A Refactoring Library for Scala Compiler Extensions

Amanj Sherwany, Nosheen Zaza, and Nathaniel Nystrom

Università della Svizzera italiana (USI), Faculty of Informatics, Lugano, Switzerland
{amanj.sherwany,nosheen.zaza,nate.nystrom}@usi.ch

Abstract. Compiler plugins enable languages to be extended with new functionality by adding compiler passes that perform additional static checking, code generation, or code transformations. However, compiler plugins are often difficult to build. A plugin can perform arbitrary code transformations, easily allowing a developer to generate incorrect code. Moreover, the base compiler assumes many complex, sometimes undocumented invariants, requiring plugin developers to acquire intimate knowledge of the design and implementation of the compiler. To address these issues in the context of the Scala compiler plugin framework, we introduce Piuma. Piuma is a library that provides, first, an API to perform many common refactoring tasks needed by plugin writers, and, second, a DSL to eliminate much of the boilerplate code required for plugin development. We demonstrate the usefulness of our library by implementing five diverse compiler plugins. We show that, using Piuma, plugins require less code and are easier to understand than plugins developed using the base Scala compiler plugin API.

Keywords: Scala, compiler extensions, refactoring.

1 Introduction

To build complex applications more easily and efficiently, developers often use *domain-specific languages* (DSLs) [4, 22]. These special-purpose languages have abstractions tailored for specific problem domains. Often, DSLs are implemented on top of a general-purpose programming language like Scala [33], Ruby [15], Haskell [34], or Java [18]. This approach has the advantage that the DSL can reuse all the existing tools and infrastructure of the host language, such as IDEs, profilers, and debuggers. However, building a DSL on top of a general-purpose host language requires the host language to be easily extensible, allowing changes to both its semantics and syntax.

The Scala programming language [33] is a multi-paradigm language implemented on the Java virtual machine, that supporting both object-oriented and functional programming features. The language provides an expressive static type system and supports extending the syntax with new operators.

Moreover, like other recent languages including Java [18], C♯ [16, 21], and X10 [9], Scala provides an API to extend its compiler. Developers write *compiler plugins* to add new passes to the base Scala compiler. Compiler plugins are a useful tool in DSL development because they allow passes to, for instance, perform additional static analysis, to instrument code with additional dynamic checks, to perform optimizations, or to generate code for new language features.

Plugin developers can make nearly arbitrary changes to the base compiler, permitting implementation of complex language features, but unfortunately also permitting plugins to violate invariants assumed by the compiler. Breaking these often undocumented invariants may cause the compiler to generate incorrect Java bytecode or even to crash.

There are many ways to generate malformed bytecode using Scala compiler plugins. For example, a compiler plugin can add a "ghost" field to a class that can be seen by the Java VM when running the code, but not by the Scala compiler itself when importing the generated bytecode. This problem occurs because the Scala compiler embeds Scala-specific type information into the generated Java bytecode. If a plugin where to add a field but omit this extra information, another instance of the Scala compiler would not even see the field even though the field is present in the Java bytecode. This can result in other compilation units failing to compile correctly.

Since plugins add passes to the Scala compiler, running a plugin at the wrong point in the compilation can also allow bad code to be generated. For instance, a plugin could rename a field to have the same name as an existing field. If the plugin did this after the Scala type-checker runs, the error would not be detected and the bytecode would contain a duplicate field.

Based on an evaluation of several existing Scala compiler plugins, we developed Piuma, a refactoring library for the Scala compiler that enables easy implementation of correct compiler plugins. The library provides a set of refactoring methods commonly needed in compiler plugins. These methods are used, for instance, to safely rename definitions, to add new class, trait, or object members, and to extract expressions into methods. The library also provides a DSL for generating the boilerplate code necessary for writing Scala compiler plugins.

The rest of the paper is organized as follows: Section 2 introduces Piuma and motivates its usefulness. Section 3 covers the Piuma DSL. In Section 4, we demonstrate the design and usage of Piuma's refactoring libraries through code examples and use cases. We evaluate the library in Section 5 using a set of five case studies. Related work is discussed in Section 6. Finally, Section 7 concludes with a discussion of future work.

2 Overview

Scala offers a rich compiler API. However, because this API was designed primarily to implement the Scala compiler, merely exposing it to plugin developers does not provide the high-level abstractions for performing many of the common tasks in compiler plugins. For instance, a plugin might need to add a parameter to a method, to extract code into a method, or to rename a field. Performing these tasks with the Scala compiler plugin API requires the developer to implement complex AST transformations and to manage auxiliary data associated with the ASTs. Furthermore, exposing the entire compiler API permits programmers to perform potentially unsafe operations that can lead to exceptions during compilation, or worse, can generate malformed bytecode.

These shortcomings were the main motivation for Piuma, a refactoring framework for Scala compiler plugins. Piuma is composed of two components, a DSL that facilitates defining a compiler plugin without tedious boilerplate code, and a rich library of refactoring utilities. In the remainder of this section, we describe these two components and motivate their usefulness in more detail.

```
// define a new extension
class Example(val global: Global) extends Plugin {
  val components = List[PluginComponent](new Phase1(this))
}

// define a compilation phase
class Phase1(val plugin: Example) extends PluginComponent
  with Transform
  with TypingTransformers {

  val global: plugin.global.type = plugin.global
  override val runsRightAfter = Some("typer")
  val runsAfter = List[String]("typer")
  override val runsBefore = List[String]("patmat")
  val phaseName = "example"

  def newTransformer(unit: CompilationUnit) =
    new Phase1Transformer(unit)

  class Phase1Transformer(unit: CompilationUnit)
    extends TypingTransformer(unit) {

    override def transform(tree: Tree): Tree =
      super.transform(tree)
  }
}
```

Fig. 1. A simple Scala compiler extension

2.1 The Piuma DSL

The Scala compiler gives plugins access to nearly all features of the base Scala compiler, and gives them the flexibility to extend the semantics of the Scala language almost arbitrarily. Plugins can create and manipulate ASTs and symbols, extend the type system, and generate code. The design of the compiler follows the "cake pattern" [48], which allows both *datatype extension* by adding new ASTs, and *procedural extension* by adding more operations to existing AST nodes. However, this flexibility comes at the cost of ease-of-use and safety. Even simple extensions require a lot of complex boilerplate code. As an example, Fig. 1 shows the minimum setup required to create an extension with a single phase that does nothing more than traversing the AST.

To better understand how plugins are used and implemented, we performed a small survey of several publicly available Scala compiler plugins, including Avro [45], Scala-Dyno [3], Miniboxing [46], Uniqueness [19], and Continuations [35]. We found that none of these plugins added new types of AST nodes, while all added new functionality to existing AST node types. Based on this survey, we conclude that only procedural extension is needed for the majority of Scala compiler plugins. Adding AST node types is rarely necessary due to Scala's already flexible syntax.

```
// define a new extension
@plugin(Phase1)
class Example

// define a compilation phase
@treeTransformer("example")
class Phase1 {
  rightAfter("typer")
  before(List("patmat"))

  def transform(tree: Tree): Tree = super.transform(tree)
}
```

Fig. 2. Piuma DSL version of the simple compiler extension from Fig. 1

We have designed a macro-based DSL for implementing such extensions. Fig. 2 demonstrates how the same simple plugin as shown in Fig. 1 is implemented using this DSL. The DSL program defines a new plugin `Example` with a single phase `Phase1`. The phase performs a tree transformation, in this case the identity transformation. It runs immediately after the Scala typer and before the Scala compiler's pattern matcher phase. In Section 3 we describe this DSL in detail, using this example and others.

2.2 The Piuma Library

Suppose a developer is writing a plugin that performs partial evaluation [17, 23]. In the implementation, she creates specialized versions of a method, adding them into the class's companion object.[1] If the companion object does not exist, the plugin introduces one. This can be a tedious task, as shown in Fig. 3.

Every AST node in Scala has a `Symbol` attached to it, which includes type and other information about the node. The code first introduces a symbol for the companion object, called a "module" in the Scala compiler API. The code then initializes the symbol with its owner (its containing package or class), its type, and its parents (its supertypes). It then updates the owner of the method we want to insert into the module (code elided). It introduces a default constructor that calls the constructor of the new object's superclass, `AnyRef`, and appends the new constructor to the body of the module. Finally, it types the module tree. Failing to do any of these steps may lead to an exception during compilation, or worse, the compiler might silently generate malformed bytecode. With Piuma we can introduce a companion object in just a few lines of code, as shown in Fig. 4. The library ensures the object has a correct constructor and handles the symbol management for the new object and the method added to it, ensuring that other phases of the compiler can correctly access the object and method. In Section 4 we describe the design of the Piuma utilities.

[1] Scala, unlike Java, does not support static fields or methods. Instead, each class has a *companion object*, a singleton object with the same name as the class that contains the "static" members for the class.

```
// clazz: the symbol of the class for which we
//        want to create a companion object
// mthd:  a method we want to include in the object
val moduleName = clazz.name
val owner = clazz.owner
val moduleSymbol = clazz.newModule(moduleName.toTermName,
                            clazz.pos.focus, Flags.MODULE)
val moduleClassSymbol = moduleSymbol.moduleClass
moduleSymbol.owner = owner
moduleClassSymbol.owner = owner

val parents = List(Ident(definitions.AnyRefClass))
val moduleType = ClassInfoType(parents.map(_.symbol.tpe),
                          newScope, moduleClassSymbol)
moduleClassSymbol setInfo moduleType
moduleSymbol setInfoAndEnter moduleClassSymbol.tpe

// elided code: plugin writer needs to fix the owner
// of mthd and its children

val constSymbol =
  moduleClassSymbol.newClassConstructor(moduleSymbol.pos.focus)

constSymbol.setInfoAndEnter(MethodType(Nil, moduleSymbol.info)

val superCall = Select(Super(This(tpnme.EMPTY), tpnme.EMPTY),
                       nme.CONSTRUCTOR)
val rhs = Block(List(Apply(superCall, Nil)),
              Literal(Constant(())))
val constructor = DefDef(constSymbol, List(Nil), rhs)

localTyper.typed {
  ModuleDef(moduleSymbol, Template(parents, noSelfType,
                     List(cnstrct, mthd)))
}
```

Fig. 3. This listing shows how a new companion object with a single method is created for a class using the Scala compiler API

```
// clazz: the symbol of the class for which we
//        want to create a companion object
// mthd:  a method we want to include in the object
val module0 = clazz.mkCompanionObject
val module = module0.addMember(mthd)
```

Fig. 4. This listing shows how a new companion object with a single method is created for a class using the Piuma DSL

3 The Piuma DSL

The Piuma DSL extends Scala with features for defining compiler plugins and their components. In this section, we explain the design and use of this DSL in detail. We start by describing the general structure of a Scala compiler plugin and then describe the DSL constructs and the functionality they provide. Finally, we briefly describe how the DSL is implemented.

The Scala compiler consists of a sequence of compilation phases. Developers extend the compiler by creating plugins, composed of one or more phases inserted into this sequence. Compiler plugins are implemented by extending the `Plugin` class and providing a list of `PluginComponent`. Each of these components specifies a compiler phase and where it occurs in the compilation sequence. They also provide factory methods for creating the tree and symbol transformers that implement the phase.

The Piuma DSL extends Scala with four class annotations: `@plugin`, `@checker`, `@treeTransformer`, and `@infoTransformer`. The `@plugin` annotation generates boilerplate code for a compiler plugin itself. The other annotations generate boilerplate code for different `PluginComponent` implementations: `@checker` generates a type-checking component, `@treeTransformer` generates an AST-transforming component, and `@infoTransformer` generates a type-transforming component. Since the Scala compiler requires plugins and phases to be concrete classes, these annotations cannot appear on traits, abstract classes, or singleton objects. Annotated classes may still implement other traits. Fig. 5 shows the syntax of a compiler extension in the DSL.

The Piuma DSL is implemented using Scala's *annotation macros* [8]. For each annotation, macro expansion modifies the annotated class to extend a corresponding class from the Scala compiler API. It then mixes-in appropriate Piuma traits to facilitate access to the Piuma library.

3.1 The `@plugin` Annotation

The `@plugin` annotation is placed on a class that implements a compiler plugin. After macro expansion, the annotated class automatically extends the Scala class `Plugin`. The list of components provided by the plugin are specified as annotation arguments. Optionally, the class may provide a short description of its purpose, used when generating command-line usage information for the compiler.

3.2 Component Annotations

The three annotations `@checker`, `@treeTransformer`, and `@infoTransformer` are used to annotate classes that implement plugin components. Macro expansion generates boilerplate code in the annotated class for inserting the phase into the execution order. These annotations also specify the name of the phase using an annotation parameter.

In the body of a class annotated with one of the phase annotations, the programmer can optionally specify the class of the compiler plugin itself using the syntax `plugin` *PluginClass*. This introduces a field of the appropriate type into the class that refers to the plugin object. This field can be used to share information across the plugin's various compiler phases.

```
@plugin(MyChecker, MyTransformer, MyInfoTransformer)
class MyPlugin {
  describe("short␣description")
  ...
}

@checker("my_checker")
class MyChecker {
  plugin MyPlugin // optional

  after(List("phase1", "phase2", ...))  // optional
  rightAfter("phase1")                  // optional
  before(List("phase1", "phase2", ...)) // optional

  def check(unit: CompilationUnit): Unit = ...
  ...
}

@treeTransformer("my_transformer")
class MyTransformer {
  plugin MyPlugin // optional

  after(List("phase1", "phase2", ...))  // optional
  rightAfter("phase1")                  // optional
  before(List("phase1", "phase2", ...)) // optional

  def transform(tree: Tree): Tree = ...
  ...
}

@infoTransformer("my_info_transformer")
class MyInfoTransformer {
  plugin MyPlugin // optional

  after(List("phase1", "phase2", ...))  // optional
  rightAfter("phase1")                  // optional
  before(List("phase1", "phase2", ...)) // optional

  def transform(tree: Tree): Tree = ...
  def transformInfo(sym: Symbol, tpe: Type): Type = ...
  ...
}
```

Fig. 5. Syntax of the Piuma DSL

`@checker`. This annotation is placed on classes that implement type-checking phases. A checker phase cannot perform AST transformations but can perform static analysis of a compilation unit. The class must implement a method with the signature: `check(CompilationUnit): Unit`. After macro expansion, the class extends the `PluginComponent` class from the Scala compiler API and implements a factory method for creating compiler phase objects that invoke the `check` method for each compilation unit.

`@treeTransformer`. The `@treeTransformer` annotation is placed on component classes that implement AST transformations. Annotated classes must implement a method with the signature: `transform(Tree): Tree`. After expansion, the class extends `PluginComponent with TypingTransform`. The expanded class creates a `TreeTransformer` that traverses the AST and invokes the provided `transform` method at each node.

`@infoTransformer`. The last annotation is `@infoTransformer`, which is placed on classes that transform types in the AST. The annotation is similar to `@treeTransformer`; however, classes must provide not only a `transform(Tree): Tree` method, but also a `transformInfo(Symbol, Type): Type` method. After expansion, an annotated class will extend `PluginComponent with InfoTransform`. A generated `InfoTransformer` class traverses the AST and invokes the provided `transform` and `transformInfo` methods for each node and symbol encountered.

4 The Piuma Library

Piuma offers a rich set of utilities for generating and refactoring Scala compiler ASTs. In general, these methods implement common refactoring and creation patterns when writing compiler plugins. They are implemented using the AST generators of the Scala compiler API, and Scala's quasiquotes, which are a notation that permits generating trees directly from code snippets [39]. We aimed to provide a library that is easy to use, yet flexible and expressive. Library users can use the Scala compiler API alongside library code if they need lower-level access to the compiler internals.

In this section, we discuss the design of the refactoring library and demonstrate its use through use cases and code examples. The library is divided into four main categories, as shown in Fig. 6. The reader can refer to Piuma's project page[2] for documentation and more examples.

4.1 Tree Extractors

Methods in the *tree extractor* category permit the selection of a sequence of ASTs to be placed in another, compatible class, object, trait, method or variable. Since Scala ASTs are immutable, extractor operations generate new ASTs and do not affect the original tree to which they are applied.

[2] `https://github.com/amanjpro/piuma`

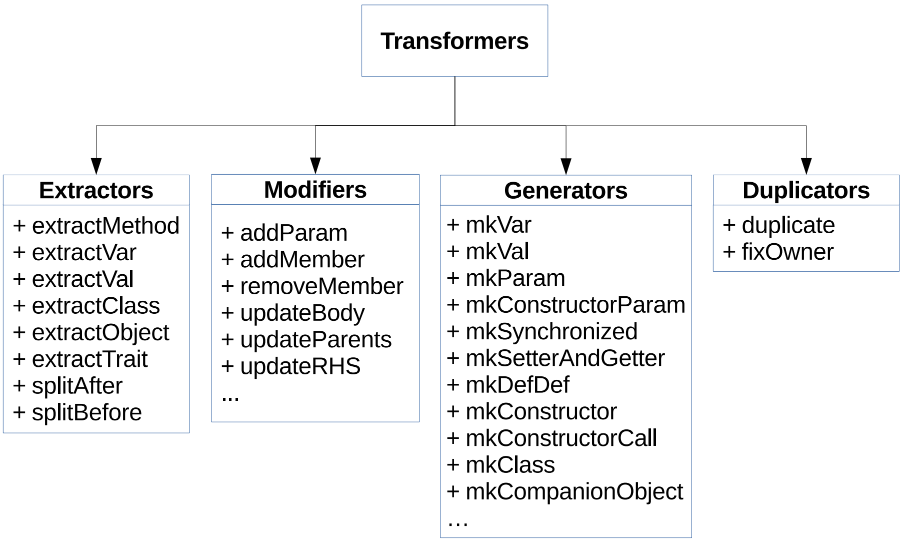

Fig. 6. The high-level design of Piuma's Library

Fig. 7(a) shows a code snippet that demonstrates a refactoring using a pair of tree extractor functions. Fig. 7(b) and (c) show the effect of applying the refactoring to a small method. The extractor function `splitAfter` scans the AST of the body of the original method `foo` in Fig. 7(b). When `splitAfter` finds the first occurrence of an AST node that satisfies a given predicate and returns the AST before and include the node and the AST after the node. In the example, the predicate matches a call to the `splitMe` method, specified using the Scala quasiquotes library [39]. Thus `splitAfter` returns the AST for the body of `foo` up to and including the call to `splitMe`, and another AST for the rest of the body.

It is possible to insert either or both of the ASTs returned by `splitAfter` into ASTs. In Fig. 7(c), `suffix` becomes the body of a new method `bar`. This is done by calling `extractMethod`, which takes a list of trees and a new method name as parameters. The new name must be unique in its scope, otherwise an exception is thrown.

The `extractMethod` operation handles free variables in the given method's body by adding parameters to the extracted method. That is, all free variables in `suffix` become parameters of `bar`. `extractMethod` also creates the AST of the new method and types it, and generates a call to the extracted method that can be substituted into the original method, as described in the next section. Piuma also creates a symbol for the new method, rebinding symbols of the extracted trees to their new owner, and placing them in the symbol table of the method owner. Without Piuma, the programmer would have to perform all these operations manually using the Scala compiler API.

```
// orig: the original method
// body: the body of orig

// Split body after the call to splitMe(),
val (prefix, suffix) = body.splitAfter((x:Tree) => x == q"splitMe()")

// Extract the code after the split into method bar
val (extracted, apply) = extractMethod(
    suffix,      // The body of the extracted method
    "bar"        // the name of the extracted method
    orig.symbol // the symbol of the original method
    ).get

// Replace the extracted code with a call to the new method
orig.updateRHS(Block(prefix, apply))
```

(a) Refactoring using Piuma tree extractor and tree modifier utilities

```
def foo(a: Int, b: String,
        c: Int): Int = {

  println(a)
  val d = c
  splitMe()
  println(d)
  a + b.size
}
```

(b) Before applying the refactoring in (a)

```
def foo(a: Int, b: String,
        c: Int): Int = {
  println(a)
  val d = c
  splitMe()
  bar(a, b, d)
}
def bar(a: Int, b: Int,
        d: Int): Int = {
  println(d)
  a + b.size
}
```

(c) After applying the refactoring in (a)

Fig. 7. A short example of refactoring with Piuma

Other useful Piuma extractors allow member extraction. It is possible, for instance, to extract an inner class and make it an outer class (using `extractClass`), to convert a local variable to a field (using `extractVar` or `extractVal`), or to move fields or methods across classes.

4.2 Tree Modifiers

Tree modifier utilities change AST nodes, for instance, adding or removing method parameters (`addParam` and `removeParam`, respectively), modifying a class or trait's body (`updateBody`, `addMember`, and `removeMember`) or its supertypes (`updateParents`). The `rename` method is used to change the name of a class, method, or variable. When a field is renamed, Piuma handles renaming its setters and getters as well. The tree modifiers also support updating method bodies (`updateRHS`). As an example, we can modify method `foo` from Fig. 7(b) to call the extracted method, as shown in Fig. 7(c). We do this with the call to `updateRHS`, as shown in Fig. 7(a).

```
// orig: the method to be specialized
// p:    the parameter we want to specialize
// v:    the value that we want to specialize p with
val specialized = orig.duplicate("new_name").removeParam(p, v)
```

Fig. 8. A simple example showing how a tree duplicator works

4.3 Tree Duplicators

Tree duplicators are mainly used to implement other Piuma functionalities, but can also be useful when writing plugins. For example, if a programmer needs to specialize a method, they can duplicate it first, then remove the specialized parameter from the duplicated tree, and substitute it with a value, as shown in Fig. 8.

Piuma's `duplicate` method differs from the compiler API's `duplicate` method in that it handles the creation of a new symbol for the new tree, and changes the binding of the node's children's symbols from their original owner to the duplicate.

The `fixOwner` method traverses an AST that has been duplicated and inserted into the AST at another location. It rebinds symbols in the traversed AST to change their owner (i.e., containing method, class, etc.) to the symbol at the new location in the AST.

4.4 Tree Generators

Piuma offers various AST generators that are simpler to use than the Scala compiler API tree generators. For instance, they handle setting the required flags that distinguish trees used to represent more than one syntactic construct (e.g., `var` and `val`, or class and trait trees). Symbol creation and binding is also handled for a generated AST, including all its descendant ASTs. Piuma generators facilitate other tasks that require multiple setup steps, such as surrounding the body of a method with a synchronization block, creating constructor parameters and constructor calls, and others.

5 Evaluation

To evaluate the design of Piuma, we performed a case study by reimplementing five Scala compiler plugins. In our selection, we considered plugins with varying purposes and which use a wide range of Scala compiler API functions. Below, we describe the original plugin and how Piuma is used to implement the plugin. We discuss the ease of implementation and compare the sizes of the Piuma version of the plugin with the original. All the plugins can be found at the Piuma project page. The five plugins and our results are summarized in Table 1. We used CLOC [10] to measure the number of lines in each plugin, ignoring comments and blank lines.

5.1 Memoization Plugin

This compiler plugin enables memoization of computations by introducing simple annotations in code. The design of this plugin is influenced by CEAL [20], a C-based

Table 1. Lines of code of the compiler plugin case studies

Plugin	Lines of code	
	Original	Piuma
Memoization	1190	410 (34.5%)
HPE	2007	1422 (70.9%)
Atomic	510	209 (41.0%)
ScalaDyno	209	169 (80.9%)
Avro	914	681 (74.5%)

language for self-adjusting computation [1]. Using the plugin, programmers annotate a variable to be *modifiable*, indicating that the variable may be modified throughout the program execution and that any computations that depend on the variable are to be recomputed incrementally when the variable is modified. A modifiable variable can be read or written like any other variable.

The plugin is implemented as a code transformation. When the plugin encounters a read of a modifiable variable in a given method, it extracts the continuation of the read within that method into a new method. The extracted statements are replaced with a call to a closure, which takes the parameters of the extracted method and the modifiable variable, and only calls the extracted method if it has not already been called for the same arguments. The original plugin has a single phase, and uses the Scala compiler API to perform mainly AST creation and transformation.[3]

We mostly relied on `splitAfter`, `extractMethod` and `addMember` methods from the Piuma library, in addition to Piuma DSL's annotations. The first two methods perform AST refactoring as described in Section 4. The methods take care of binding symbols of the transformed trees to their new owners, as well as rebinding the symbols of all their children AST nodes. This avoids errors when rebinding symbols manually. The `addMember` method is a tree generator that generates type-checked trees. Using Piuma, the source code is clearer and easier to understand, as the complexity of the above tasks is implemented in the library. We were able to reduce the size of the plugin from 1190 to 410 lines, a reduction of 65.6%.

5.2 Hybrid Partial Evaluation Plugin

Hybrid-partial evaluation (HPE) [40] borrows ideas from both online and offline partial evaluation. It performs offline-style specialization using an online approach without static binding-time analysis. We constructed a Scala plugin that implements HPE by allowing programmers to annotate variables that should be evaluated at compile time. The main transformation performed by the plugin is method specialization, which requires AST transformation and generation.[4]

We used `mkCompanionObject`, `duplicate`, and `removeParam` functions, and other AST transformation utilities, in addition to Piuma DSL's annotations. We were able to

[3] The original plugin was developed by the first author and can be found at `https://github.com/amanjpro/piuma/tree/kara/kara`

[4] The original plugin was developed by the first author and can be found at `https://github.com/amanjpro/mina`

reduce the code size from 2007 to 1422 (a reduction of 29.2%). Piuma's usefulness here comes from its ability to automatically change method symbol information after adding or removing parameters.

5.3 Atomic Scala Plugin

This plugin is a port of Java's atomic sets [47] implementation. It allows declaring atomic memory regions, to which object fields may be added. Fields in the same atomic set are protected by the same lock, and the compiler acquires this lock whenever one of the fields in the atomic set is used in a public method. The syntax of this plugin is expressed using Scala annotations, which are processed using the plugin to insert (possibly nested) synchronization blocks in public method bodies. The plugin is composed of six phases, one performs type-checking, another two process annotations and store global information to be shared with other phases, and the rest perform AST creation and transformation operations: adding fields to classes, changing constructor signatures and altering constructor calls accordingly, and surrounding method bodies with synchronization blocks.[5]

Various tree generators and modifiers were employed, as well as Piuma DSL's annotations. The code size was reduced from 510 to 209 lines (a 59.0% reduction), again making code clearer and more concise, due to hiding the boilerplate of creating trees and managing their symbols, as well as the symbols of their owners and child-ASTs.

5.4 ScalaDyno Plugin

ScalaDyno [3] simulates dynamic typing in Scala. It works by replacing any compilation errors encountered by the compiler with warnings, postponing type checking to run time. This was the smallest plugin we reimplemented, with only 209 lines of code. It mainly performs info-transformation. Still, we were able to reduce its size to 169 lines of code, a reduction of 19.14%, by applying Piuma's type-transformation functions and by using the Piuma DSL's annotations. The main source of code reduction was the DSL, which eliminated the boilerplate code for declaring the plugin and its phases.[6]

5.5 Avro Records Plugin

This compiler plugin is used to auto-generate Avro serializable classes [2] from case class definitions. Avro is a framework developed by Apache's Hadoop group for remote procedure calls and data serialization. The programmer mixes in the `AvroRecord` trait into their case classes; at compile time, the plugin automatically generates the necessary methods to make these classes Avro-serializable. The plugin mostly generates companion objects, adds methods and fields to classes. Other than using the Scala compiler

[5] The original plugin was developed by the second author and can be found at `https://github.com/nosheenzaza/as`

[6] The source of the original plugin can be found at `https://github.com/scaladyno/scaladyno-plugin`

API, it also employs Scala's TreeDSL [38], which aims to make the AST generation code simpler and more readable.[7]

This plugin was written for the Scala 2.8 compiler runtime. To use Piuma, we needed to port the plugin to Scala 2.11. We mostly relied on the tree generators `mkVar`, `mkSetterAndGetter`, and `mkCompanionObject` and on the tree modifiers `addMember` and `updateRHS`, as well as Piuma DSL's annotations. Even though the original plugin used a DSL that is more concise than Scala compiler API, we were still able to reduce the code size from 914 to 681 lines (a 25.49% reduction), again, mainly because symbol creation and binding was handled under the hood by Piuma rather than by the plugin code itself. Our line count results compare the Piuma version with the original Scala 2.8 version of the plugin.

5.6 Discussion

Piuma led to a reduction in code size for all benchmarks. The resulting code was more concise and easier to understand. The main advantage of using Piuma is its ability to hide the complexity of tree and symbol creation, binding, and tree typing. Symbol binding is especially tedious when refactoring already existing ASTs. The library also provides many functions to perform common AST refactoring tasks.

Applying Piuma's library functions can sometimes yield malformed trees. For example, classes in Scala, as in many other languages cannot contain a return statement, so attempting to add a block that contains a return statement causes a type-checking error. We rely on Scala's type checker to report such errors to the programmer, and we found it to be an adequate aid when we performed evaluation.

While reimplementing the plugins in Piuma, we noticed and took advantage of opportunities to extend the library to support commonly occurring refactoring patterns. Three of the case study plugins were originally implemented by the first two authors of this paper. The authors found that reimplementing the plugins using Piuma was far less time consuming and error-prone than writing the original implementation. While this observation is highly subjective, we feel it is worth mentioning.

6 Related Work

Language and compiler extensibility has been well studied for decades. This work includes macros [12, 41, 44], extensible compiler frameworks [13, 32], and language frameworks [25, 27].

Scala recently introduced a Lisp-like macro system [7]. Scala limits macro implementation to statically accessible modules. To make writing macros easier, Scala also provides quasiquotes [39], which Piuma uses also for AST generation and matching. Quasiquotes in Scala are not hygienic [26], nor are they statically type checked. A number of DSLs are already implemented using Yin-Yang [24], a macro-based framework for implementing DSLs, for instance, OptiML [43], a DSL for machine learning.

[7] The source of the original plugin can be found at
`https://code.google.com/p/avro-scala-compiler-plugin/`

Unlike compiler plugins, macros can fall short when it comes to modifying the static semantics of the programming language. Macros can only manipulate the ASTs on which they are explicitly invoked. This limitation makes implementing something like the partial evaluation plugin impossible with macros, as it needs to access and modify non-local ASTs.

Lightweight Modular Staging [36, 37] and the Scala-virtualized extension [30] are used to implement deep embedded DSLs. Users programmatically generate ASTs, taking advantage of Scala-virtualized's more expressive operator overloading features to support domain-specific syntax.

In Java, one can add type checkers using JavaCOP [29], the Checker framework [11], or the standard *annotation processor* mechanism found in Java 6 [5]. Java 8 type annotations, defined by JSR308 [14], are based on the Checker framework. All these approaches allow a developer to extend the type system with pluggable type checkers, but they do not allow any code transformation [6]. Piuma, on the other hand, permits code transformation, which is inherently more problematic than the restrictive compiler extensions that only perform static checking [31].

Scala has a refactoring library [42] which provides a set of refactoring utilities to be used within IDEs. The .NET compiler platform [16] also provides an API for code analysis and refactoring. Wrangler [28] is a refactoring framework for Erlang that integrates with Emacs and Eclipse. What makes our work different from these is that Piuma provides refactoring for the ASTs to be used in compiler plugins, while the others focus on source code refactoring for use in IDEs.

Extensible compiler frameworks such as JastAdd [13] and Polyglot [32] are used for implementing new languages on top of a base language compiler. Unlike compiler plugins, these frameworks are designed to allow arbitrary changes to the base language. Many of the issues that occur with compiler plugins occur also with these frameworks.

7 Conclusions and Future Work

Our evaluation shows Piuma's usefulness to plugin writers. All plugins became more concise and clearer. Plugins written in Piuma can concern themselves more with the plugin logic rather than with tedious details of AST transformations or symbol management.

The Scala compiler assumes many complex, undocumented invariants. While developing Piuma we discovered many of these invariants by trial-and-error, that is by inadvertently violating them and observing the compiler crash or generate incorrect bytecode. We are investigating ways to test plugins for correctness and ways to automatically re-establish invariants that do get broken.

Implementing a refactoring library inside a complex compiler such as Scala's enables not only compiler extensibility through plugins. The Piuma library could also be leveraged for implementing refactorings in an IDE, in optimization tools, or in other refactoring tools. We plan to extend the Piuma framework with more refactorings and implement additional static analyses to ensure that plugin writers have more assurance that the refactorings are being used correctly.

Acknowledgments. This work was supported by the Swiss National Science Foundation (project 200021-138091).

References

1. Acar, U.A.: Self-adjusting computation (an overview). In: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2009, pp. 1–6. ACM, New York (2009)
2. Apache Foundation: Apache Avro Records, `http://avro.apache.org/`
3. Bastin, C., Ureche, V., Odersky, M.: ScalaDyno: Making Name Resolution and Type Checking Fault-tolerant. In: Proceedings of the Fifth Annual Scala Workshop, SCALA 2014, pp. 1–5. ACM, New York (2014)
4. Bentley, J.: Programming pearls: little languages. Commun. ACM 29(8), 711–721 (1986)
5. Bloch, J.: JSR 175: A metadata facility for the Java programming langauges (2004), `http://jcp.org/en/jsr/detail?id=175`
6. Bracha, G.: Pluggable type systems. In: OOPSLA 2004 Workshop on Revival of Dynamic Languages (2004)
7. Burmako, E.: Scala Macros: Let Our Powers Combine! In: 4th Annual Workshop Scala 2013 (2013)
8. Burmako, E.: Macro annotations (2014), `http://docs.scala-lang.org/overviews/macros/annotations.html`
9. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 519–538. ACM, New York (2005)
10. Danial, A.: Cloc, `http://cloc.sourceforge.net/`
11. Dietl, W., Dietzel, S., Ernst, M.D., Muslu, K., Schiller, T.W.: Building and Using Pluggable Type-Checkers. In: Software Enginnering in Practice Track, International Conference on Software Engineering (ICSE) (May 2011)
12. Dybvig, R., Hieb, R., Bruggeman, C.: Syntactic abstraction in Scheme. LISP and Symbolic Computation 5(4), 295–326 (1993)
13. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA 2007, pp. 1–18. ACM, New York (2007)
14. Ernst, M.: JSR 308: Annotations on Java Types (2004), `https://jcp.org/en/jsr/detail?id=308`
15. Flanagan, D., Matsumoto, Y.: The Ruby Programming Language, 1st edn. O'Reilly (2008)
16. Foundation, N.: NET compiler platform ("roslyn"), `https://github.com/dotnet/roslyn/` (2014)
17. Futamura, Y.: Partial evaluation of computation process–an approach to a compiler-compiler. Higher-Order and Symbolic Computation 12(4), 381–391 (1999)
18. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java(TM) Language Specification, 3rd edn. (Java (Addison-Wesley)). Addison-Wesley Professional (2005)
19. Haller, P., Odersky, M.: Capabilities for uniqueness and borrowing. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 354–378. Springer, Heidelberg (2010)
20. Hammer, M.A., Acar, U.A., Chen, Y.: CEAL: A C-based language for self-adjusting computation. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 25–37 (2009)

21. Hejlsberg, A., Wiltamuth, S., Golde, P.: C# Language Specification. Addison-Wesley Longman Publishing Co., Inc.„ Boston (2003)
22. Hudak, P.: Modular domain specific languages and tools. In: The Fifth International Conference on Software Reuse (ICSR) (1998)
23. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., Upper Saddle River (1993)
24. Jovanovic, V., Nikolaev, V., Pham, N.D., Ureche, V., Stucki, S., Koch, C., Odersky, M.: Yin-Yang: Transparent Deep Embedding of DSLs. Technical report, EPFL (2013), `http://infoscience.epfl.ch/record/185832`
25. Klint, P., van der Storm, T., Vinju, J.: Rascal: A domain specific language for source code analysis and manipulation. In: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, pp. 168–177. IEEE (2009)
26. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, pp. 151–161. ACM (1986)
27. Konat, G.D.P., Vergu, V.A., Kats, L.C.L., Wachsmuth, G., Visser, E.: The Spoofax Name Binding Language. In: SPLASH, pp. 79–80. ACM (2012)
28. Li, H., Thompson, S., Orosz, G., et al.: Refactoring with Wrangler: Data and process refactorings, and integration with Eclipse. In: Proceedings of the Seventh ACM SIGPLAN Erlang Workshop (September 2008)
29. Markstrum, S., Marino, D., Esquivel, M., Millstein, T., Andreae, C., Noble, J.: JavaCOP: Declarative Pluggable Types for Java. ACM Trans. Program. Lang. Syst. 32(2), 4:1–4:37 (2010)
30. Moors, A., Rompf, T., Haller, P., Odersky, M.: Scala-virtualized. In: Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, pp. 117–120. ACM, New York (2012)
31. Nystrom, N.: Harmless Compiler Plugins. In: Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs, FTfJP 2011, pp. 4:1–4:6. ACM, New York (2011)
32. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
33. Odersky, M., et al.: The Scala programming language (2006-2013), `http://www.scala-lang.org`
34. Peyton Jones, S.L.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003), `http://www.haskell.org/definition/haskell98-report.pdf`
35. Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. ACM Sigplan Notices 44(9), 317–328 (2009)
36. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. Commun. ACM 55(6), 121–130 (2012)
37. Rompf, T., Sujeeth, A.K., Amin, N., Brown, K.J., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., Odersky, M.: Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In: Giacobazzi, R., Cousot, R. (eds.) POPL, pp. 497–510. ACM (2013)
38. Scala Developers: TreeDSL, `https://github.com/scala/scala/blob/2.11.x/src/compiler/scala/tools/nsc/ast/TreeDSL.scala`
39. Shabalin, D., Burmako, E., Odersky, M.: Quasiquotes for Scala. Technical report, EPFL (2013), `http://infoscience.epfl.ch/record/185242`
40. Shali, A., Cook, W.R.: Hybrid Partial Evaluation. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2011, pp. 375–390. ACM, New York (2011)

41. Steele, G.L., Gabriel, R.P.: The evolution of Lisp. In: The Second ACM SIGPLAN Conference on History of Programming Languages, HOPL-II, pp. 231–270. ACM, New York (1993)
42. Stocker, M., Sommerlad, P.: Scala Refactoring. Master's thesis, University of Applied Sciences Rapperswil (2010)
43. Sujeeth, A.K., Lee, H., Brown, K.J., Rompf, T., Chafi, H., Wu, M., Atreya, A.R., Odersky, M., Olukotun, K.: OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In: Getoor, L., Scheffer, T. (eds.) ICML, pp. 609–616. Omnipress (2011)
44. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages As Libraries. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 132–141. ACM, New York (2011)
45. Tu, S., et al.: A Scala compiler plugin for Avro records (2011), http://code.google.com/p/avro-scala-compiler-plugin
46. Ureche, V., Talau, C., Odersky, M.: Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, pp. 73–92. ACM (2013)
47. Vaziri, M., Tip, F., Dolby, J., Hammer, C., Vitek, J.: Type System for Data-Centric Synchronization. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 304–328. Springer, Heidelberg (2010)
48. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: 12th International Workshop on Foundations of Object-Oriented Languages (2005)