaprograms and APIs require certain capabilities to be able to execute. For example, `isImmutable` calls semantic APIs, so it needs a mirror. Additionally, it uses quasiquotes, so it needs a dialect.

This contract is successfully met by meta blocks, because they provide a mirror, which is the most powerful capability in scala.meta. Note how all this machinery works transparently for the metaprogrammer thanks to Scala implicits.

Runtime execution. In an experimental branch of scala.meta, we have a prototype of a runtime mirror that works on top of the JVM reflection. Apart from type tags, which we have not worked on yet, our implementation achieves feature parity with the functionality described in subsection 3.1.4.

However, given the limitations of the JVM as well as major difficulties in implementing runtime introspection for platforms like Scala.js and Scala Native, there is a tendency in the Scala community to shy away from runtime metaprogramming. Therefore, we are unsure whether we should support runtime execution in scala.meta, and to what extent that would be practically useful. The fate of the prototype of a runtime mirror for scala.meta remains future work.

```
scala> final class Metadata { ... }
defined class Metadata

scala> final class MyConfig[T](payload: T, metadata: List[Metadata])
defined class MyConfig

scala> val c = new MyConfig(42, Nil)
c: MyConfig[Int] = MyConfig@29ea179c

scala> import scala.meta._
import scala.meta._

scala> implicit val m = Mirror(".../scala-library-2.11.8.jar")
m: Mirror = ...

scala> val t = c.getType
t: Type = MyConfig

scala> isImmutable(t)
res2: Boolean = false
```

8.2 Language model

The language model of scala.meta is designed from scratch to: 1) minimize the surface of the API, and 2) ensure interoperability with existing compilers and tools - all that while maintaining feature parity with scala.reflect (chapter 3).

In order to achieve the first goal, as many concepts as possible are modelled with syntax trees. In order to achieve the second goal, the model is not tied to any particular implementation. There exist converters that transform existing language models to and from scala.meta (section 8.4).

Analogously to section 3.2, in this section we will to compile a simple program, and see how scala.meta represents it. We are going to use the same program that we used to illustrate the language model of scala.reflect.

```
object Test {
  // create a list and compute its head
  val xs = List(42)
  val head: Int = xs.head
  println((xs, head))
}
```

An experimental branch of scala.meta contains a compiler plugin called "scalahost" that injects itself after the `typer` phase of the Scala compiler and converts all typechecked trees from the scala.reflect format to the scala.meta format. By providing an internal debug flag, we will ask scalahost to print out the results of conversion to the console.

```
$ scalac -Xplugin:scalahost.jar -Dconvert.debug Test.scala
```

The first part of the printout shows the prettyprinted representation of a scala.meta tree corresponding to the program being compiled. Note that at this point this tree already carries both syntactic and semantic information about the source.

```
[[scala.meta trees at the end of typer]]// Scala source: Test.scala
object Test {
  // create a list and compute its head
  val xs = List(42)
  val head: Int = xs.head
  println((xs, head))
}
...
```

At a glance, it becomes apparent that scala.meta does not throw away formatting details (including comments) and does not perform desugarings. The abstract syntax tree produced by scalahost looks exactly the same as the code that we have provided in the test program. This is an immediate improvement upon scala.reflect.

In order to achieve full fidelity in representation of syntactic details, scala.meta remembers all tokens that constitute trees that correspond to source files.

Tokens are atomic fragments of Scala programs that capture the entirety of Scala's lexical syntax. In addition to identifiers, keywords, delimiters and literals, we also model syntactic trivia (whitespace and comments) as dedicated tokens, though we may choose to coalesce adjacent whitespace characters together in the future to make things more efficient. For example, here is how the tokens look like for the scala.meta abstract syntax tree in question.

```
object (0..6),  (6..7), Test (7..11),  (11..12),
{ (12..13), \n (13..14),  (14..15),  (15..16),
// create a list and compute its head (16..53),
\n (53..54),  (54..55),  (55..56), val (56..59),
 (59..60), xs (60..62),  (62..63), = (63..64), ...
```

An important consequence of retaining tokens as a low-level representation of abstract syntax trees is that tokens automatically provide positions for their trees. An AST has an extent that begins at the start of its first token and ends at the end of its last token.

The second part of the printout contains internal representation of the converted tree. Next to some AST nodes we can see numbers, which stand for bits of semantic information associated with these nodes.

```
1  [[scala.meta trees at the end of typer]]// Scala source: Test.scala
2  ...
3  Source(Seq(
4    Defn.Object(Nil, Term.Name("Test")[1]{1}, Template(
5      Nil,
6      Nil,
7      Term.Param(Nil, Name.Anonymous()[2], None, None){1},
8      Some(Seq(
9        Defn.Val(
10         Nil, Seq(Pat.Var.Term(Term.Name("xs")[3]{2})),
11         None,
12         Term.Apply(
13           Term.Name("List")[4]{3}<1>,
14           Seq(Lit(42){4})){5}),
15       Defn.Val(
16         Nil, Seq(Pat.Var.Term(Term.Name("head")[5]{6})),
17         Some(Type.Name("Int")[6]),
18         Term.Select(
19           Term.Name("xs")[3]{2}<2>,
20           Term.Name("head")[7]{6}){4}),
21       Term.Apply(
22         Term.Name("println")[8]{7}<3>,
23         Seq(Term.Tuple(Seq(
24           Term.Name("xs")[3]{2}<2>,
25           Term.Name("head")[5]{6}<4>)){8}<5>)){9}))))))
26 ...
```

Scala.meta trees are much more detailed than scala.reflect trees. For example, in the printout above, we can see a dedicated `Term.Tuple` node (line 23) that represents tuple literals which don't exist in scala.reflect. Also, we can notice that names are modelled as trees and modifiers are modelled as lists of trees (all of them are empty in this particular case), instead of being represented with non-tree data structures like in scala.reflect.

An important consequence of the comprehensive nature of scala.meta trees is that it is impossible to create a scala.meta tree that violates language syntax. For example, the tree for a concrete val is defined as `@ast class Val(mods: Seq[Mod], pats: Seq[Pat] @nonEmpty, decltpe: Option[impl.Type], rhs: Term)`, ensuring that only correct types of AST nodes can be inserted into its fields. This significantly improves user experience of quasiquotes (section 8.3).

In addition to the fields responsible for syntactic structure, trees may have denotations, types and desugarings. This is how scala.meta trees store semantic information, as shown on the remainder of the printout.

```
[[scala.meta trees at the end of typer]]// Scala source: Test.scala
...
[1] {10}::_empty_.Test
[2] {0}::_empty_.Test.this
[3] {1}::_empty_.Test.xs()Lscala/collection/immutable/List;
[4] {11}::scala.collection.immutable.List
[5] {1}::_empty_.Test.len()I
[6] {12}::scala#Int
[7] {5}::scala.collection#IterableLike.head()Ljava/lang/Object;
[8] {13}::scala.Predef.println(Ljava/lang/Object;)V
...
```

In the printouts above, numbers in brackets signify denotations, i.e. internal data structures of scala.meta that represent meanings for names. A denotation consists of a prefix, represented by a number in braces to the left of `::`, and a symbol, represented by a string to the right of `::`. Much like denotations, prefixes and symbols are internal data structures and are not exposed in the public API.

Prefixes carry the type of the owner from which the referenced definition is selected. This solves the problem of symbols forgetting their signatures in scala.reflect.

For example, the denotation `[7]` represents the meaning of `head` in `xs.head`. Now, it not only knows that `head` comes from `IterableLike`, but it also remembers that its prefix is `{5}`, which stands for `List[Int]`. As a result, when a metaprogrammer asks `head` about its type, it will be able to correctly respond with `Int`, unlike a scala.reflect symbol that would have responded `T`, because `IterableLike.head` is generic (section 3.5).

Symbols carry the identity of the referenced definition. Unlike in scala.reflect, symbols are dumb identifiers. Global definitions are represented with fully-qualified names, possibly with full signatures to disambiguate overloaded methods. Local definitions are represented with UUIDs. In addition to that, all symbols carry a platform-dependent token that indicates their origin in order to distinguish same-named symbols loaded from different source files or class files.

In scala.reflect, references are linked to definitions via symbols. In order to find out, whether a given reference to a given definition or whether two references mean the same, one compares their symbols via reference equality (subsection 3.2.3).

In scala.meta, references are linked to definitions via names, which are ASTs that encapsulate denotations. Syntactically, every reference and every definition have a name (for example, `val xs = List(42)` has a name on line 10, and a reference to `xs` is also represented by a name on line 24). In order to find out, whether a given reference to a given definition or whether two references mean the same, one compares their names via the semantic equality operator `Tree.=:=`. Semantic equality looks into the denotations internally underlying the names and uses their symbols to provide an answer.

```
[[scala.meta trees at the end of typer]]// Scala source: Test.scala
...
{1} Type.Singleton(Term.Name("Test")[1]{1})
{2} Type.Apply(Type.Name("List")[13], Seq(Type.Name("Int")[6]))
{3} Type.Singleton(Term.Name("List")[4]{3}<1>)
{4} Type.Name("Int")[6]
{5} Type.Apply(Type.Name("List")[13], Seq(Type.Name("Int")[6]))
{6} Type.Name("Int")[6]
{7} Type.Method(
      Seq(Seq(
        Term.Param(
          Nil, Term.Name("x")[14]{19},
          Some(Type.Name("Any")[15]), None){19})),
      Type.Name("Unit")[16])
...
```

In the printouts above, numbers in braces signify types of abstract syntax trees and prefixes of denotations.

In scala.meta, we represent types with abstract syntax. For example, an application of type `List` to a type `Int` is modelled as a type application syntax tree from the language model, as illustrated by type `{5}`.

In order to accommodate all types from the language specification, we had to introduce `Type.Method` and `Type.Lambda` that do not have concrete syntax. These types are necessary to model non-value types from the specification - method types, polymorphic method types and type constructors. For example, `println` has type `{7}`, which stands for a method that has parameter `x: Any` and returns `Unit`.

As a result, in our design, types can use all the infrastructure that already exists for abstract syntax trees. For example, note how types can themselves contain names which, also being abstract syntax trees, can also contain denotations and maybe even other types and desugarings.

```
[[scala.meta trees at the end of typer]]// Scala source: Test.scala
...
<1> Term.ApplyType(
      Term.Name("List")[4]{14}<6>,
      Seq(Type.Name("Int")[6])){15}
<2> Term.Select(
      Term.This(Name.Indeterminate("Test")[1]){1},
      Term.Name("xs")[3]{2}){5}
<3> Term.Select(
      Term.Select(
        Term.This(Name.Indeterminate("scala")[9]){12},
        Term.Name("Predef")[10]{13}){13},
      Term.Name("println")[8]{7}){7}
...
```

In the printouts above, numbers in angle brackets represent desugarings. When converting scala.reflect trees to scala.meta trees, scalahost remembers desugared representations of corresponding pieces of syntax.

For example, this is how scala.meta knows that `List` in `List(42)` actually stands for `List[Int]` (desugaring `<1>`). In that desugaring, `List` can be desugared further to mean `List.apply` (desugaring `<6>`, not shown on the printout).

Much like types, desugarings are full-fledged trees, which makes it possible to treat them uniformly with other parts of the language model.

From the discussion above, we can see that scala.meta represents programs in a minimalistic, yet powerful way. The public part of the language model only contains tokens and trees, which is surprisingly enough for feature parity with scala.reflect that contains trees, symbols, types and about a dozen other auxiliary data structures. In addition to providing functionality similar to scala.reflect, scala.meta also can provide a more precise view of Scala programs, comprehensively capturing syntactic details and giving metaprogrammers explicit control over desugarings.

8.3 Notations

Scala.meta supports quasiquotes as its one and only notation for reflection artifacts. This design decision is the direct consequence of our experience with scala.reflect.

Since statically-typed notations, i.e. reify (subsection 3.3.1) and type tags (subsection 3.3.3), did not perform well in scala.reflect, we decided to forgo them. To the contrast, quasiquotes (subsection 3.3.2) garnered very positive user feedback, so we implemented them in scala.meta.

Syntactic aspect. Scala.meta trees are more detailed than scala.reflect trees, so scala.meta supports more quasiquote interpolators than scala.reflect (16 versus 5). There are several familiar ones, e.g. statements, types and patterns, which are similar to scala.reflect, but there are also more exotic interpolators, e.g. import clauses, import selectors and modifiers, which do not have analogues in scala.reflect. For comprehensive documentation, refer to [16].

```
scala> import scala.meta._
import scala.meta._

scala> q"List(42)"
res0: Term.Apply = List(42)

scala> t"List[Int]"
res1: Type.Apply = List[Int]

scala> p"case List(x) => x"
res2: Case = case List(x) => x

scala> p"List(x)"
res3: Pat.Extract = List(x)

scala> enumerator"x <- List(42)"
res4: Enumerator.Generator = x <- List(42)

scala> importer"foo.bar"
res5: Importer = foo.bar

scala> importee"_"
res6: Importee.Wildcard = _

scala> mod"private"
res7: Mod.Private = private
```

Much like in scala.reflect, in scala.meta quasiquotes support unquoting and unquote splicing and make it possible to statically validate these operations based on the type of the unquotee and the location where it is inserted.

Unlike in scala.reflect, in scala.meta it is impossible to create trees that violate language syntax, so scala.meta quasiquotes provide additional safety guarantees in comparison with scala.reflect quasiquotes.

For example, if we compare the REPL printout below with the analogous printout for scala.reflect (subsection 3.3.2), we will see that scala.meta statically rejects both attempts to create nonsensical trees, whereas scala.reflect not only compiles the second attempt, but also successfully runs it, producing a syntactically invalid tree.

```
scala> val args = List(q"42")
args: List[Lit] = List(42)

scala> val list42 = q"List(..$args)"
list42: Term.Apply = List(42)

scala> q"println($list42)"
res8: Term.Apply = println(List(42))

scala> q"class $list42"
<console>:32: error: type mismatch when unquoting;
 found   : Term.Apply
 required: Type.Name
       q"class $list42"
               ^

scala> q"val weird: $list42 = ???"
<console>:32: error: type mismatch when unquoting;
 found   : Term.Apply
 required: Option[Type]
       q"val weird: $list42 = ???"
                    ^
```

Analogously to scala.reflect, scala.meta quasiquotes support pattern matching, allowing to conveniently deconstruct abstract syntax trees using unquoting and unquote splicing.

Scala.meta trees are more precise than in scala.reflect, so the resulting types of matched variables are also more precise. For example, note how `fn` in the pattern match below has type `Term` as opposed to just `Tree` in the same pattern match performed by scala.reflect quasiquotes.

```
scala> val list42 = q"List(42)"
list42: Term.Apply = List(42)

scala> val q"$fn(..$args)" = list42
fn: Term = List
args: Seq[Term.Arg] = List(42)
```

Finally, scala.meta quasiquotes also support type-directed unquoting and unquote splicing via dedicated typeclasses that express lifting and unlifting.

In scala.meta, these typeclasses have two type parameters, with the second parameter controlling the non-terminal representing the position of insertion or extraction. This additional functionality is possible thanks to scala.meta trees being more precise.

```
scala> implicit def liftInt[O <: Int, I >: Lit]: Lift[O, I] =
     | Lift{ x => Lit(x) }
liftInt: [O <: Int, I >: Lit]=> Lift[O,I]

scala> q"${42}"
res12: Stat = 42

scala> implicit def unliftInt[I >: Lit]: Unlift[I, Int] =
     | Unlift{ case Lit(x: Int) => x }
unliftInt: [I >: Lit]=> Unlift[I,Int]

scala> val q"${fortyTwo: Int}" = res12
fortyTwo: Int = 42
```

Semantic aspect. This part of the story of quasiquotes in scala.meta is exactly the same as in scala.reflect. Quasiquotes deliberately allow ill-typed code snippets for the sake of flexibility, so resulting trees do not contain any semantic information.

To put it in a nutshell, the design of scala.meta quasiquotes is mostly similar to the design of scala.reflect quasiquotes. Thanks to scala.meta trees being more detailed and more precise, user experience has markedly improved, but notation remained mostly the same. Internal implementation has been significantly simplified, but that discussion is outside the scope of our dissertation. Finally, scala.meta inherits the same problems with semantic information and hygiene that are inherent to scala.reflect quasiquotes (subsection 3.3.4). Addressing these problems remains future work.

8.4 Environments

As explained in subsection 8.1.1, different scala.meta APIs require different capabilities to run. Therefore, the question of whether an environment can run a scala.meta metaprogram is equivalent to a question of whether the environment can provide capabilities requested by the metaprogram.

At the moment, scala.meta defines two capabilities: dialects that power syntactic APIs and mirrors that power semantic APIs.

Dialects that correspond to Scala 2.10, Scala 2.11, Dotty and popular build definition formats are readily available in the standard distribution of scala.meta. Accompanying these dialects are in-house implementations of a tokenizer, a parser and a prettyprinter for Scala syntax. As a result, syntactic APIs of scala.meta are available in any environment that can run these implementation, which at the moment includes all JVM-based environments (the Scala compiler, Dotty and the JVM runtime). In the future, we plan to expand this list to include Scala.js and Scala Native.

Mirrors are completely different. According to a conservative estimate (obtaining the LOC count of files in the `typechecker` package of the Scala compiler and in the `internal` package of the scala.reflect language model), an analogue of the functionality required from a mirror comprises more than 50k lines of code of `scalac`. Reimplementing this logic on top of the scala.meta language model in a reasonably compatible way would most likely require multiple years of effort, so we decided against providing an in-house implementation of mirrors.

As a result, scala.meta only specifies the contract for `Mirror` and requires environments to provide their own implementations of this contract. Learning from the experience with scala.reflect whose hundreds of APIs clearly deterred third-party implementors, we minimized the surface of `Mirror` to just a dozen methods. The surface of semantic APIs exposed to metaprogrammers is much richer, but it is all funneled through the dozen methods in mirrors.

In the listing below, we can see an adapted definition of `Mirror` taken from an experimental branch of scala.meta. `@opaque` is a macro annotation (section 5.3) that makes all members in the annotated definition private to scala.meta in order to make sure that it is used as a capability and not as a source of APIs.

```
@opaque
trait Mirror {
  def dialect: Dialect

  def typecheck(tree: Tree): Tree

  def defns(ref: Ref): Seq[Member]
  def members(tpe: Type): Seq[Member]
  def supermembers(member: Member): Seq[Member]
  def submembers(member: Member): Seq[Member]

  def isSubtype(tpe1: Type, tpe2: Type): Boolean
  def lub(tpes: Seq[Type]): Type
  def glb(tpes: Seq[Type]): Type
  def supertypes(tpe: Type): Seq[Type]
  def widen(tpe: Type): Type
  def dealias(tpe: Type): Type
}
```

Creating a mirror for an environment requires developing two high-level components: 1) a *converter* that translates environment artifacts into scala.meta artifacts and vice versa, 2) an *adapter* that translates the calls to the `Mirror` API to the calls to internal metaprogramming APIs of the environment. Let us see how this worked out in our experiments.

Scala compiler. Scalahost (section 8.2) provides a prototype implementation of `Mirror` that wraps `scala.tools.nsc.Global`, i.e. the Scala compiler. We have successfully used this mirror to execute scala.meta metaprograms at compile time, achieving an implementation of an early predecessor of new-style macros described in chapter 9.

A `scalac` implementation of the scala.reflect `Universe` was available automatically from day one of scala.reflect, because `Global` is a scala.reflect universe (subsection 3.4.1). The situation with the scala.meta `Mirror` is drastically different. Implementing the adapter was very easy, but the converter caused and keeps on causing us a lot of trouble.

First, the main challenge in implementing a converter is the desugaring problem. Since compiler trees are heavily desugared (section 3.2), and scala.meta trees are designed to comprehensively describe language elements, a lot of work is required to convert from scala.reflect to scala.meta (the reverse direction is much simpler, because scala.meta trees are more detailed).

A particularly difficult task is to take a desugared scala.reflect tree, figure out which of the dozens desugarings have been applied to it by the Scala compiler and then undo these desugarings. Unfortunately, the compiler rarely keeps track of original, undesugared trees, so within the current infrastructure it is impossible to carry out this task.

We tried multiple approaches that revolved around patching the typechecker to retain information about desugarings, but we quickly hit a complexity wall. The desugarings are numerous and undocumented, so significant engineering effort is required to undo them. Judging from our initial experience, it may very well be that the desugaring problem does not have a practical solution. Confirming or disproving that is an extremely critical element of our future work.

Secondly, another important challenge in implementing a converter is obtaining scala.meta trees from the compilation classpath. Translating scala.reflect symbols that represent the definitions from the classpath is fairly straightforward, but scala.meta trees obtained this way only have signatures and lack bodies. This is not a problem per se, because that is the best that scala.reflect can do anyway, but it is nonetheless unsatisfying.

As a result, in scala.meta we postulate mandatory AST persistence for syntax trees after typer. Thanks to the results of Mathieu Demarne and Adrien Ghosn [30], we know that AST persistence is feasible and practical, so we are now experimenting with it. Scalahost patches the JVM backend of the Scala compiler to save serialized scala.meta trees into class files, so that we can later load these trees and gain access to their original bodies.

To sum it up, implementing a Scala compiler mirror requires non-trivial engineering effort to restore information about code that the compiler typically throws away. After fighting with the Scala compiler for quite a while, we now appreciate the compilers that are designed with programmability in mind.

Dotty. We have been collaborating with the Dotty team to implement a mirror that would allow to run scala.meta metaprograms in Dotty. Following through the initial prototype is an exciting prospect for future work. Moreover, additional effort may be required to support runtime reflection of JVM programs produced by Dotty, because Dotty features compilation flags that enable whole-program optimization [93].

IntelliJ. From the very inception of scala.meta, we have been working together with the IntelliJ team, and recently this collaboration has resulted in a prototype of an IntelliJ mirror. In his dissertation, Mikhail Mutcianko from the IntelliJ team describes design and implementation of the mirror [86].

JVM. Based on our experience with developing a JVM universe for scala.reflect (subsection 3.4.4), we decided against building a classloader-based JVM mirror for scala.meta. Instead, we require a classpath, and based on that classpath we launch an instance of the Scala compiler. We then proxy the calls to the mirror API to an auxiliary Scala compiler mirror that wraps the underlying compiler instance.

Other runtimes. The situation with mirrors for other runtimes is analogous to the same situation in scala.reflect (subsection 3.4.5 and subsection 3.4.6). Neither Scala.js nor Scala Native currently provide enough reflective facilities to host a scala.meta mirror, and it is doubtful that the situation will change in the future.

8.5 Operations

Much like section 3.5, this section is also going to be a practical exploration of a metaprogramming API. In the REPL printout below, we will demonstrate a mixture of syntactic and semantic APIs implemented in scala.meta. The syntactic part has been recently shipped in a stable release, but the semantic part is only available as a prototype in an experimental branch of the project.

We will begin with syntactic APIs, which according to subsection 8.1.1, require a wildcard import and a dialect. Let us work with the `Scala211` dialect.

```
scala> import scala.meta._
import scala.meta._

scala> import scala.meta.dialects.Scala211
import scala.meta.dialects.Scala211
```

Obtaining a scala.meta tree is similar to scala.reflect. We can: 1) parse it from an `Input`, i.e. anything that can provide an array of characters, 2) create it with a quasiquote (subsection 3.3.2), 3) receive it from an environment, e.g. from a macro engine during macro expansion.

Unlike in scala.reflect, parsing does not require a helper (i.e. a toolbox or a macro context) and is not hardcoded to parse just terms. In a type argument of `parse`, metaprogrammers can choose from more than a dozen non-terminals of the Scala grammar that they would like the scala.meta parser to produce.

```
scala> "List(42)".parse[Term]
res0: Parsed[scala.meta.Term] = List(42)

scala> "List(42)".parse[Type]
res1: Parsed[scala.meta.Type] =
<input>:1: error: end of file expected but ( found
List(42)
    ^

scala> q"List(42)"
res2: Term.Apply = List(42)

scala> t"List[Int]"
res3: Type.Apply = List[Int]
```

Similarly to scala.reflect, there are multiple ways to prettyprint a tree, with analogues existing for both `show` and `showRaw` from scala.reflect. Thanks to the comprehensive nature of scala.meta trees, prettyprinting produces exactly the syntax that was used to create a tree, including all formatting details.

```
scala> val x = "val x = 42 // the answer to...".parse[Stat].get
x: Stat = val x = 42 // the answer to...

scala> x.syntax
res4: String = val x = 42 // the answer to...

scala> x.structure
res5: String =
Defn.Val(Nil, Seq(Pat.Var.Term(Term.Name("x"))), None, Lit(42))
```

As shown above, low-level syntactic details are not encoded in the case class structure. Instead, these details are externalized in tokens - first-class entities of the scala.meta language model that are associated with abstract syntax trees during parsing.

```
scala> x.tokens
res6: Tokens =
Tokens(, val,  , x,  , =,  , 42,  , // the answer to..., )

scala> x.tokens.structure
res7: String = Tokens(BOF [0..0), val [0..3),  [3..4),
x [4..5),  [5..6), = [6..7),  [7..8), 42 [8..10),
  [10..11), // the answer to... [11..30), EOF [30..30))
```

In order to start experimenting with semantic APIs, we will need a mirror. For the sake of simplicity, we will use a runtime mirror that is created from a classpath (section 3.4). This does not lose generality, because exactly the same API is available at compile time.

```
scala> implicit val m = Mirror(".../scala-library-2.11.8.jar")
m: Mirror = ...
```

In contrast to scala.reflect, the semantic API of scala.meta does not use anything but trees. Symbols are represented by names, and types are represented by their abstract syntax (section 8.2).

Moreover, there is no publicly visible distinction between attributed and unattributed trees. As we have seen in section 8.2, the language model includes the notions of denotations, types and desugarings, but these notions are internal to the scala.meta framework and its implementors, which means that users do not know about them.

Whenever a metaprogrammer requests semantic information from a tree that internally happens to be unattributed, scala.meta calls `Mirror.typecheck` on that tree, and it is a responsibility of the mirror to produce a correct result. Specifying exactly what "correct" means is inherently linked to hygiene (subsection 3.3.4), so we leave that for future work. In the meanwhile, our prototype typechecks unattributed trees at the top level of the underlying mirror.

The REPL printout below illustrates name resolution and signature inspection - the tasks that are typically done with symbols in the scala.reflect language model. We start with `t"List"`, a quasiquote that produces a regular abstract syntax tree, and then call `Ref.defn` on that tree.

`Ref.defn` sees that `t"List"` is internally unattributed (because quasiquotes create unattributed trees as per subsection 3.3.2), so it calls `Mirror.typecheck` on a mirror that is provided to it as an implicit parameter. An unqualified `List` typechecks as a reference to a type alias that is declared in the `scala` package.

Having gotten ahold of an attributed tree corresponding to `t"List"`, `Ref.defn` resolves the reference `List` and returns the result to the metaprogrammer. However, unlike scala.reflect, the metaprogrammer does not get back a symbol, but instead sees a representation of the `scala.List` type alias encoded in an abstract syntax tree.

```
scala> val list = t"List".defn
list: Defn.Type = type List[+A] = List[A]
```

In order to inspect the signature of a definition, we do not need any special APIs like `Symbol.info` or `Symbol.typeParams`. Instead, we inspect the structure of the tree and extract the relevant parts. Alternatively, if writing a quasiquote to destructure a definition is too verbose, we can call a helper method that does the same internally.

```
scala> list.structure
res8: String = Defn.Type(
  Nil, Type.Name("List"),
  Seq(Type.Param(Seq(Mod.Covariant()), Type.Name("A"), ...)),
  Type.Apply(Type.Name("List"), List(Type.Name("A"))))

scala> val q"..$_ type $_[..$tparams] = $_" = list
tparams: Seq[Type.Param] = List(+A)

scala> list.tparams
res9: Seq[Type.Param] = List(+A)
```

Naturally, the resulting abstract syntax trees can also participate in semantic APIs. For example, it is possible to obtain the body of the `scala.List` type alias and recursively resolve it to a definition, obtaining an abstract syntax tree representing `scala.collection.immutable.List`. In that case, no call to `typecheck` would be required, because that tree was returned by the mirror and preattributed in advance.

Scala.meta provides many more semantic APIs, with the goal to achieve feature parity with scala.reflect. For example, similarly to scala.reflect, there exist `Type.<:<`, `Type.members` and the like.

```
scala> t"List[Int]" <:< t"List[Any]"
res10: Boolean = true

scala> t"List[Int]".foreach(println)
override def companion: GenericCompanion[List] = ???
def ::[B >: Int](x: B): List[B] = ???
...
```

There are several noteworthy facts about the REPL printout above. First, scala.meta does not forget prefixes when enumerating members of a type. For example, the `List[A].::` method, whose definition looks like `def ::[B >: A](x: B): List[B]`, remembers the fact that it has been obtained from `List[Int]` and adjusts its signature accordingly.

Secondly, since our AST persistence technology is still experimental, more often than not we do not have ASTs for dependencies loaded from the classpath. Therefore, we have to emulate these trees by reconstructing their signatures from available metadata and replacing their bodies (if applicable) with a marker. This looks somewhat strange, but does not affect introspective capabilities of scala.meta.

In the examples above, we have seen that scala.meta noticeably improves upon scala.reflect in comprehensiveness and simplicity of its API. Even though at the time of writing our semantic operations are only available in a prototype, we are very excited by our initial results and are looking forward to developing them further in the future.

8.6 Conclusion

Scala.meta was originally created to supersede scala.reflect (chapter 3) in the capacity of the metaprogramming API powering Scala macros (chapter 4 and chapter 5).

Initially, the design of scala.meta was dictated by the desire to avoid three problems of scala.reflect: 1) unconventional architecture requiring knowledge of advanced language features (subsection 3.1.1), 2) extravagant and brittle API based on compiler internals (section 3.2), 3) lackluster interoperability with third-party tools (chapter 7).

In order to improve on scala.reflect, we designed scala.meta from first principles. Starting from scratch, we created an independent language model for Scala (section 8.2) and designed a set of operations that works with it (section 8.5), carefully documenting the capabilities that are required to run these operations (subsection 8.1.1).

Moving from data structures of compiler internals to a language model defined in a library was somewhat ironic. It was just several years ago that the move in the opposite direction - from `Code` and `Manifest` (section 2.3) to compiler internals - was considered an important achievement. Nevertheless, our experience with scala.meta has demonstrated that having a standalone language model is paramount to solving the interoperability problem of scala.reflect.

Our progress with scala.meta looks quite promising. Even though we do not yet have a fully-featured implementation, scala.meta has already started catalyzing the next step in the evolution of metaprogramming in Scala.

First, we have successfully presented a prototype of a new macro system that works both in the Scala compiler and in IntelliJ (section 9.6) - something that has been out of reach of scala.reflect for almost five years.

Secondly, we have discovered that a unified metaprogramming API can be useful not only as a standard facility underlying compile-time and runtime reflection, but also as a foundation for the ecosystem of novel developer tools (section 8.4).

9 New-style macros

Def macros (chapter 4) and macro annotations (section 5.3) have become an integral part of the Scala ecosystem. They marry metaprogramming with types and work in synergy with other language features, enabling practically important use cases (chapter 6).

Unfortunately, Scala macros have also gained notoriety as an arcane and brittle technology. The most common criticisms of Scala macros concern their subpar tool support (chapter 7) and overcomplicated metaprogramming API powered by scala.reflect (chapter 3).

While trying to fix these problems via evolutionary changes to the current macro system, we have realized that they are all caused by the decision to use compiler internals as the underlying metaprogramming API. Extensive use of desugarings and existence of multiple independent program representations may have worked well for compiler development, but they turned out to be inadequate for a public API.

The realization that we cannot make meaningful progress while staying within the confines of scala.reflect meant that our macro system needs a redesign. We took the best part of current macros - their smooth integration into the language - and discarded the rest, replacing scala.reflect with scala.meta (chapter 8) and coming up with a lightweight declaration syntax. In this chapter, we present the current version of the design.

Unlike most technologies presented in this dissertation, the new macro system only exists as a prototype [13, 86]. Nevertheless, based on the feedback from the community, we are confident that it is moving in the right direction and are excited to develop it further.

The new design was created together with Denys Shabalin and Martin Odersky and was further elaborated with Mikhail Mutcianko, Vladimir Nikolaev, Vojin Jovanovic, Sébastien Doeraene, Dmitry Petrashko, Valentin Rutz and the tool support squad (chapter 7). The prototype was developed with contributions of Hiroshi Yamaguchi, Tamer Mohammed Abdul-Radi, Takeshi D. Itoh, Oleksandr Olgashko, David Dudson, Dale Wijnand and Ólafur Páll Geirsson.

9.1 Motivation

In addition to the four design goals stated in section 4.1, we have introduced further requirements to make sure that the new macro system is free from the most important problems characteristic for its predecessor.

5) Enable comprehensive tool support. By associating themselves with scala.reflect, the current macros make it virtually impossible for third-party tools to support them fully. Code comprehension, error reporting, incremental compilation, testing, debugging - all suffer because of the limitations of scala.reflect (chapter 7).

6) Simplify the metaprogramming API. Practical examples from subsection 3.1.2, section 4.2 and other sections show that even not very complicated macros occasionally require knowledge of compiler internals. We would like the new macro system to have an API that avoids such problems.

7) Prevent inadvertent name capture. As illustrated in section 4.2, it is trivial to make hygiene mistakes, and it is hard to notice them. The current macro system neither prevents these mistakes from happening, nor assists metaprogrammers with detecting them. Our anecdotal experience with reviewing open-source Scala macros shows that hygiene bugs are one of the most common oversights in macro writing.

8) Simplify macro declaration syntax. From subsection 4.3.4 and subsection 5.3.2, we can see that defining macros in the current macro system is associated with significant amounts of boilerplate. Now that the programmers in the Scala community are familiar with the concept of language-integrated compile-time metaprogramming, we can afford switching to a less explicit, but more concise syntax.

9) Lift the separate compilation restriction. The inability of the current macro system to handle macro applications and their macro impls in the same compilation run (subsection 4.3.3) presents a practical inconvenience. Since macro-generating macros are almost never used in Scala, this restriction is rarely a significant problem, but it does bring the trouble of having to tinker with the build system.

10) Provide feature parity. Even though the current macro system has a lot of shortcomings, it also has a lot of users. While we probably will not be source-compatible with a great number of existing scala.reflect metaprograms, we feel the pressure to strive for feature parity between the current and the new macro systems.

The first three of the additional requirements belong to the domain of scala.meta (chapter 8), which was initially designed specifically to address them. The quest of satisfying the last three requirements is described in subsequent sections of this chapter.

9.2 Intuition

Let us go through the LINQ example from section 4.2, porting it to new-style macros and discussing similarities and differences with the current macro system. This example features a significant amount of domain-specific infrastructure, so you may find it useful to revisit the explanation of this infrastructure provided in the original section where the example was introduced.

```
1  trait Query[T]
2  case class Table[T: TypeTag]() extends Query[T]
3  case class Select[T, U](q: Query[T], fn: Node[U]) extends Query[U]
4
5  trait Node[T]
6  case class Ref[T](name: String) extends Node[T]
7
8  object Query {
9    implicit class QueryApi[T](q: Query[T]) {
10     def map[U](fn: T => U): Query[U] = { ... }
11   }
12 }
13
14 case class User(name: String)
15 val users = Table[User]()
16 users.map(u => u.name)
17 // translated to: Select(users, Ref[String]("name"))
```

In a nutshell, the listing above introduces an infrastructure that represents queries to datasources (lines 1-6), and we want to expose it to users in a convenient way that resembles the operations on the standard library collections (line 8-12). For that, we need a facility that would translate calls to such operations (line 16) into query constructors as shown on line 17.

In section 4.2, we turned `Query.map` into a macro, associating applications of `Query.map` with our custom metaprogram that does the required translation at compile time. The listing below provides the source code of such a macro implemented with the current macro system.

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.blackbox.Context
3
4  object Query {
5    implicit class QueryApi[T](q: Query[T]) {
6      def map[U](fn: T => U): Query[U] = macro QueryMacros.map
7    }
8  }
9
```

```
10 object QueryMacros {
11   def map(c: Context)(fn: c.Tree): c.Tree = {
12     import c.universe._
13
14     // c.prefix looks like:
15     // Query.QueryApi[<T>](<prefix>)
16     val q"$_.$_[$_]($prefix)" = c.prefix
17
18     val node: Tree = fn match {
19       case q"($param) => $body" =>
20         val sym = param.symbol
21         body match {
22           case q"$qual.$_" if qual.symbol == sym =>
23             q"Ref[${body.tpe}](${sym.name.decodedName.toString})"
24         }
25     }
26
27     q"Select($prefix, $node)"
28   }
29 }
```

In the listing above, `Query.map` is a macro def (line 6) that is associated with the metaprogram `QueryMacros.map` (lines 11-28) which is called a macro impl. Whenever the compiler encounters an application of `Query.map`, it calls the corresponding macro impl to generate code that will replace the original application at compile time. In the macro impl, we analyze the `fn` argument (lines 18-25), translate it into a corresponding `Node` (lines 19-24), and then produce a `Select` query that represents the application of `Query.map` (line 27).

User interface of def macros is a seamless extension to the existing language interface. Thanks to macro defs looking like regular methods and macro applications looking like regular method applications, users are likely to not even realize that they are using macros.

Unfortunately, metaprogrammer interface leaves much to be desired. First, in the current macro system, metaprograms have to be defined separately from their signatures, typically involving helper objects and duplication of parameters (lines 10-29). Secondly, the underlying metaprogramming API requires knowing bits and pieces of compiler internals. For example, in the case of `Query.map`, the metaprogrammer had to know the internal representation of the `QueryApi` application (line 16) and internal details of how names are represented (line 23).

New-style macros provide the same user interface, and at the same time significantly improve metaprogrammer interface by enabling lightweight macro declaration syntax and using a better metaprogramming API.

```
1  object Query {
2    implicit class QueryApi[T](q: Query[T]) {
3      inline def map[U](fn: T => U): Query[U] = meta {
4        import scala.meta._
5
6        val q"$_($prefix)" = this
7
8        val node: Tree = fn match {
9          case q"($param: $_) => $body" =>
10           body match {
11             case q"$qual.$field" if qual =:= param =>
12               q"Ref[${body.tpe}](${field.toString})"
13           }
14       }
15
16       q"Select($prefix, $node)"
17     }
18   }
19 }
```

At a glance, new-style macros simply forgo a noticeable amount of ceremony, merging the previously disparate macro defs and macro impls as well as swapping around a few keywords in the process. The underlying metaprogram does not seem to have changed much, still using quasiquotes and key APIs like `Tree.tpe` and `Name.toString`.

First impression notwithstanding, the new design revamps a lot of underlying mechanisms. Below we provide a high-level overview of our new approach to compile-time metaprogramming, and refer curious readers to section 9.3 for information concerning the new expansion mechanism and to chapter 8 for details about the new metaprogramming API underlying the design.

Expansion mechanism. The new design takes the notions of inlining and compile-time execution from the current macro system and turns them into separate language features.

The goal of the `inline` modifier on `Query.map` is to signify that applications of this method are inlined, i.e. replaced with the method body, in which references to enclosing `this`, self, and formal parameters are rewritten accordingly (subsection 9.3.2).

The goal of the `meta` keyword is to demarcate code that executes at compile time and to provide that code with scala.meta capabilities (subsection 8.1.1). The metaprogram written inside the meta scope has access to multiple implicit capabilities and can get ahold of representations of certain values and types from lexical scope. The compiler runs this metaprogram, interprets its result as an abstract syntax tree and replaces the meta expression with that tree (subsection 9.3.4).

When the compiler encounters a call to `Query.map`, it simply inlines the body of `map` into the callsite without performing any compile-time function execution. For the running example of `users.map(u => u.name)`, this inlining produces the result illustrated in the listing below.

```
{
  val prefix$1: QueryApi = QueryApi(users)
  val fn$1: User => String = u => u.name
  meta {
    import scala.meta._

    val q"$_($prefix)" = q"QueryApi(users)"

    val node: Tree = q"u => u.name" match {
      case q"($name: $_) => $body" =>
        body match {
          case q"$qual.$_" if qual =:= name =>
            q"Ref[${body.tpe}](${name.toString})"
        }
    }

    q"Select($prefix, $node)"
  }
}
```

Before inlining a method application, the compiler first hoists the prefix and by-value arguments of the application into temporary variables. This is done in order to guarantee that applications of inline methods are semantically equivalent to applications of regular methods.

Afterwards, the compiler replaces the application with a block that consists of hoisted values and the transformed method body. In the method body, all regular references to `this` and self as well as by-value parameters are rewritten to references to the corresponding temporary variables. If such references are part of a meta scope, they are replaced with scala.meta-based representations of the prefix and the corresponding arguments, without going into temporary variables.

When the compiler comes across a meta expression that is not part of an inline method, it executes the code inside the expression and replaces the expression with the result of execution. For our running example, this produces the desired expansion that invokes query constructors corresponding to the LINQ notation.

New metaprogramming API. Scala.meta noticeably improves on scala.reflect in the convenience of the API and no longer requires metaprogrammers to understand compiler internals in order to write macros.

In particular, on line 6 we do not need to know that construction of the implicit class involves expanding the reference to `QueryApi` into a fully-qualified name and inferring the missing type argument. The particular implementation of the scala.meta API that runs the meta expression may or may not do these desugarings, and scala.meta shields us from this fact. We can use the WYSIWYG pattern `q"$_($prefix)"` in order to unwrap the original prefix of the call.

Moreover, on line 12 we do not have to worry about the compiler internally mangling non-alphanumeric names. Again, even if the underying macro engine internally does name mangling, scala.meta abstracts away such implementation details.

Finally, on line 9 we are able to improve on scala.reflect thanks to more precise quasiquotes. If in the current macro system, we used `q"($name: $_) => ..."` to match the `fn` argument, `name` would capture a scala.reflect `Name` that does not carry any semantic information. In scala.meta, names are full-fledged trees, so we can use `Tree.=:=` to semantically compare the name with the qualifier of the field selection on line 11.

In order to use semantic APIs, scala.meta metaprograms need the mirror capability (subsection 8.1.1), so it is implicitly provided inside meta scopes. As a result, meta expressions can use the full spectrum of APIs available in scala.meta.

To put it in a nutshell, the new-style macro system combines two language features - inline definitions (subsection 9.3.1) and meta expressions (subsection 9.3.3) - in order to provide a more lightweight syntax for the current def macros. Moreover, this macro system does a switchover from scala.reflect to scala.meta, featuring a new API that is more convenient and does not require compiler knowledge to be utilized effectively.

9.3 Inline and meta

9.3.1 Inline definitions

The main motivation behind `inline` is to provide a templating system that can express simple use cases of compile-time metaprogramming in an lightweight declarative style that does not involve explicit introspection.

Towards that end, we introduce a new reserved word: `inline`, which can be used as a modifier for concrete vals, concrete methods and parameters of inline methods, as illustrated in the listing below.

```
inline val x = 4

inline def square(x: Double) = x * x
```

```
inline def pow(b: Double, inline n: Int): Double = {
  if (n == 0) 1
  else b * pow(b, n - 1)
}
```

Inline vals are guaranteed to be compile-time constants, superseding the existing approach of declaring such constants with `final val`. Inline parameters get the same treatment, adding a previously non-existent functionality of guaranteeing that an argument of a method is a compile-time constant.

A value is considered to be a compile-time constant if it is a Scala literal, or it is equivalent to one by the means of inline/meta expansion and constant folding. Future work may extend this notion to custom classes, but this discussion lies outside the scope of the dissertation.

Inline defs are guaranteed to be inlined at compile time, superseding the existing approach of annotating methods with the `@inline` annotation. After introducing `inline`, we expect to deprecate and phase out `@inline`.

The problem with `@inline` is that it does not provide guarantees. As specified in documentation, `@inline` tells the compiler to "try especially hard to inline the annotated method". However, different backends interpret this request differently. The JVM backend ignores this annotation if optimizations are disabled and sometimes skips inlining even when optimizations are enabled. Both Scala.js and Scala Native always inline methods that are marked with this annotation. In contrast, `inline` achieves guaranteed inlining, regardless of the backend.

Inline defs are very similar to macro defs in the sense that they look like regular defs and that they expand at compile time. As a result, the rules in subsection 4.6.1 also apply to inline defs with three exceptions.

First, inline defs effectively final; they cannot be overridden. Inline members also never override other members. The idea of allowing macro defs to override regular defs did not find compelling use cases, so we prohibit this for inline defs.

Secondly, inline defs can have default parameters. Not supporting default parameters for macro defs was an oversight of the initial design, so now we fix this oversight in the new macro system.

Thirdly, inline defs have regular bodies, just like regular defs. When an inline def is typechecked, its body is typechecked according to the usual rules, which means that inline defs are eligible for return type inference. If an inline def does not explicitly specify its result type, the result type gets inferred from the type of its body.

9.3.2 Inline reduction

When the compiler encounters certain patterns of code that involve references to inline vals and applications of inline defs, it will perform the rewritings provided below, if and only if these patterns appear outside the bodies of inline vals and defs.

1) If `prefix.v` refers to an inline value in a method call that involves only compile-time constants or in a condition of a conditional expression that is a compile-time constant, simplify the corresponding expression. This rewriting is necessary to make sure that methods like `pow` from subsection 9.3.1 successfully expand.

2) If `prefix.v` refers to an inline value elsewhere, replace it with the body of `v`.

3) If `prefix.f[Ts](args1)...(argsN)` refers to a fully applied inline method, hoist the prefix and the arguments into temporary variables with fresh names and then replace the expression with the method body. In the body, replace parameter references with references to temporary variables created for the corresponding arguments. Also, replace enclosing `this` and self references with references to the temporary variable created for the prefix.

```
{
  val prefix$1 = prefix
  val param1$1 = arg1
  ...
  val paramN$1 = argN
  <transformed body of f>
}
```

The hoisting is intended to preserve the semantics of method applications under inlining. A method call should have the same semantics with respect to side effects independently on whether the method was made inline or not. If an inline method has by-name parameters, then corresponding arguments are not hoisted.

The rewriting is done in accordance with hygiene. Any references from the method body to its lexical scope will be kept in the rewritten code. If the result of the rewriting references private or protected definitions in the class that defines the inline method, these references will be changed to use accessors generated automatically by the compiler. To ensure that the rewriting works in the separate compilation setting, it is critical for the compiler to generate the accessors in advance. Most of these accessors can be pregenerated by analyzing the bodies of inline methods, except for members that are referred to inside meta scopes. Such references are disallowed, because it is impossible to generate them in advance.

4) If `prefix.f[Ts](args1)...(argsN)` refers to a partially applied inline method, an error is raised. Eta expansion of inline methods is prohibited.

The rules of inline reduction are similar to the rules of feature interaction for macro applications (subsection 4.6.2) as well as relevant parts of the rules of def macro expansion (section 4.4) with four exceptions.

Firstly and most importantly, inline reductions in their current form preclude whitebox expansion (section 4.4). Since bodies of inline vals and defs are typechecked when their definitions are typechecked, potential meta expansions that may happen afterwards will not be able to change their types. Implications of this are discussed in subsection 9.3.5.

Secondly, by-value and by-name arguments behave differently. By-value arguments are hoisted in temporary variables, while by-name arguments remain as they were. This does not affect meta expansions (subsection 9.3.4), but it does make a semantic difference for parts of inline definitions that are not inside meta scopes. Note that `this` and self references always follow the by-value scheme, because there is no syntax in Scala that allows to define them as by-name.

Thirdly, named and default arguments are fully supported. Again, not including them in the initial release of def macros was an oversight, which is now fixed.

Finally, the rules of rewriting mandate the "outside in" style, i.e. calls to inline methods are expanded before possible calls to inline methods in their prefixes and arguments. This is different from how the "inside out" style of def macros (section 4.4), where prefixes and arguments are expanded first. Previously, it was very challenging to take a look at unexpanded trees, but now the metaprogrammer can easily switch between unexpanded and expanded views (section 9.4).

From the discussion above, we can see that inline reduction closely resembles the inlining aspect of the current macro system. The only significant difference is lack of support for whitebox expansion, which will be discussed in subsection 9.3.5.

9.3.3 Meta expressions

A meta expression is an expression of the form `meta { ... }`, where `{ ... }` is some block of Scala code, called meta scope. (In fact, `meta` may prefix arbitrary expressions, but blocks are expected to be used most commonly).

Meta expressions can appear both in the bodies of inline methods (then their expansion is going to be deferred until the enclosing method expands) and in normal code (in that case, their expansion will take place immediately at the place where the meta expression is written).

Meta scopes can contain arbitrary code, but they must return values that are either of type `scala.meta.Term` or are convertible to it via the `scala.meta.Lift` typeclass.

There are standard instances of the typeclass that lift simple values to literals as well as ones that support frequently used collections of liftable values. Metaprogrammers may define and use their own instances as long as they are available in corresponding meta scopes.

Inside meta scopes, metaprogrammers can use a combination of general-purpose and expansion-specific metaprogramming facilities. Refer to section 9.4 for more information.

Since meta scopes must return scala.meta trees, and scala.meta trees are by design statically untyped (section 8.2), the type of meta expressions can not be computed from the inside and has to come from the outside. Therefore, meta expressions can only be used in contexts that specify an expected type.

```
// allowed, expected type provided by the return type
inline def map[U](fn: T => U): Query[U] = meta { ... }

// not allowed, no explicit return type
val x = meta { ... }

// allowed, expected type of a condition is Boolean
if (meta(T.isPrimitive)) { ... }
```

Meta scopes can reference names in their lexical scope outside the enclosing meta expression. While crossing the `meta` boundary, references change their meaning. Concretely, here is how the transformation works for different references:

Inline vals and inline parameters. These are guaranteed to be compile-time constants, so an inline value of type `T` is available inside a meta scope as a regular value of type `T`.

Inline defs. When viewed from within a meta scope, inline defs become tree transformers. Types of their inline parameters are unchanged, their non-inline parameters and return type become typed as `scala.meta.Term`. For example, `inline def pow(b: Double, inline n: Int): Double` transforms into `def pow(b: scala.meta.Term, n: Int): scala.meta.Term`.

Term parameters of an inline method. References to statically known term parameters, enclosing `this` or self become values of type `scala.meta.Term`. This is similar to `c.Expr`/`c.Tree` parameters of macro impls (subsection 4.3.1), but more lightweight, because there is no need to declare these parameters explicitly.

Type parameters of an inline method. References to statically known type parameters become values of type `scala.meta.Type`. This is similar to `c.WeakTypeTag` parameters of macro impls (subsection 4.3.2), but more lightweight, because metaprogrammers do not need to explicitly manifest their interest in given type parameters in order to get access to their representations.

This rule can create clashes between term and type namespaces. If such a clash happens, i.e. if a reference to a type parameter is ambiguous with an eponymous term, the compiler emits an error. This is regrettable, but Scala naming conventions make this situation unlikely, and there is always a workaround of renaming the type parameter.

Global definitions. Statically available types and terms, e.g. `List` or `Int`, are usable inside meta scopes as themselves. This means that meta expressions, just like macro impls, can use the standard library and arbitrary third-party libraries.

Other definitions. Macro scopes cannot reference definitions that do not fall into one of the categories mentioned above. This outlaws usages of local definitions - both terms and types. Such definitions may refer to state that only exists at runtime, so we prohibit them altogether. This has no analogues in the current macro system, because macro impls must be defined in static methods, which means that by definition their scope cannot contain local definitions.

In other words, definitions that are statically available outside meta scopes remain available in meta scopes, term and type arguments of inline methods become available as their representations, while signatures of inline methods are recursively transformed according to the rules above.

From the discussion above, we can see that meta expressions provide an analogue of the compile-time execution part of the current macro system. In addition to achieving feature parity, meta expressions also improve on the corresponding part of def macros by allowing metaprograms to easily obtain representations of their term and type parameters and making it possible to run anonymous snippets of metaprogramming code without wrapping them in a dedicated macro.

9.3.4 Meta expansion

When the compiler encounters a meta expression that appears outside the bodies of inline vals and defs, it expands that expression as described below.

A meta expression is expanded by evaluating its body and replacing the original meta expression with an expression that represents the result of the evaluation. The compiler is responsible for providing the capabilities (section 9.4) necessary for meta scopes to evaluate and for converting between its internal representation for language model elements and representations defined in scala.meta, such as `scala.meta.Term` and `scala.meta.Type`.

Meta expansion works very similarly to the relevant part of the def macro expansion pipeline (subsection 4.4.1). Code in the meta scope is precompiled and then reflectively invoked by the compiler, sharing the class loader with other metaprograms run inside

the compiler. Expansion can return normally or can be interrupted with an exception. Expansion results are typechecked against the expected type of the meta expression. Typecheck results are upcast to the expected type in blackbox style, as discussed in subsection 9.3.5.

Attentive readers may notice that the reflective style of executing meta scopes contradicts the goal of lifting the separate compilation restriction (section 9.1). That is a deliberate decision, because developing a full-fledged interpreter or JIT compiler for scala.meta trees is outside the scope of this dissertation. Nevertheless, this is a very interesting project that we plan to tackle in the future, and some initial work in that direction is described in [85].

Another practicality of meta expansion is the protocol of communication between `inline` and `meta`. On the one hand, a reasonable default for inlining that does not involve meta expressions is to hoist prefixes and by-value arguments (subsection 9.3.2). On the other hand, macro writers default to having unfettered access to abstract syntax trees without needing to write additional boilerplate. These two design preferences are at odds with each other.

Therefore, in our design `inline` both hoists eligible components of inline applications and passes their original representations into meta expressions. This way, both defaults are satisfied and, additionally, if meta expressions need to hoist something, they can use the newly introduced `hoist` API (section 9.4).

In the case when an inline method consists in a single meta expression, the new macro engine removes temporary variables that are produced by hoisting and are not claimed by `hoist`. In the case when an inline method does not contain meta expressions, all temporary variables are retained, because they are necessary to express method application semantics. Finally, in the case of a hybrid inline method, meta expressions can look into representations of hoisted expressions, but they cannot use them in their expansions without calling `hoist` first in order to preserve the order of side effects.

In a nutshell, meta expansion closely resembles the compile-time execution aspect of the current macro system. The only nuance is the interaction with hoisting performed by the mechanism of inline reduction.

9.3.5 Losing whiteboxity

In design goals (section 9.1), we manifested our desire to provide reasonable compatibility with the current macro system. So far, the biggest digression from this course is giving up whitebox expansion. In this section, we discuss what it will cost us to follow this through, identify the aspects of the new design that prevent whiteboxity and propose alternatives.

The main motivation for getting rid of whitebox expansion is simplification - both of the macro expansion pipeline and the typechecker. Currently, they are inseparably intertwined, complicating both compiler evolution and tool support. Therefore, the new design tries to disentangle these systems.

Let us recapitulate the limitations that def macro expansion (section 4.4) applies to applications of blackbox macros, outlining use cases from chapter 6 that blackboxity cannot express. This will not give us the full picture, because there are many macros in the wild that we have not classified, but nonetheless it will provide an understanding of the significant chunk of the Scala macros ecosystem.

1) When an application of a blackbox macro expands into a tree `x`, the expansion is wrapped into a type ascription `(x: T)`, where `T` is the declared return type of the blackbox macro def with type arguments and path dependencies applied in consistency with the particular macro application being expanded (subsection 4.4.1). This invalidates blackbox macros as an implementation vehicle for type providers (section 6.2).

2) When an application of a blackbox macro is used as an implicit candidate, no expansion is performed until the macro is selected as the result of the implicit search (subsection 4.4.2). This makes it impossible to dynamically calculate availability of implicit macros, precluding some advanced aspects of materialization (subsection 6.1.4).

3) When an application of a blackbox macro still has undetermined type parameters after the Scala type inference algorithm has finished working, these type parameters are inferred forcibly, in exactly the same manner as type inference happens for normal methods (subsection 4.4.3). This makes it impossible for blackbox macros to influence type inference, prohibiting fundep materialization (subsection 6.1.4).

4) When an application of a blackbox macro is used as an extractor in a pattern match (section 5.1), it triggers an unconditional compiler error, preventing customizations of pattern matching implemented with macros. This precludes blackbox macros from providing precise types to values extracted from patterns written in external DSLs (section 6.5), preventing a library-based implementation of quasiquotes (subsection 3.3.2).

As we can see, whitebox expansion plays an important role in several use cases, so it is now clear that our desire for simplification is at odds with the ways how the Scala community uses macros. In order to resolve the apparent conflict, we outline several solutions, whose evaluation we leave for future work.

Give up on simplification. This is probably the most obvious approach, in which we admit that tight coupling with the typechecker is intrinsic to the essence of Scala macros and change the new design to enable whitebox expansion.

Concretely, accommodating whiteboxity in the new macro system requires changing the

aspects of the new design specified in subsection 9.3.2 and subsection 9.3.4 according to the plan provided below.

- Force inline reductions and meta expansions inside the typechecker. Currently, both the rules of inline reduction and the rules of meta expansion are intentionally vague about the exact point in the compilation pipeline that does expansions, but whiteboxity will leave us no room for that.
- Delay typechecking of the bodies of inline vals and defs until their expansion. Meta expressions inside inline definitions are not expanded, which means that eager typechecking of these definitions precludes whitebox expansion.
- Allow using meta expressions without expected type. This captures the main idea of whitebox expansion, in which compile-time metaprogramming has the final say about the type of transformed snippets of code.

Assimilate the most popular use cases. Instead of supporting the general notion of whiteboxity, we can introduce dedicated compiler support for the most popular uses cases including the `Generic` mechanism in Shapeless (subsection 6.1.4) and quasiquote patterns in scala.reflect (subsection 3.3.2).

This is an attempt at a compromise. On the one hand, this approach allows us to follow through with simplification. On the other hand, it significantly minimizes the impact on the existing users of whitebox macros.

The downside of this solution is that it requires an extensive design process (because it involves adding new language features to Scala) and assumes that the internalized techniques have reached their final form. If a new version of Shapeless or a new version of scala.reflect (read: scala.meta) decide to adapt their designs after these designs have been assimilated, they will have hard time doing that.

Dedicated support for type-level computations. Manifestations of whiteboxity are quite diverse, but a lot of them are reducible to type-level computations.

For example, let us take a macro `join` that takes two objects and outputs an object that has fields of both. Since Scala does not have row polymorphism, it is impossible to write a type signature for this macro, so we have to declare it as whitebox.

```
scala> def join(x: Any, y: Any): Any = macro ...
defined term macro join: (x: Any, y: Any)Any

scala> join(new { val x = 2 }, new { val y = 3 })
res0: AnyRef{val x: Int; val y: Int} = $anon$1@64af328d
```

Here we can see how the whitebox macro encapsulates a type-level computation that takes

the types of both arguments (`AnyRef{ val x: Int }` and `AnyRef{ val y: Int }`) and merges them into the result type. Since this computation does not involve the abstract syntax trees of the arguments, the whitebox part of the macro can be extracted into a helper, making the macro itself blackbox.

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

trait Join[T, U, V]
object Join {
  implicit def materialize[T, U, V]: Join[T, U, V] = macro ...
}

// Exiting paste mode, now interpreting.

defined trait Join
defined object Join

scala>  def join[T, U, V](x: T, y: U)
     | (implicit ev: Join[T, U, V]): V = macro ...
defined term macro join: [T, U, V](x: T, y: U)...
```

This approach can express some macros that refine their return type, all fundep materialization macros, and even some macros that dynamically compute availability of implicits (such macros can be modified to take an additional implicit parameter whose failure to materialize can be used to control availability of the enclosing implicit) - that is, all macros whose whiteboxity depends only on types of their prefix and arguments.

Now, after eligible whitebox macros are rewritten this way, we can replace the whitebox materializers that compute types, e.g. `Join.materialize`, with a dedicated language feature that natively expresses them, similarly to our ideas briefly outlined in section 6.3. The listing below provides a sketch of an imaginary syntax that assumes inline types and type-generating meta expressions, whose evaluation is left for future work.

```
inline type Join[T, U] = meta {
  import scala.meta._
  val jointVals = union(T, U)
  t"{ ..$jointVals }"
}

inline def join[T, U](x: T, y: U): Join[T, U] = meta { ... }
```

From the discussion above, we can see that whiteboxity is an important feature of the current macro system and is used in multiple highly popular open-source libraries. As a result, giving up whiteboxity may lead to undesired practical consequences.

Future work is needed to reconcile the design of the new macro system and existing use cases, and in this section we provided several approaches to addressing this need. In the opinion of the author, the approach that involves dedicated support to type-level computations is the most promising, because it both simplifies the macro system and provides support for the most important use cases of whiteboxity.

9.4 Meta APIs

In scala.meta, all APIs are available out of the box after doing `import scala.meta._`. However, in order to use most of them, metaprogrammers must have access to certain capabilities such as `Dialect` or `Mirror` (subsection 8.1.1).

Meta scopes provide three different capabilities. On the one hand, meta scopes provide a `Dialect` and a `Mirror` which enable usage of general-purpose metaprogramming facilities of scala.meta. On the other hand, there are expansion-specific facilities enabled via a brand new capability called `Expansion`, whose simplified listing is presented below.

```
@opaque
trait Expansion {
  def abort(origin: Origin, msg: String): Nothing
  def error(origin: Origin, msg: String): Unit
  def warning(origin: Origin, msg: String): Unit
  def hoist(term: Term): Term.Ref
}
```

Comparing the functionality provided by `Expansion` with macro APIs from the current macro system outlined in section 4.5, we can make several observations.

First, `Context.prefix` is no longer necessary, because meta expressions can refer to prefixes of enclosing inline definitions via `this` (subsection 9.3.3). `Context.macroApplication` is unnecessary as well, because meta expressions may be written outside inline vals and defs, which means that they will not have a corresponding application at all.

Secondly, much like their counterparts in the current macro system, meta APIs support diagnostic messages. Domain-specific error reporting is very frequently used in practice.

Thirdly, we no longer expose APIs that involve compiler internals. Most of these APIs take care of the limitations of scala.reflect, so in the new system based on scala.meta they are simply unnecessary. Others feature advanced functionality that manipulates internal compiler state that is beyond the scope of this dissertation.

Finally, we also support `hoist`, which takes a `scala.meta.Term`, precomputes it in a temporary variable outside the meta expansion and returns a reference to it. This functionality provides `inline`-compatible hoisting of prefixes and arguments.

9.5 Extensions

Much like the current macro system can get extended to support macro annotations (section 5.3), the new macro system can get extended to support new-style macro annotations that provide similar functionality.

```
1  import scala.annotation.StaticAnnotation
2
3  class h2db(url: String) extends StaticAnnotation {
4    inline def apply(defns: Any): Any = meta {
5      ...
6    }
7  }
8
9  object Test {
10   def main(args: Array[String]): Unit = {
11     @h2db("coffees") object Db
12     val brazilian = Db.Coffees.insert("Brazilian", 99.0)
13     Db.Coffees.update(brazilian.copy(price = 100.0))
14     println(Db.Coffees.all)
15   }
16 }
```

In the current macro system, a macro annotation is a subclass of `StaticAnnotation` that define a macro def called `macroTransform`. In the new design, a macro annotation is a subclass of `StaticAnnotation` that defines an inline def called `apply`.

Expansion of new-style macro annotations is very similar to expansion of vanilla macro annotations (subsection 5.3.2). The only difference is that we only provide syntactic APIs, informed about the limitations brought by exposing semantic APIs in macros that generate top-level definitions (subsection 5.3.3). Concretely, meta scopes in macro annotations expose a `Dialect` capability instead of a `Mirror`. As a result, we can afford to expand on enter (subsection 5.3.4) without any limitations to the shape of expansion.

9.6 Tool support

The key innovation of the new macro system is the fact that it is based on scala.meta, which finally makes it feasible to provide third-party implementations of mirrors.

As described in section 8.4, there exists a prototype implementation of an IntelliJ mirror. Moreover, thanks to a further effort of Mikhail Mutcianko from the IntelliJ team, this mirror powers an interactive expander of new-style macro annotations. This recent advancement that it was a right choice to bet on scala.meta to fix code comprehension (section 7.1) and error reporting (section 7.2) issues with the current macro system.

This takes some pressure off whitebox macros. In the current macro system, there is a preference towards blackbox macros, because they are much friendlier to IntelliJ. Once we have a fully-working mirror for IntelliJ, user experience of blackbox and whitebox macros should be equivalent. For additional discussion of whiteboxity from a language design and compiler development perspective, refer to subsection 9.3.5.

Improvements in support for incremental compilation (section 7.3), testing (section 7.4) and debugging (section 7.5) also hinge on a capability of scala.meta to enable custom mirror implementations. However, they also require significant new functionality to be developed in the corresponding tools (additional dependency tracking mechanisms for the incremental compiler and changes to the vanilla debugger in IDEs).

9.7 Conclusion

Pioneered by def macros (chapter 4) and macro annotations (section 5.3), the area of language-integrated compile-time metaprogramming in Scala has shown significant practical value.

During the last several years, we have been utilizing and maintaining the current macro system in industrial projects. Learning from this experience, we have created a new design based on `inline` and `meta` that provides comparable functionality and avoids the most common pitfalls of existing macros (section 9.2).

The user interface of the new system is a conservative evolution of def macros. Inline defs (subsection 9.3.1) work similarly to macro defs (section 4.3), and meta expressions (subsection 9.3.3) play the compile-time execution role of macro impls (section 4.3). These new language features are designed to be used together, and in this capacity their look and feel can hardly be distinguished from that of def macros.

The metaprogrammer interface of the new system is a drastic redesign. We have merged macro defs and macro impls, and, more importantly, we have switched the underlying metaprogramming API from scala.reflect (chapter 3) to scala.meta (chapter 8). This has allowed us to make significant improvements in robustness and tool support (section 9.6).

The main open question of the new design is whether whiteboxity will stay or will be removed from the new macro system (subsection 9.3.5). Future work also includes hygiene (section 8.3) and lifting the separate compilation restriction (subsection 4.3.3).

At the time of writing, `inline` and `meta` are partially implemented in a prototype compiler plugin for Scala 2.11 and an accompanying experimental branch of IntelliJ. We are excited with our initial results and are planning to further experiment with the new design, eventually aiming to submit it as a Scala improvement proposal.

10 Related work

C# was initially limited to runtime metaprogramming via reflection capabilities provided by the CLR, its underlying runtime [61]. Later on, thanks to the introduction of Microsoft LINQ [79], C# gained a capability to perform type-directed reification of lambda expressions into so-called expression trees. Expression trees can be inspected, reassembled and evaluated at runtime.

Expression trees were an inspiration for `Code` (section 2.3), which in turn inspired `reify`, scala.reflect's notation for abstract syntax trees (subsection 3.3.1).

Latest developments in C# metaprogramming happen within the recently released .NET Compiler Platform, better known by its codename "Roslyn" [95]. Roslyn exposes a wide range of functionality including syntactic and semantic analyses of C# and Visual Basic .NET code. Roslyn is quite popular as a foundation for developer tooling in C#, but it is unclear whether it will be used in the language. In recent discussions, the C# team seems to deliberately avoid language-integrated metaprogramming, preferring plugin-based alternatives [123].

Roslyn was a major influence on the design process of scala.meta. Both Roslyn and scala.meta were complete rewrites of their predecessors with comprehensiveness of the language model being one of the core goals, and that made the experience of the Roslyn team of particular interest to us. The idea to use tokens (section 8.2) in order to capture formatting and other syntactic details of Scala syntax comes directly from Roslyn.

C/C++ both include support for compile-time metaprogramming via the C preprocessor [100]. The preprocessor operates via string substitution, making it possible for metaprograms to produce syntactically invalid programs. Writing robust preprocessor macros and debugging macro expansions is notoriously hard.

C++ enhances metaprogramming capabilities of C with support for templates, recipes for generating similarly structured pieces of code [55, 1]. Templates are instantiated

at compile time and, through an interplay of a disparate set of language features not originally intended to support metaprogramming, can perform static computation, code specialization and type structure analysis.

Even though the design of Scala macros (chapter 4 and chapter 5) has little in common with the design of C++ templates, our tooling situation is quite similar. For both systems, tool integration was not on the list of the original design goals, so mistakes in metaprograms oftentime result in arcane error messages, metaprograms are hard to debug, some metaprograms are not portable between implementation, etc.

Recent versions of C++ also include support for compile-time function execution via `constexpr`. With this language feature, it becomes possible to evaluate expressions that belong to a subset of the language at compile time.

D originated as reengineering of C++, so it revisits and improves much of the functionality of its predecessor. Notably, D does not have a preprocessor and has a redesigned template system that uses dedicated language features (compile-time function execution, aliases, static ifs, traits, mixins and others) to provide a straightforward metaprogramming experience [28].

Inline/meta (chapter 9), our new macro system, is partially inspired by metaprogramming in D. Thanks to the combination of dedicated metaprogramming features, D can express some metaprograms in a markedly concise way. In comparison, the current macro system (chapter 4 and section 5.3) requires non-trivial amounts of boilerplate just to declare a macro. As a result, when designing the next version of Scala macros, we kept an eye on the possibilities to simplify metaprogrammer experience, and that led to creation of `inline`.

F# is one of the languages of the .NET platform, so, similarly to C#, it supports both runtime reflection provided by the CLR and type-directed expression reification required by LINQ. Unlike C#, F# is not supported by Roslyn, but instead it features its own F# Compiler Services infrastructure [45]. Additionally, F# supports several features that are unique to the .NET ecosystem.

First, F# implements untyped and typed quotations, which can construct and deconstruct abstract syntax trees from snippets of F# code with support for unquoting. Quotations enable language virtualization, which was successfully used for implementing LINQ, accelerating F# code on GPU and performing runtime native code generation [115, 20].

Secondly, F# supports type providers [116, 117], a compile-time metaprogramming facility that can generate top-level definitions with user-defined logic. This logic only has access to literal arguments passed to type provider invocations invocations, but they cannot ask the compiler about other definitions in the program. Code generation is performed lazily, and generated types can be optionally erased.

Quotations served as a blueprint for the original prototype of tree notations in scala.reflect (section 3.3). At some point, we were pondering over the distinction between untyped and typed quasiquotes, hesitant to introduce both, and experience reports from F# provided valuable insight. Type providers were the direct inspiration for type macros (section 5.2) and one of the main use cases for macro annotations (section 5.3).

Haskell supports compile-time metaprogramming via Template Haskell [108, 78], which provides syntax to call metaprograms during compilation and insert results into their callsites. Such metaprograms can use a dedicated reflection API to ask the compiler for information about types, definitions, etc. An interesting peculiarity of Template Haskell is that metaprograms run in a monad that encapsulates internal state involved in keeping track of static metadata and generation of fresh names.

Template Haskell supports quasiquotations, which have later been generalized to allow for user-defined implementations of arbitrary notations [77].

Moreover, Haskell has runtime metaprogramming facilities that evolved from "Scrap Your Boilerplate" [72], a seminal paper on generic programming [84]. Using built-in functionality, it is possible to derive instances of magic typeclasses that capture metadata about given types.

Being one of the classic designs in compile-time metaprogramming, Template Haskell significantly influenced our work on Scala macros. In particular, Haskell's customizable approach to quasiquotations inspired the design of extensible string interpolation [114], which we then used to build quasiquotes for scala.reflect and scala.meta (subsection 3.3.2 and section 8.3). `Liftable` in scala.reflect and `Lift` in scala.meta came from the original Template Haskell paper [108]. Certain parallels can also be drawn between `meta` in scala.meta and `$` in Template Haskell.

Haskell's take on generic programming, including the infrastructure around the `Generic` typeclass, served as an inspiration for similar functionality in Shapeless [98], and Shapeless was the original driving force behind our research in materialization (section 6.1).

Idris supports compile-time metaprogramming via error reflection and elaborator reflection that reify internal compiler data structures and allow custom metaprograms to run inside the compiler. Reflection API includes support for quasiquotation as well as the ability to query global and local elaboration contexts [23].

Additionally, there is an experimental extension to Idris that implements dependent type providers [22], which can run inside the Idris elaborator and return values of type `Type`. Thanks to the dependent nature of Idris, this process does not involve reflection and resulting values can immediately participate in typechecking. Even though dependent type providers can not introduce top-level definitions, they still allow many of the benefits of F#'s type providers.

Java provides only runtime metaprogramming facilities in its standard distribution, but there exist several research systems that enrich Java with support for compile-time metaprogramming. Of note are [62, 80], which introduce dedicated language features to support static reflection and code generation in a type-safe manner.

During the initial design phase of inline/meta (chapter 9), we experimented with the idea of extending the templating capabilities provided by `inline` from the expression level to the definition level using ideas from MorphJ [62]. We provide a corresponding design sketch, along with ideas for potential use cases in [6].

Lisp and its descendants have made a tremendous contribution to compile-time metaprogramming. Over fifty years of research went into development of macro systems in different Lisp dialects [60].

Syntactic macros in Lisp are functions that take syntax and return syntax. Upon encountering certain syntax patterns, Lisp compilers call these functions and use them to rewrite these patterns. Apart from constructing and deconstructing syntax objects, macros also have access to reflection APIs.

It is typical for the languages of the Lisp family to have a minimalistic set of language features, and then use macros to implement missing functionality in libraries. Macros have been successfully used to build such complex language features as pattern matching [119], classes, mixins, traits [53, 52] and type systems [120]. Moreover, Racket goes beyond just syntactic macros, and provides facilities to build languages as libraries [121, 46]. Using these facilities, a programmer may change all aspects of a language: lexicographic and parsed notation, the static semantics, module linking, and optimizations.

Such intense use of macros motivated research into idioms and technologies of effective macro writing, including quasiquotation [3, 70], hygiene [69, 70, 40, 24, 51, 50], phase separation [48, 49], tool integration [26, 47, 25], so Lisp macros are now the gold standard in macro systems.

In Scala, a language with an established type system and a wide range of already existing language features, we limited our initial design of the macro system (chapter 4) to well-formed and well-typed method applications. Being aware of the risks of language fragmentation associated with Lisp-style macro systems [125], we decided to contain the metaprogramming powers using types.

Our experience shows that there are practically important macros that go beyond the capabilities of the Scala type system (subsection 6.1.4, section 5.3), but nevertheless we are trying to restrict them too (subsection 9.3.5). There have also been experiments in adding parser macros [39] to test the waters, but so far parser macros have not yet found their adopters.

Of particular interest to Scala macros is Lisp's extensive body of literature on hygiene. We hope that adapting Racket's technologies will provide us with insight to address our hygiene issues (subsection 3.3.4). Recent advances in hygiene described in [50] are particularly inspiring, because they reformulate previous approaches in a simpler, yet similarly powerful manner.

MetaML was one of the first programming languages designed to support multi-staged programming. By a combination of three dedicated language constructs - brackets, escape and run - MetaML enables runtime code generation and compilation [118]. Requiring that code representation is statically typed and cannot be taken apart guarantees that well-typed programs always generate well-typed programs.

MacroML demonstrated that using brackets and escape from MetaML and collapsing the tower of stages to just compile time and runtime results in a generative macro system [54]. Even though this macro system does not support analytic macros or reflection APIs, it guarantees that well-typed macros always generate well-typed code.

Our early design of def macros (chapter 4) that used `Expr[T]` to represent parameters of macro impls (subsection 4.3.1) and `reify` (subsection 3.3.1) to combine exprs into bigger exprs was somewhat similar to MacroML. Even though we also exposed additional functionality via the scala.reflect API (section 4.5), the core that dealt with tree rewriting shared many characteristics with MacroML.

Our experience shows that the majority of practically important macros (chapter 6) can not be expressed within the limitations that come with the MacroML style of compile-time metaprogramming, so eventually we abandoned reify in favor of quasiquotes (subsection 3.3.2).

Nemerle is a hybrid object-oriented/functional programming language for the .NET platform. It is noticeably similar to C#, both syntactically and semantically, but it also adds new features like global type inference and pattern matching. Moreover, Nemerle introduces macros that can rewrite both expressions and definitions at compile time according to user-defined logic [111, 110].

Macros in Nemerle are top-level functions that take syntax and return syntax. Depending on they way how a macro is defined, it can be used like a normal function (in that case, it will perform expression rewriting) or like an attribute on definitions (in that case, it will rewrite definitions). Macros can use quasi-quotes and interact with the compiler, obtaining semantic information about their arguments and other definitions in the program. Additionally, specially-defined macros can modify the syntax of the language.

Nemerle was the direct inspiration for Scala macros (chapter 4 and chapter 5). Nemerle provides a success story of taking a mainstream statically-typed language with C-like

syntax and transparently enriching it with macro powers, and that experience was very important for legitimizing our ideas.

Many aspects of the Nemerle macro system, including transparent macro invocation syntax, the ability to constrain macro arguments with method signatures and the separate compilation restriction have been mirrored in our original design. Some others, for example having quasiquotes as an integral part of the reflection API and the ability to rewrite definitions, have been added over time.

Nevertheless, despite of heavily drawing inspiration from Nemerle, Scala macros were designed and evolved in a markedly unique way. First, we enforce well-formedness and well-typedness restrictions on parameters of def macros, relying on other DSL-friendly features of Scala to customize language syntax in a principled manner. Secondly, we further tighten the integration of macros into the language, allowing macros to be members of classes, traits, objects, etc. This makes it possible to transparently use macros in language features that desugar to method calls (section 4.6). Finally, we give significant attention to the reflection API, upgrading its status from a compiler API to a unified metaprogramming framework (chapter 3 and chapter 8).

Rust supports compile-time metaprogramming via macros that can change language syntax. Rust macros come in two flavors: 1) `macro_rules` [17] that can only use declarative syntax inspired by [70], 2) procedural macros [18] that can rewrite syntax trees or token trees into syntax trees using custom logic and internal compiler APIs.

Even though Rust is a statically-typed language, Rust macros deliberately expand before typechecking, so they do not have access to semantic information about the program.

Procedural macros in Rust are similar to Scala macros in the sense that they are both awkward to define, they both use an API derived from compiler internals, and they are both so useful that they are widely used regardless. There is an ongoing initiative in the Rust community to evolve procedural macros, introducing a more lightweight definition syntax and a new metaprogramming library to be used in procedural macros [19]. The circumstances and the high-level design goals of this initiative clearly resemble our own situation with the inline/meta proposal (section 9.3), which we find quite fascinating.

11 Conclusion

At the time of writing, scala.reflect and def macros have been an experimental part of the Scala compiler for about five years.

Our technologies have appeared in the right place at the right time, receiving enthusiastic feedback from the Scala community, which alowed us to perform an extensive practical evaluation of our ideas. Based on practical experience, we can now confidently pass judgment on the experiment with unified metaprogramming. Let us revisit the original thesis statement sentence by sentence.

> A metaprogramming framework that unifies compile-time and runtime reflection in a statically-typed programming language is feasible and useful...

Within about a year of effort, we were able to ship a production release of scala.reflect, in no small part because the desire for unification motivated us to reuse the already existing compile-time metaprogramming infrastructure provided by the Scala compiler. This confirms that unified metaprogramming is feasible.

The comprehensive nature of scala.reflect has earned it a place in the standard distribution and the status of one of the most influential open-source project in the Scala community. This shows that unified metaprogramming is useful.

Usefulness of unification comes from the fact that it promotes ontological correspondence of metaprogramming facilities and the underlying language [7], motivating the language model to correspond to the ontology of the language regardless of the environment.

However, as our experience shows, the degree to which ontological correspondence can be achieved is limited by the introspection capabilities of the environment. We observed the troubles brought by desugarings employed by the Scala compiler, type erasure used by the JVM and aggressive optimizations performed by Scala.js and Scala Native. Future research is needed to determine if and how these adverse effects can be compensated.

> ...A portable language model that naturally arises from unification fosters reuse of metaprogramming idioms across a multitude of environments...

Scala.reflect has delivered on its promise to provide a unified language model that can be used to write metaprograms targetting both compile time and runtime.

First, we created and shipped macros, compile-time metaprograms that run inside the Scala compiler using scala.reflect. Moreover, we introduced a way to do runtime reflection in terms of Scala features - again thanks to scala.reflect.

Both of these use cases utilize the same API and share the same techniques, e.g. using quasiquotes and type tags to represent abstract syntax trees and types. This demonstrates reuse of metaprogramming idioms across environments.

Again, depending on the environment, the array of available idioms may vary. We have seen how the language model of scala.reflect exhibits a split between trees and symbols, because compiled programs on the JVM do not carry original source code. In order to combat that, we have started experimenting with AST persistence, but since then Scala has gained new backends that can dramatically restructure definitions over the course of whole-program optimizations. It looks like metaprogrammability and performance are at odds with each other, and future work is needed to balance them.

> ...Such a common vocabulary of idioms can simplify and stimulate development of code analysis and generation tools.

Delivering a metaprogramming framework that exposes almost the entire surface of the language entailed a prolonged and focused engineering effort in an area that was practically important but was historically lacking attention. Almost inevitably, this resulted in significant innovation.

This innovative spirit most strongly manifested itself in Scala macros. Armed with a powerful vocabulary of metaprogramming idioms provided by scala.reflect, thousands of enthusiasts from the Scala community have rushed into exploring compile-time metaprogramming.

In addition to creating obvious value in the form of new techniques and libraries, this enthusiasm also indirectly stimulated bugfixes and subsequent feature developments in future versions of scala.reflect.

The demand in evolution was so high that it has even led to creation of scala.meta, an effort aimed at rebuilding the language model from scratch. This shows that unified metaprogramming not only allowed to solve new practical problems, but also created the foundation for future developments in the area.

We will now wrap up our account of unification that happened in Scala metaprogramming and its effects on the ecosystem. As promised in the introduction, we explained what worked, what did not and what directions were taken by further evolution of metaprogramming in Scala.

Concretely, our best idea was to expose a powerful language model, since that led to development of macros, a feature that has now became an integral part of the Scala ecosystem. Our worst idea was to expose compiler internals in an API, since that led to subpar metaprogrammer experience and inadequate tool support. Our plans for the future involve a new implementation-independent language model that is tuned for metaprogrammer convenience and integration with third-party tools.

In conclusion, we would like to thank the readers for following through the exciting history of explosive growth of Scala metaprogramming. We hope that our experiences and discussions will inspire future developments in other programming languages.

Bibliography

[1] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[2] N. Amin. *Dependent Object Types*. PhD thesis, École polytechnique fédérale de Lausanne, 2016.

[3] A. Bawden et al. Quasiquotation in lisp. In *PEPM*, pages 4–12. Citeseer, 1999.

[4] E. Béguet and E. Burmako. Traversal query language for scala. meta. Technical report, 2015.

[5] E. Béguet and M. Jonnalagedda. Accelerating parser combinators with macros. In *Proceedings of the Fifth Annual Scala Workshop*, pages 7–17. ACM, 2014.

[6] A. Biboudis and E. Burmako. Morphscala: safe class morphing with macros. In *Proceedings of the Fifth Annual Scala Workshop*, pages 18–22. ACM, 2014.

[7] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *ACM SIGPLAN Notices*, volume 39, pages 331–344. ACM, 2004.

[8] T. Brown. *Vampire methods for structural types*, 2013 (accessed 19 August 2016).

[9] T. Brown. *Getting a structural type with an anonymous class's methods from a macro*, 2013 (accessed 25 October 2016).

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

[11] E. Burmako and contributors. *Macro Paradise 1.0*.

[12] E. Burmako and contributors. *Macro Paradise 2.x: Empowers production Scala compiler with latest macro developments*.

[13] E. Burmako and contributors. *Macro Paradise 3.0: Implementation of new-style ("inline") macros for scalac*.

[14] E. Burmako and D. Shabalin. *Macrology 201*, 2014 (accessed 19 August 2016).

[15] E. Burmako, D. Shabalin, and contributors. *Scala.meta*.

[16] E. Burmako, D. Shabalin, and O. Olgashko. *Quasiquote documentation (scala.meta)*, 2016 (accessed 19 August 2016).

[17] N. Cameron. *Macros in Rust pt1*, 2015 (accessed 19 August 2016).

[18] N. Cameron. *Macros in Rust pt2*, 2015 (accessed 19 August 2016).

[19] N. Cameron. *Procedural macros*, 2016 (accessed 19 August 2016).

[20] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. *ACM SIGPLAN Notices*, 48(9):403–416, 2013.

[21] A. Cheptsov. *IntelliJ API to Build Scala Macros Support*, 2015 (accessed 19 August 2016).

[22] D. R. Christiansen. Dependent type providers. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, pages 25–34. ACM, 2013.

[23] D. R. Christiansen. *Practical Reflection and Metaprogramming for Dependent Types*. PhD thesis, InstituttetThe Department, Software and SystemsSoftware & Systems, Software Development GroupSoftware Development Group, 2016.

[24] W. Clinger and J. Rees. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162. ACM, 1991.

[25] R. Culpepper and M. Felleisen. Debugging hygienic macros. *Science of Computer Programming*, 75(7):496–515, 2010.

[26] R. Culpepper and M. Felleisen. Fortifying macros. In *ACM Sigplan Notices*, volume 45, pages 235–246. ACM, 2010.

[27] D Language Contributors. *D Programming Language Specification*.

[28] D Language Contributors. *Templates Revisited*.

[29] M. Demarne and E. Burmako. Style checking with scala.meta. Technical report, 2015.

[30] M. Demarne, A. Ghosn, and E. Burmako. Scala ast persistence. Technical report, 2014.

[31] T. Disney and contributors. *Massive hygiene simplification*.

[32] T. Disney, N. Faubion, D. Herman, and C. Flanagan. Sweeten your javascript: Hygienic macros for es5. In *ACM SIGPLAN Notices*, volume 50, pages 35–44. ACM, 2014.

[33] S. Doeraene. *Proof of concept of a "linker plugin" enabling some reflection for Scala.js*.

[34] S. Doeraene and contributors. *Scala.js*.

[35] G. Dubochet. *Embedded domain-specific languages using libraries and dynamic metaprogramming*. PhD thesis, École polytechnique fédérale de Lausanne, 2011.

[36] M. N. Duhem. *Status of support for macros in incremental compiler*.

[37] M. N. Duhem and E. Burmako. Making sbt macro-aware. Technical report, 2014.

[38] M. N. Duhem and E. Burmako. Macros in sbt: Problem solved! Technical report, 2015.

[39] M. N. Duhem and E. Burmako. Parser macros for scala. Technical report, 2015.

[40] R. K. Dybvig. Writing hygienic macros in scheme with syntax-case. 1992.

[41] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp and symbolic computation*, 5(4):295–326, 1993.

[42] École polytechnique fédérale de Lausanne. *Dotty: A next generation compiler for Scala*.

[43] École polytechnique fédérale de Lausanne and Lightbend Inc. *The Scala programming language*.

[44] Ensime Contributors. *ENSIME*.

[45] F# Language Contributors. *F# Compiler Services*.

[46] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. The racket manifesto. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[47] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A programming environment for scheme. *Journal of functional programming*, 12(02):159–182, 2002.

[48] M. Flatt. Composable and compilable macros:: you want it when? In *ACM SIGPLAN Notices*, volume 37, pages 72–83. ACM, 2002.

[49] M. Flatt. Submodules in racket: you want it when, again? In *ACM SIGPLAN Notices*, volume 49, pages 13–22. ACM, 2013.

[50] M. Flatt. Binding as sets of scopes. *ACM SIGPLAN Notices*, 51(1):705–717, 2016.

[51] M. Flatt, R. Culpepper, D. Darais, and R. B. Findler. Macros that work together. *Journal of Functional Programming*, 22(02):181–216, 2012.

[52] M. Flatt, R. B. Findler, and M. Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems*, pages 270–289. Springer, 2006.

[53] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM, 1998.

[54] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: type-safe, generative, binding macros in macroml. *ACM SIGPLAN Notices*, 36(10):74–85, 2001.

[55] R. Garcia and A. Lumsdaine. Toward foundations for type-reflective metaprogramming. In *ACM Sigplan Notices*, volume 45, pages 25–34. ACM, 2009.

[56] Ó. P. Geirsson. scalafmt: opinionated code formatter for scala. Master's thesis, École polytechnique fédérale de Lausanne, 2016.

[57] A. Ghosn and E. Burmako. Obey: Code health for scala.meta. Technical report, 2015.

[58] P. Haller and H. Miller. Ray: Integrating rx and async for direct-style reactive streams. In *Workshop on Reactivity, Events and Modularity*, number EPFL-CONF-188383, 2013.

[59] T. Hallgren. Fun with functional dependencies. In *Proc Joint CS/CE Winter Meeting, Chalmers Univerity, Varberg, Sweden*, 2001.

[60] T. P. Hart. Macro definitions for lisp. 1963.

[61] K. Hazzard and J. Bock. *Metaprogramming in. NET*. Manning Pub, 2013.

[62] S. S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. In *European Conference on Object-Oriented Programming*, pages 399–424. Springer, 2007.

[63] L. Hupel. Reducing boilerplate by generating type class instances automatically. Scala Workshop 2013, 2013.

[64] Information uhology – programming languages – c++. Standard, International Organization for Standardization, Dec. 2014.

[65] JetBrains. *Scala Plugin for IntelliJ IDEA*.

[66] V. Jovanovic. *Uniting Language Embeddings for Fast and Friendly DSLs*. PhD thesis, École polytechnique fédérale de Lausanne, 2016.

[67] V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, and M. Odersky. Yin-yang: concealing the deep embedding of dsls. In *Acm Sigplan Notices*, volume 50, pages 73–82. ACM, 2014.

[68] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM, 2004.

[69] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM, 1986.

[70] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 77–84. ACM, 1987.

[71] E. Kotelnikov. Type-directed language extension for effectful computations. In *Proceedings of the Fifth Annual Scala Workshop*, pages 35–43. ACM, 2014.

[72] R. Lämmel and S. P. Jones. *Scrap your boilerplate: a practical design pattern for generic programming*, volume 38. ACM, 2003.

[73] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. *Acm sigplan notices*, 33(10):36–44, 1998.

[74] Lightbend Inc. *sbt: the interactive build tool*.

[75] Lightbend Inc. and contributors. *Slick: Functional Relational Mapping for Scala*.

[76] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition*. Pearson Education, 2014.

[77] G. Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

[78] G. Mainland. Explicitly heterogeneous metaprogramming with metahaskell. In *ACM SIGPLAN Notices*, volume 47, pages 311–322. ACM, 2012.

[79] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[80] W. Miao and J. Siek. Compile-time reflection and metaprogramming for java. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, pages 27–37. ACM, 2014.

[81] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In *Acm Sigplan Notices*, volume 48, pages 183–202. ACM, 2013.

[82] H. Miller, P. Haller, and M. Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *European Conference on Object-Oriented Programming*, pages 308–333. Springer, 2014.

[83] H. Miller, P. Haller, B. C. d. S. Oliveira, and C. Bruno. Self-assembly: Lightweight language extension and datatype generic programming, all-in-one! Technical report, 2014.

[84] D. R. Musser and A. A. Stepanov. Generic programming. In *International Symposium on Symbolic and Algebraic Computation*, pages 13–25. Springer, 1988.

[85] M. Mutcianko. Static and dynamic type inference for compiled macros in scala. Specialist's Thesis (in Russian), ITMO University, 2014.

[86] M. Mutcianko. Compiler-agnostic model of macro expansion with dynamic type inference in scala. Master's Thesis (in Russian), ITMO University, 2016.

[87] D. Naydanov. Macro debugging for scala programming language. Specialist's Thesis (in Russian), Saint Petersburg University, 2013.

[88] M. Odersky, L. Spoon, and B. Venners. *Programming in scala*. Artima Inc, 2008.

[89] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *ACM Sigplan Notices*, volume 45, pages 341–360. ACM, 2010.

[90] F. R. R. Oliveira. Exploring the scala macro system for compile time model-based generation of statically type-safe rest services. 2015.

[91] Oracle Corporation. *Java Development Kit*.

[92] L. Parreaux. Semantic quasiquotes for domain-specific optimization. Technical report, 2016.

[93] D. Petrashko and contributors. *Dotty Linker*.

[94] H. Plociniczak. *Decrypting Local Type Inference*. PhD thesis, École polytechnique fédérale de Lausanne, 2016.

[95] Roslyn Contributors. *The .NET Compiler Platform ("Roslyn") provides open-source C# and Visual Basic compilers with rich code analysis APIs*.

[96] M. Sabin. *Implicit macros*, 2012 (accessed 19 August 2016).

[97] M. Sabin. *Shapeless Meets Implicit Macros*, 2013 (accessed 19 August 2016).

[98] M. Sabin and contributors. *Shapeless: Generic programming for Scala*.

[99] Scala IDE Contributors. *Scala IDE for Eclipse*.

[100] H. Schildt. *The annotated ANSI C Standard American National Standard for Programming Languages—C: ANSI/ISO 9899-1990*. Osborne/McGraw-Hill, 1990.

[101] D. Shabalin. Hygiene for scala. Master's thesis, École polytechnique fédérale de Lausanne, 2014.

[102] D. Shabalin. *A tutorial on how to embed subset of Joy into Scala via extensible string interpolation and macros*, 2014 (accessed 19 August 2016).

[103] D. Shabalin and E. Burmako. *Quasiquote documentation (scala.reflect)*, 2014 (accessed 19 August 2016).

[104] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for scala. Technical report, 2013.

[105] D. Shabalin and contributors. *Scala Native*.

[106] D. Shabalin and M. Odersky. Region-based off-heap memory for scala. Technical report, 2015.

[107] A. Shaikhha. An embedded query language in scala. Technical report, 2013.

[108] T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[109] A. Sherwany, N. Zaza, and N. Nystrom. A refactoring library for scala compiler extensions. In *International Conference on Compiler Construction*, pages 31–48. Springer, 2015.

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

[111] K. Skalski, M. Moskal, and P. Olszta. Meta-programming in nemerle. *Proceedings Generative Programming and Component Engineering*, 2004.

[112] L. Spoon. *Writing Scala Compiler Plugins*, 2009 (accessed 19 August 2016).

[113] N. Stucki and V. Ureche. Bridging islands of specialized code using macros and reified types. In *Proceedings of the 4th Workshop on Scala*, page 10. ACM, 2013.

[114] J. Suereth. *String Interpolation*, 2013 (accessed 19 August 2016).

[115] D. Syme. Leveraging. net meta-programming components from f#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*, pages 43–54. ACM, 2006.

[116] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, et al. Strongly-typed language support for internet-scale information sources. Technical report, 2012.

[117] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of the 2013 workshop on Data driven functional programming*, pages 1–4. ACM, 2013.

[118] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Notices*, volume 32, pages 203–217. ACM, 1997.

[119] S. Tobin-Hochstadt. Extensible pattern matching in an extensible language. *arXiv preprint arXiv:1106.2578*, 2011.

[120] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.

[121] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

[122] P. van der Walt. Reflection in agda. Master's thesis, Universiteit Utrecht, 2012.

[123] M. Warren. *[Proposal] enable code generating extensions to the compiler*, 2015 (accessed 19 August 2016).

[124] M. Weiel, I. Maier, S. Erdweg, M. Eichberg, and M. Mezini. Towards virtual traits in scala. In *Proceedings of the Fifth Annual Scala Workshop*, pages 67–75. ACM, 2014.

[125] R. Winestock. *The Lisp Curse*, 2011 (accessed 19 August 2016).

[126] H. Yamaguchi and S. Chiba. Inverse macro in scala. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 85–94. ACM, 2015.

Eugene Burmako
Minsk, Belarus
eugene.burmako@gmail.com

INTERESTS

Programming languages,
Compiler construction,
Developer tools,
Domain-specific languages.

HIGHLIGHTS

Designed and implemented reflection and macros for Scala together with Martin Odersky, enabling a number of new patterns used in popular community libraries: Akka, Play, Sbt, Scala.js, ScalaTest, Shapeless, Slick, Spark, Spire and others.

Contributed ∼750 commits, ∼750 tests and ∼150k lines of code to the Scala compiler, fixed ∼350 issues, being the 2nd top contributor to the Scala 2.10 release and the 3rd top contributor to the Scala 2.11 release.

Bootstrapped the Scala macros community from zero to self-sustaining, gave >20 talks about metaprogramming, managed >10 successful student projects, supervised the work of >40 contributors, provided help with >30 open-source projects.

PROJECTS

Scala (https://github.com/scala/scala) is a statically-typed object-functional programming language. Since Sep 2011, I'm a committer to scalac, the official Scala compiler. During my first two years, I was a very active contributor to the compiler before moving my research into separate repositories, but even now I enjoy hacking up an occasional typechecker trick.

- Contributed and supported a novel language feature: def macros, along with the underlying metaprogramming API.
- Implemented Macro Paradise, a popular compiler plugin with additional macro flavors, clocking 20k+ monthly downloads.
- Discovered and codified new tricks and idioms on the intersection of macros and vanilla Scala features together with prominent community members.
- Worked together with tool developers to enable support for the newly introduced language features and patterns.
- Fixed hundreds of issues in scalac, patching various subsystems of the compiler including the parser and the typechecker.

Scala Macros (http://scalamacros.org/). After joining the PhD program at EPFL in Sep 2011, I've come up with an idea to implement macros in Scala. I realized the initial vision in the official Scala compiler under the supervision of Martin Odersky and then proceeded to research and preach new flavors of macros, gradually gaining support in the community. Very soon, macros spread like wildfire, and now they are used in many popular libraries, including major components of the Lightbend platform.

Activities & contributions:

- Co-designed and implemented scala.reflect, the framework that has become the foundation for compile-time and runtime metaprogramming in Scala 2.10.
- Co-designed and implemented def macros for Scala 2.10.
- Designed and realized a number of other macro flavors (dynamic macros, interpolation macros, implicit macros, fundep materializers, type macros, type providers and macro annotations).
- Promoted, documented and maintained Scala macros, communicating with the adopters and contributors from the Scala community.
- Provided consulting and support for a number of production and research users, ramping up adoption of the new technologies.

Conferences & papers:

- A. Biboudis, E. Burmako "MorphScala: Safe Class Morphing with Macros", Scala Workshop, Uppsala, Jul 28-29, 2014
- E. Burmako, D. Shabalin "Macrology 201", flatMap, Oslo, May 12-13, 2014
- E. Burmako, T. Brown "Macro-Based Type Providers in Scala", Scalar, Warsaw, Apr 5, 2014
- E. Burmako, L. Hupel "Macros vs Types", NE Scala, New York, March 1-2, 2014
- E. Burmako "What Are Macros Good For?", Scala eXchange, London, December 2-3, 2013
- E. Burmako, D. Shabalin "Scala Macros", Google PhD Student Summit on Compiler & Programming Technology, Munich, November 25, 2013
- E. Burmako "Philosophy of Scala Macros", Strange Loop, St. Louis, September 18-19, 2013
- E. Burmako "What Are Macros Good For?", Scalapeño, Tel Aviv, July 17, 2013
- E. Burmako "Scala Macros: Let Our Powers Combine!", Scala Workshop, Montpellier, July 2, 2013
- E. Burmako "Applied Materialization", Bay Area Scala Enthusiasts, San Francisco, June 13, 2013
- E. Burmako "Half a Year in Macro Paradise", Scala Days, New York City, June 10-12, 2013
- E. Burmako "Macros in Scala", Codefest, Novosibirsk, March 30-31, 2013
- E. Burmako "Metaprogramming with Macros", Candidacy Exam, Lausanne, September 10, 2012
- E. Burmako, M. Odersky "Scala Macros, a Technical Report", Valentin Turchin Workshop on Metacomputation, Pereslavl-Zalessky, July 5-9, 2012
- E. Burmako "Metaprogramming in Scala 2.10", Minsk, April 28, 2012
- E. Burmako, J. C. Vogt "Scala Macros", Scala Days, London, April 17-18, 2012
- E. Burmako "Project Kepler: Compile-Time Metaprogramming for Scala", Lausanne, September 27, 2011

scala.meta (http://scalameta.org/). Inspired by the success of scala.reflect and learning from user feedback, in Oct 2013 I've set out to build a new metaprogramming platform for Scala. Together with my colleague Denys Shabalin, we've laid out the axioms: 1) user convenience comes first, 2) never lose a token or irreversibly desugar a tree, 3) always save typed ASTs into the resulting bytecode. 2 years and 2000+ commits later, scala.meta has a working language model and several production users.

Activities & contributions:

- Co-designed and realized lexical, syntactic and semantic models for Scala, tested them out in practice using the experience of working on scalac.
- Co-designed and implemented other aspects of the scala.meta vision (lightweight and portable reflection API, quasiquoting and hygiene, AST interpreter, AST persistence, integration with build tools, IDEs, etc).
- Closely worked with tool developers from the community to make sure that the new metaprogramming platform adequately addresses their needs.
- Planned and managed the timeline of the project, organized 20+ open-source contributors and their contributions.

Conferences & papers:

- E. Burmako "Metaprogramming 2.0", Scala Days, Berlin, Jun 17, 2016
- E. Burmako "Metaprogramming 2.0", Scala Days, New York City, May 11, 2016
- E. Burmako "What Did We Learn In Scala Meta?", ScalaSphere DevTools Summit, Kraków, Feb 11, 2016
- E. Burmako "State of the Meta, Fall 2015", fby.by, Minsk, Nov 28, 2015
- E. Burmako "Hands on Scala.Meta", Scala World, Lake District, Sep 21, 2015
- E. Burmako "State of the Meta, Summer 2015", Scala Days, Amsterdam, Jun 09, 2015
- E. Burmako "State of the Meta, Spring 2015", Scala Days, San Francisco, Mar 17, 2015
- E. Burmako "State of the Meta, Fall 2014", fby.by, Minsk, Nov 22, 2014
- E. Burmako, D. Shabalin "Easy Metaprogramming For Everyone!", Scala Days, Berlin, Jun 16-18, 2014
- E. Burmako "Rethinking Scala Macros", NE Scala, New York, March 1-2, 2014

Pickling (https://github.com/scala/pickling) is a native persistence framework for Scala that uses implicits and macros to derive extremely fast and customizable serializers, matching and surpassing the performance of Java frameworks. Pickling has gained popularity and is now included in sbt, the main build tool in the Scala ecosystem.

Activities & contributions:

- Co-designed object-oriented pickler combinators.
- Implemented and optimized the initial version of compile-time pickler generation facilities that stood the test of time for more than two years.

Conferences & papers:

- H. Miller, P. Haller, E. Burmako, M. Odersky. "Object-oriented pickler combinators and an extensible generation framework", Object-Oriented Programming, Systems, Languages & Applications, Indianapolis, October 26-31, 2013

Slick (https://github.com/slick/slick) is a modern database query and access library for Scala, which allows to work with stored data as if it were Scala collections.

- Took part in development of a LINQ-like direct embedding in fall/winter 2011, one of the Slick front-ends that provides language-integrated queries via macros.

Conflux (http://code.google.com/p/conflux/) is a parallel programming platform for C# that I developed in 2010-2011. Conflux features a domain-specific language and a high-level programming model that transparently distribute computational algorithms to GPUs both in single-machine and cluster environments:

Activities & contributions:

- Designed and developed a high-level programming model in the form of an embedded DSL for C#.
- Implemented a C# frontend: a C# AST and a decompiler library.
- Implemented a CUDA backend: a code generator for the PTX assembler of NVIDIA graphics processors and a JIT compilation library.
- Implemented a CPU backend: a code generator for multicore CPUs.
- Provided integration with the Visual Studio debugger.

Conferences & papers:

- E. Burmako, R. Sadykhov "Conflux: Embedding Massively Parallel Semantics in a High-Level Programming Language", Pattern Recognition and Information Processing, Minsk, 2011
- E. Burmako "Conflux: GPGPU for .NET", 1st International Conference, Application Developer Days, Yaroslavl, 2010
- A. Varanovich, V. Tulchinsky, E. Burmako, V. Falfushinsky, R. Sadykhov "Automated Software Development in Heterogeneous GPU/CPU Environments for Seismic Modeling", EAGE, Barcelona, 2010
- E. Burmako, R. Sadykhov "Compilation of high-level programming languages for execution on graphical processor units", Supercomputer Systems and Applications, Minsk, 2010
- A. Varanovich, E. Burmako, R. Sadykhov "Implementation of CUDA kernels for execution in multiprocessor systems with the use of dynamic runtime", Technologies of high-performance computing and computer-aided modeling, St. Petersburg, 2010

Relinq (http://code.google.com/p/relinq/) is a framework for transforming C# expression trees into JavaScript code and vice versa. Relinq can be used to implement such scenarios as: querying LINQ data sources from JavaScript, invoking cross-process LINQ queries and constructing LINQ queries dynamically.

- Designed and developed a JavaScript-based query language.
- Extracted and documented a coherent subset of the C# 3.0 specification that suffices for the domain.
- Implemented an expression compiler for the aforementioned subset of C# that supports generics, lambdas and local type inference.

Truesight (http://code.google.com/p/truesight-lite/) is a decompiler for the bytecode of the .NET virtual machine. The library provides intermediate representation for both low-level (bytecode) and high-level (abstract syntax tree) views of code. Truesight supports basic constructs of C# and is capable of decompiling structured control flow: conditionals, loops, and restricted gotos (i.e. returns, breaks and continues).

- Implemented a parser for CIL that reifies the bytecode and preserves debug information.
- Designed and documented an intermediate representation for decompiled code.
- Implemented visualization for the IR: graphviz dumps, integration with the Visual Studio debugger.
- Created and implemented an algorithm for decompilation of structured programs.

Tiller (http://code.google.com/p/elf4b/) is part of a C# software suite for designing and generating WYSIWYG interactive reports. It features a programming language that can be used to specify formulae and to script report generation process. The programming language comes with a JIT compiler that speeds up execution:

- Designed and developed a script language.
- Implemented a JIT compiler for the language.
- Implemented a thread-safe ZIP-based engine for storing report components.

EMPLOYMENT

Twitter
Senior Software Engineer
Nov 2016 - future
Activities: At Twitter, I will be continuing the work on my open-source Scala projects (scala.meta, macros, macro paradise, etc) to make Scala tooling better - both internally inside the company and publicly in the Scala community.

École Polytechnique Fédérale de Lausanne
Doctoral Assistant
Sep 2011 - Nov 2016
Job duties: Doing a PhD at EPFL, on the academic side of the Scala programming language team. Along with the open-source work on putting my research in practice into Scala, I'm also writing occasional papers and doing teaching for the faculty.

ScienceSoft
Software Engineer
Nov 2007 - Aug 2011
Job duties: Enterprise software development with C# and a little bit of JavaScript, with integration into Microsoft Office and Microsoft SharePoint.

Itransition
Software Engineer
Oct 2005 - Sep 2007
Job duties: Enterprise software development with C#.

Omega Software
Software Engineer
Oct 2004 - Sep 2005
Job duties: Enterprise software development with C++.

EDUCATION

École Polytechnique Fédérale de Lausanne
Sep 2011 - Nov 2016
Field of study: Computer science
Degree: PhD dissertation defended on 27 Sep 2016, PhD degree pending

United Institute of Informatics Problems
Nov 2009 - Oct 2012
Field of study: Computer science
Degree: Researcher

Belarusian State University
Sep 2003 - Jun 2008
Field of study: Computer science
Degree: Specialist