Self-Assembly: Lightweight Language Extension and Datatype Generic Programming, All-in-One!

Heather Miller
EPFL
heather.miller@epfl.ch

Philipp Haller
Typesafe, Inc.
philipp.haller@typesafe.com

Bruno C. d. S. Oliveira
The University of Hong Kong
bruno@cs.hku.hk

Abstract

In this paper we show a general mechanism, called `self-assembly`, for *lightweight language extensions* (LLEs). LLEs allow users to define *generic operations* or *properties* that operate over a large class of types. With LLEs it is possible, for example, for users to define their own Java-style automatic serialization mechanism; or implement simple forms of custom pluggable type system extensions like an immutability checker. However unlike language built-in mechanisms (such as Java serialization), LLEs are *user-definable*, *multi-purpose* (they can be used to define various forms of generic functionality), and *highly customizable and extensible*. The key idea, inspired by existing datatype-generic programming approaches, is to provide programmers with a generic mechanism for providing automatic implementations of *type classes*. We implemented our technique as a library, `self-assembly`, for Scala, and evaluated its practicality by migrating a full-featured industrial-strength serialization framework, Scala/Pickling, keeping the same published performance numbers while reducing the code size for type class instance generation by 56%.

1. Introduction

Defining functionality that should apply to a large set of types is a common problem faced by both language designers and normal users. One common approach is to provide specialized functionality across arbitrary types at the level of the compiler or runtime. For example, in Java, every object is synthetically provided with a few methods; `toString`, `equals`, `clone`, and `hashCode`. Serialization, on the other hand, is also an ubiquitously needed functionality, but unlike the above, Java does not ensure that serialization functionality exists for every type. Instead, serialization in Java is opt-in; if a class implements a `Serializable` interface then instances of that class are automatically serializable by the JVM. While compiler/runtime-integrated approaches such as Java's serialization are typically easy to use (no boilerplate required), they are inflexible and are often impossible to customize. For example, it is not possible to adapt Java serialization to work with other formats (such as JSON or XML).

Library-based approaches to generic programming which require *type classes* [33] as a language feature are a lot more flexible. Type classes provide a mechanism where a certain functionality can be captured in an interface. When programmers need certain types of values to support a given functionality, they can implement an *instance* of a type class. Type classes support *retroactive extensibility* [17]; functionality can be implemented *after* the type or class has been defined. This is in contrast with conventional OO programming, where all methods (such as `toString` or `equals`) are implemented together with the definition of the class. Retroactive extensibility enables flexibility and the possibility to customize behavior. As a result, several authors have argued for the software engineering benefits of using type classes [17, 26], and Scala has embraced them [21, 24, 26].

Type classes are more general than built-in mechanisms like Java serialization, since any functionality (including serialization) can be modeled as a type class. However, an approach based on type classes is not without challenges. To provide functionality across a large number of types, users are required to implement many type class instances manually. To reduce this vast amount of boilerplate, there have been a number of proposals for *datatype-generic programming* (DGP) [15, 29]. DGP is an advanced form of *generic programming* [23], where generic functions can be defined by inspecting the structure of types. For this, library-based approaches typically introduce run-time type representations. However, those come with a significant performance penalty [1]. Moreover, the vast majority of DGP approaches has been developed for Haskell, and are thus fundamentally limited when ported to mainstream OO languages, due to their lack of support for subtyping or object identity.

These compiler-based and library-based approaches using type classes are at odds with one another. On the one hand, language-integrated approaches can be more powerful in the sense that they can do a great deal of static analysis, and because they are so specialized, typically require no boilerplate to programmers. However this is done at the cost of customizability and extensibility. On the other hand, with type class-based approaches, one must contend with an enormous amount of boilerplate or pay a non-negligible performance penalty;[1] in all cases, however, type class-based approaches offer no way to statically restrict runtime behavior. Perhaps most important for mainstream languages is the lack of support for pervasively used object-oriented features such as subtyping and object identity, which so far have not been addressed, except for specialized functionality [21].

In this paper, we attempt to strike a sweet spot in the design space. Our approach is guided by the following principles:

- **Extensibility and customizability**. Like for type class-based approaches, retroactive extensibility and type-based customization should be supported.
- **Little boilerplate**. Like language-integrated approaches, usage of generic code should *feel* built-in. Users shouldn't have to define type class instances or provide a lot of scaffolding.
- **Performance**. Generic functions written by library authors or library users should have the same or better performance than approaches with compiler/runtime support.
- **Generality**. In addition to generic functions, lightweight static analysis capabilities should be supported.

In our previous work on Scala/Pickling [21], we sought to achieve many of these goals for one particular application: serialization. Scala/Pickling is based on type classes which are generated and composed at compile time, according to their type signatures. Due to its compile-time properties, serialization code is fast and inlined, without requiring any boilerplate. Due to the fact that it

[1] Some approaches trade type-safety for performance [1].

is completely based upon type classes, flexibility and extensibility come for free. However, the approach is specialized on providing type class instances for only the `Pickling` type class. Other type classes or generic functions are not supported.

In this paper, we present self-assembly, a general technique for *lightweight language extensions* (LLEs). LLEs allow users to define generic operations or properties that operate over a large class of types. Importantly, the technique supports many features of mainstream OO languages such as subtyping, object identity, and separate compilation. So far, these features have been missing in existing approaches for DGP; in addition, we also support these features for generic properties.

We additionally provide a library, also called `self-assembly`, for Scala, which embodies this technique. To validate our approach, we migrated the full-featured, industrial-strength Scala/Pickling[2] framework to be based upon self-assembly. Importantly, the refactoring preserves its high performance, flexibility, customization, and absence of boilerplate. In addition, the use of self-assembly led to a significant reduction in code size, and improved code clarity.

Finally, we also show a different application of LLEs: *generic properties*. In `self-assembly` it is possible to define some forms of lightweight static checking, which guarantee that a certain property, *e.g.,* deep immutability, holds. In this case, if a class is immutable, the immutability checker generates a type class instance for that class, which certifies that property.

In summary the contributions of this paper are:

- **Self-Assembly**, a general technique for LLEs that requires little boilerplate; shares the extensibility and customizability properties of type classes; and, due to compile-time code generation, provides high performance. It allows defining generic functions in a statically type-safe way.
- **A full-featured DGP approach for OOP.** `self-assembly` enables the definition of datatype-generic functions that support features present in production OO languages, including subtyping, object identity, and generics.
- **Support for generic properties.** `self-assembly` enables lightweight pluggable type system extensions to guarantee that certain static properties hold at runtime, *e.g.,* immutability.
- **The `self-assembly` library**, a complete and full-featured implementation of our technique in and for Scala. The library includes several auxiliary definitions, such as generic queries and transformations, that help define new LLEs. Importantly, `self-assembly` doesn't require any extension to the language or compiler.
- **A case study on basing Scala/Pickling on `self-assembly`.** We evaluate the expressivity and performance of `self-assembly` by porting a full-featured serialization framework, keeping the same published performance numbers while reducing the code size for type class instance generation by 56%.

2. Type Classes and a Boilerplate Problem

This section provides an introduction to type classes [33] and reviews how to encode them in Scala using implicits and conventional OO features [26]. This section also observes that type class instances for various types tend to require code that follows a common pattern. The pattern can be viewed as a source of code boilerplate, since similar code needs to be repeated throughout several definitions. The remainder of the paper aims at showing how to capture the pattern as reusable code and generate type class instances automatically from that code.

[2] `https://github.com/scala/pickling`

```
trait Show[T] {def show(visitee : T) : String}

implicit object IntInstance extends Show[Int] {
  def show(o : Int) = o.toString()
}
```

Figure 1. `Show` type class and corresponding instance for integers.

2.1 Implicits

In Scala, it is possible to select values automatically based on type. These capabilities are enabled when using the `implicit` keyword. For example, a method `log` with multiple parameter lists may annotate their last parameter list using the `implicit` keyword.

```
def log(msg: String)(implicit o: PrintStream) =
  o.println(msg)
```

This means that in an invocation of `log`, the implicit argument list may be omitted if, for each parameter of that list, there is exactly one value of the right type in the *implicit scope*. The implicit scope is an adaptation of the regular variable scope. Imported implicits, or implicits declared in an enclosing scope are contained in the implicit scope of a method invocation.

```
implicit val out = System.out
log("Does not compute!")
```

In the above example, the implicit val `out` is in the implicit scope of the invocation of `log`. Since `out` has the right type, it is automatically selected as an implicit argument.

2.2 Type Classes

Type classes are a language mechanism that provide a disciplined alternative to ad-hoc polymorphism. They have been popularized by Haskell. Type classes allow functions to be defined over a set of types. If values of a type `T` should provide a certain functionality then that functionality can be specified as an *instance* of a type class.

In Scala type classes can be implemented using a combination of standard OO features (traits, classes and objects) and implicits [26]. The Scala encoding of type classes is essentially a *design pattern* [13]: instead of having built-in language concepts for type classes, Scala uses general language features to model type classes. A type class is simply an interface that provides operations over one (or more) generic types. Such interfaces can be modeled as traits in Scala. An example of a type class is shown in Figure 1. The trait `Show[T]` models a type class that provides pretty printing functionality for some type `T` via a method `show`.

The main conceptual difference between standard OO methods and type-class methods is that the later are provided *externally* to objects. Suppose that we wanted to add pretty printing functionality to integers. To do this we create an instance of the type class `Show` where the generic type parameter `T` is instantiated to `Int`. The **`object`** `IntInstance` in Figure 1 models such instance in Scala using regular objects. In that object, the `show` method takes an argument `o` of type `Int` an invokes the `toString()` method on `o`.

Type-Directed Resolution of Instances An interesting aspect of type classes is that instances can be automatically determined using a type-directed resolution mechanism. This type-directed resolution mechanism allows type classes to be used from client code through a mechanism similar to overloading. This is achieved in Scala using an implicit parameter:

```
def ishow[T](v : T)(implicit showT : Show[T]) =
  showT.show(v)
```

In `ishow` the idea is that the method takes two parameters, with the last of these (`showT`) being implicit. As we have seen in Section 2.1 this means that the second parameter can be automatically determined by the compiler. For example if we wanted to use `show` on integers we could simply write a program such as:

```
sealed trait Tree
case class Fork(left : Tree, right : Tree)
  extends Tree
case class Leaf(elem : Int) extends Tree

implicit object TreeInst extends Show[Tree] {
 def show(visitee : Tree) : String = visitee match {
   case Fork(l,r) =>
    "Fork(" + show(l) + ", " + show(r) + ")"
   case Leaf(x) => "Leaf(" + x.toString() + ")"
 }}
```

Figure 2. Trees of integers and corresponding `Show` instance.

```
def test1 = ishow(5)
```

Provided that an **implicit** value of type `Show[Int]` is in the implicit scope (for example `IntInstance` from Figure 1), the second parameter is automatically inferred by the compiler.

Context Bounds Type classes are pervasively used in Scala. Because of this Scala offers an alternative convinient syntax sugar called *context bounds*. Context bounds allows code using type classes to be written more compactly and arguably more intuitively. With context bounds, instead of writting `ishow` we could write:

```
def show[T : Show](v : T) =
  implicitly[Show[T]].show(v)
```

The idea of context bounds comes from the fact that type classes can also be seen as a generic programming mechanism [23], which allows generic parameters to be constrained. In this case the type of `show` can be read as a generic method where the generic type argument must be an instance of `Show`. A small problem with context bounds there is no parameter name to be used in the definition of `show`. However, it is possible to *query* the implicit scope for a value of a certain type using a simple auxiliary method called `implicitly`:

```
def implicitly[T](implicit x : T) : T = x
```

This precludes the need for having to have the name of the implicit argument in hand in order to use it. From the client perspective, using `show` is similar to using `ishow`.

2.3 Pretty Printing Complex Structures

Of course it is also possible to apply type classes to more complex structures. For example consider a simple type of binary trees with integers at the leafs. Figure 2 shows how to model such trees in Scala using *case classes* [11] and *sealed traits*. The keyword **`sealed`** in Scala means that the trait can only be implemented by definitions in the existing compilation unit. Together with case classes this allows modeling *algebraic datatypes*, which are a well-know concept from functional programming. The `Tree` trait is the type of trees. The case class `Fork` models the binary nodes of the tree, wheres the case class `Leaf` models the leafs containing an integer value.

To define pretty printing for `Tree` using the `Show` type class we create an **`object`** `TreeInst`. This object provides a definition for the `show` method that pattern matches on the two tree constructors (cases) of `Tree`. The implementation of the two cases is unremarcable: both cases print the constructors names and the arguments.

A simple test program illustrating the use of `TreeInst` is shown next. The value `tree` defines a simple tree and the definition `test3` pretty prints that tree.

```
val tree : Tree = Fork(Fork(Leaf(3),Leaf(4)),Leaf(5))
def test3 = show(tree)
```

Recursive Resolution and Compositionality of Instances Another interesting aspect of type classes is that they provide a highly compositional way to define instances. Lets consider a variant of trees, shown in Figure 3, which is parametrized by some element type `A`. The type these trees is `PTree[A]` and there are two types of nodes: `Branch` nodes with an element of type `A` and two branches; and `Empty` nodes with no content.

```
sealed trait PTree[A]
case class Branch[A](x: A, l: PTree[A], r: PTree[A])
  extends PTree[A]
case class Empty[A] extends PTree[A]

implicit def PTreeInst[A : Show] : Show[PTree[A]] =
  new Show[PTree[A]] {
    def show(visitee : PTree[A]) = visitee match {
      case Branch(x,l,r) =>
        "Branch(" + implicitly[Show[A]].show(x) +
        ", " + show(l) + ", " + show(r) + ")"
      case Empty() => "Empty()"
  }}
```

Figure 3. Parametrized trees and corresponding `Show` instance.

Like other types it is possible to define an instance (`PTreeInst`) for the type `PTree[A]`. However in order to pretty print such trees it is necessary to know how to print the elements of type `A` as well. To accomplish this we require that the generic type parameter `A` has a `Show` instance using a context bound. To print the elements in the `Branch` case, the instance can be retrieved from the implicit scope using `implicitly` and then used to print the element. With this instance it is possible to print trees with integer elements, such as:

```
val ptree : PTree[Int] = Branch(5,Empty,Empty)
def test4 = show(ptree)
```

However, more interestingly, it is also possible to print trees where for any element type that has a `Show` instance. For example:

```
val ptree2 : PTree[PTree[Tree]] =
  Branch(Branch(tree,Empty,Empty),Empty,Empty)
def test5 = show(ptree2)
```

Here `ptree2` has elements of type `PTree[Tree]`. To print `ptree2` the instance for `PTree` is used twice: once for values of type `PTree[PTree[Tree]]`; and another time for values of type `PTree[Tree]`. In fact it is possible to use arbitrarely many instances of the various types (possible multiple times) during type-directed resolution, which makes the process very compositional. This is possible because the type-directed resolution mechanism is recursive.

2.4 A Boilerplate Problem

Although type classes are nice, they often require similar code for different instances. For example consider the two instances in Figures 2 and 3. The code that is needed in both instances is quite similar and it follows a common pattern: for each case the constructor name and parameters are printed. Therefore code tends to be quite similar across instances. This code can be viewed as a form of boilerplate since we could hope that it could be mechanically generated.

3. Type-Safe Meta-Programming in Scala

Scala macros [5, 6] enable a form of type-safe meta-programming. Macros are methods that are invoked at compile time. Instead of runtime values, macros operate on and return typed expression trees. In the following we provide an overview of macros, type checking, and properties.

3.1 Definition

Macro defs are methods that are transparently loaded by the compiler and executed (or expanded) during compilation. A macro is defined like any normal method, but it is linked using the `macro` keyword to an additional method that provides its implementation, which operates on expression trees.

```
def assert(x: Boolean, msg: String): Unit =
  macro assert_impl
def assert_impl(c: Context)
```

```
object Show extends Query[String] {
  def mkTrees[C <: SContext](c: C) = new Trees(c)

  class Trees[C <: SContext](override val c: C)
      extends super.Trees(c) {
    import c.universe._
    type SExpr = c.Expr[String]

    def combine(left: SExpr, right: SExpr) =
      reify { left.splice + right.splice }

    def delimit(tpe: c.Type) = {
      val start = constant(tpe.toString + "(")
      (start, reify(", "), reify(")"))
    } }

  implicit def generate[T]: Show[T] =
    macro genQuery[T, this.type]
}
```

Figure 4. Implementing the `Show` type class using `self-assembly`.

```
  (x: c.Expr[Boolean], msg: c.Expr[String]):
                            c.Expr[Unit] = ...
```

In the above example, the parameters of `assert_impl` are typed expression trees, which the body of `assert_impl` operates on, itself returning an expression of type `Expr[Unit]`. `assert_impl` is evaluated at compile time, and its result is inlined at the call site of `assert`. Note that expression trees are typed, *i.e.,* `assert`'s parameter of type `Boolean` corresponds to a typed expression tree of type `Expr[Boolean]`.

In the type-safe subset of macros that we consider in this paper, expression trees are built using `reify`/`splice`:

```
val expr: c.Expr[Boolean] = reify {
  if (x.splice > 10) x.splice
  else true
}
```

Here, the body of `reify` consists of regular Scala code. Expressions in the enclosing scope are spliced into the result expression using the `splice` method. Importantly, the code within `reify` is type-checked at its definition site. This means, for the above code, Scala's type checker reports type errors not in terms of the generated code, but in terms of the high-level user-written code.

Due to limitations in the reify API, we use quasiquotes (type-checked during macro expansion) to circumvent the above type-checking in a small trusted core of `self-assembly`, shielded from users. However, we never lose soundness, since, unlike MetaML [32], all splicing is done at compile time, and generated expressions are always re-type-checked after expansion.

3.2 Properties

Constant Type Signatures In this work, we focus on one of two macro def varieties: "blackbox" macros. In this case, the type signature of the macro provides all information necessary for type-checking all of its invocations. That is, the macro does not have to be expanded prior to type-checking. This has important software engineering benefits, namely that abstract, type-based reasoning about programs is maintained independently of the macro's corresponding implementation. This is particularly useful when reasoning about the result type of a macro. For blackbox macros, the implementation (and expansion) is not required to determine the result type.

Local Expansion Since macros are simply methods that are invoked at compile time, they are expanded and inlined at invocation site. For this reason, we consider macro defs to be "local compiler extensions." They cannot change the compiler's global symbol table. Thus, they cannot introduce new top-level type definitions.

4. Basic Self-Assembly

Section 2 showed how to write type classes like `Show[T]` manually, pointing out a source of significant boilerplate code. In section 4.1, we outline the basic usage of the `self-assembly` library, which allows defining type classes desired in a way where the required boilerplate is automatically generated. Section 4.2 explains the mechanics of the automatic type class generation implemented in the `self-assembly` library. Section 4.3 outlines how one can customize the generation of type classes for specific types.

4.1 Basic Usage

The `self-assembly` library allows implementing type classes instances automatically on demand at compile time. This main idea is introduced using the simple `Show` type class in Figure 1. Section 6 shows how our approach extends to different forms of type classes, commonly referred to as queries and transformations [18].

Generating Instances for `Show` Suppose a user wants to provide instances of `Show[T]` for as many types as possible. Using `self-assembly` we can create a singleton object that extends a library-provided trait, and that implements two factory methods, `generate` and `mkTrees`. Figure 4 shows the `Show` companion object,[3] which extends the `Query` trait. The `mkTrees` factory method, abstract in `Query`, creates a new `Trees` instance; `Trees[C]` provides a number of methods that are invoked by the `self-assembly` library at *compile time* to obtain AST fragments that are inlined in the generated code. The `Show` type class converts objects to strings; thus, the query has to define how to assemble result strings, based on an associative combination operator (`combine`), begin/end delimiters (`first`/`last`), and a separator. As mentioned in Section 3, the syntax `reify { ... }` creates a typed expression based on Scala code. `left.splice` splices the expression `left` into the result expression. The compiler type-checks `reify` blocks at their definition site.

Apart from implementing a subclass of `Trees[C]`, the `Show` singleton object also needs to define a generic implicit method (here, `generate`) that invokes the generation macro `genQuery`. The `genQuery` macro is provided by our library.[4]

Result With the `Show` singleton object defined as in Figure 4 it is no longer necessary for the user to define a type class instance for every single type manually. Instead, whenever an instance of type, say, `Show[MyClass]`, is required (typically, using an implicit parameter), Scala's type checker automatically inserts a call to the *implicit def* `generate[MyClass]`; this implicit def generates a suitable implementation of the searched type class instance on-the-fly. As a result, type class instances do not have to be defined manually.

4.2 Generation Mechanism

We illustrate the general idea of our generation technique through a simple example based solely on closed ADT-style datatypes in Scala. Such datatypes consist of either sealed traits or case classes extending such traits. In subsequent sections, we generalize this view to richer types.

Our treatment is centered on an example, in which, our goal is to automatically "derive" type class instances that "show" information about a given type. Think of it as a `toString` method that traverses the structure of a type, and nicely prints information about all of the fields of that type.

We structure our treatment into three distinct steps: (1) in Section 4.2.1, we show how our generation is triggered; (2) in Section 4.2.2, we explain our macro-based generation technique; (3) in

[3] A companion object is a singleton object with the same name as a trait.

[4] The type argument `this.type` is the type of the enclosing singleton object; it is passed to `genQuery` to identify the type class and the `mkTrees` method that should be used by the library to generate instances.

```
trait Query[R] ... {
  def mkTrees[C <: Context with Singleton](c: C)
    : Trees[C]

  abstract class Trees[C <: Context with Singleton]
    (override val c: C) extends super.Trees(c) { }

  def genQuery[T:c.WeakTypeTag, S:c.WeakTypeTag]
    (c: Context): c.Tree = {
    import c.universe._
    val tpe = weakTypeOf[T]
    val stpe = weakTypeOf[S]
    val tpeOfTypeClass =
      stpe.typeSymbol.asClass.companion.asType
          .asClass.toTypeConstructor
    val qresTpe =
      tpeOfTypeClass.decls.head.asMethod.returnType

    val trees = mkTrees[c.type](c)
    ...
```

Figure 5. Macro-based generation: set-up

Section 4.2.3, we show some example type class instances that result from our generation technique, and relate them to the type class pattern introduced in Section 2.2.

4.2.1 Triggering Generation

To be able to generate suitable instances for all possible types for which `Show[T]` can be defined, we put an implicit macro into the companion object of `Show[T]`. The fact that the implicit macro is inside the companion object means that whenever an instance `Show[S]` is requested, Scala's implicit lookup mechanism searches the members of the companion object `Show` where it finds the implicit macro:

```
object Show extends Query[String] {
  ...
  implicit def generate[T]: Show[T] =
    macro genQuery[T, this.type]
}
```

Thus, the implicit lookup mechanism inserts an invocation of the macro method `genQuery`.

4.2.2 Macro-Based Generation

Being a macro, `genQuery` returns an abstract syntax tree instead of a (runtime) value. It is declared as follows:

```
def genQuery[T:c.WeakTypeTag, S:c.WeakTypeTag]
    (c: Context): c.Tree = ...
```

Note that in this declaration, the type parameters `T` and `S` are annotated with *context bounds* `c.WeakTypeTag`. First, the macro collects information about the types and the type class for which an instance should be generated. Second, the macro creates an instance of the user-provided `Trees` class by invoking the `mkTrees` factory method. These steps are shown in Figure 5.

The body of the type class is generated using:

```
val tpe = weakTypeOf[T] // see Fig. 5
...
val (first, separator, last) =
  trees.delimit(tpe)
val body = trees.combine(
  fieldsExpr(first, separator), last)
```

To create the result expression, the macro utilizes the `trees` instance (of type `Trees`) that we initialize in the set-up phase (see Figure 5). Calling `delimit` returns three expressions ("delimiters") of type `Expr[R]` based on the reified type `tpe`. Recall that `tpe` corresponds to type parameter `T`, which is the type for which the macro generates a type class instance. The `fieldsExpr` method creates an `Expr[R]` by folding the `Expr[R]`s obtained for each field (see below) using the user-overridden `combine` method:

```
if (paramFields.size < 2)
  ...
else
  paramFields.tail.foldLeft(first) { (acc, sym) =>
    val withSep = trees.combine(acc, separator)
    trees.combine(withSep, fieldValue(sym))
  }
```

For example, Figure 4 shows that the definition of `combine` for `Show` is just string concatenation. As a result, this code concatenates the string values of all fields separated with `separator`.

The expression tree `fieldValue(sym)` is obtained as follows. For each field declared in type `tpe`, the following subexpression is generated:

```
val symTp = sym.typeSignatureIn(tpe)
val fieldName = sym.name.toString.trim
trees.fieldValueExpr(visitee, fieldName,
  symTp, tpeOfTypeClass)
```

The invocation of `fieldValueExpr` expands to (a) a nested lookup of a type class instance for the field, and (b) an invocation of the type class method:

```
def fieldValueExpr(visitee: c.Expr[T], name: String,
  tpe: c.Type, tpeOfTypeClass: c.Type): c.Expr[R] =
c.Expr[R](
  q"""
    implicitly[${appliedType(tpeOfTypeClass, tpe)}]
      .apply($visitee.${TermName(name)})
  """)
```

The syntax `q"""..."""` indicates the use of a quasiquote to create an *untyped* tree that is cast to an `Expr[R]`, effectively forming part of a small trusted core of `self-assembly`. The main reason for creating an untyped tree at this point is that the value of field "name" is obtained using only the field's name–the selection `$visitee.${TermName(name)}` must fundamentally be untyped. It is clear, though, that the result will be of type `R`, since that's the result type of all type class instances of type `tpeOfTypeClass`.

```
(1) implicit object CShowInstance extends Show[C] {
  (2) def show(visitee: C): String = {
        var result = "C("

    (3) val inst_1 = implicitly[Show[D1]]
(5) (4) result += inst_1.show(visitee.p_1)
        ...
        val inst_n = implicitly[Show[DN]]
        result += inst_n.show(visitee.p_n)
        result += ")"
    }
  }
```

Figure 6. Basic generation of type classes.

4.2.3 Generated Type Class Instances

The generation technique explained in the previous section produces implicit (singleton) objects which correspond to the type class instances portion of the type class pattern introduced in Section 2.2.

Let's say the datatype that we'd like to call `show` on is the `Tree` type in Figure 2. In order to create a type class instance of type `Show[Tree]`, we also create type class instances for `Tree`'s two subclasses, `Fork` and `Leaf`. `Fork` and `Leaf` are case classes with the general shape:

```
case class C(p_1: D_1, ..., p_n: D_n)
  extends E_1 with ... with E_m { ... }
```

An arbitrary type class instance (implicit singleton object) can be generated using the technique described in the previous section. Figure 6 shows the general structure that is generated for an arbitrary shape `C`. The implicit object (1) is exactly the same as in the manual type class pattern described in Section 2.2. (2) is the implementation of the single abstract method of the type class (the

```scala
// File PersonA.scala:
abstract class Person {
  def name: String
  def age: Int
}
case class Employee(n: String, a: Int, s: Int)
  extends Person {
  def name = n
  def age = a
}

// File PersonB.scala:
case class Firefighter(n: String, a: Int, s: Int)
  extends Person {
  def name = n
  def age = a
  def since = s
}
```

Figure 7. Open class hierarchy

show method of the Show trait). (3) is the result of expanding the implicitly invocation within the method fieldValueExpr above. (4) corresponds to the accumulation logic which itself results from the fold of paramFields above (to simplify the presentation we use the result accumulator variable instead of a deeply nested tree). Finally, (5) corresponds to first and last in the body of the macro-generated implementation of Show's single abstract method, show.

4.3 Customization

Generation as provided by self-assembly is convenient, but in some cases it is desirable to have full control over the type class instances for specific types (one strength of the type class pattern as introduced in Section 2.2). When using the self-assembly library, customization is still possible. It is sufficient to define custom instances for selected types manually; these custom instances are then transparently picked up and chosen in place of automatically-generated ones. It is even possible to use Scala's scoping and implicit precedence rules to prioritize certain instances over others.

5. Self-Assembly for Object Orientation

A cornerstone of the design of self-assembly is its support for features of mainstream OO languages. The following Section 5.1 explains how our approach supports subtyping polymorphism in the context of open class hierarchies (Section 5.1.1) and separate compilation (Section 5.1.2). In Section 5.2 we discuss how self-assembly handles cyclic object graphs, which are easily created using mutable objects with identity.

5.1 Subtyping

Object-oriented languages like Java or Scala enable the definition of a *subtyping relation* based on class hierarchies. Given the pervasive use of subtyping in typical object-oriented programs, our approach is designed to account for *subtyping polymorphism*. In addition, we provide mechanisms that enable the object-oriented features even in a setting where modules/packages are separately compiled.

5.1.1 Open Hierarchies

Classes defined in languages like Java are by default "open," which means that they can have an unbounded number of subclasses spread across several compilation units. By contrast, *final classes* cannot have subclasses at all. In addition, *sealed classes* in Scala can only have subclasses defined within the same compilation unit.

Our approach enables the generation of type class instances even for open classes. For example, consider the class hierarchy shown in Figure 7. The self-assembly library can automatically generate an instance for type Person:

```scala
val em = Employee("Dave", 35, 80000)
val ff = Firefighter("Jim", 40, 2004)
val inst = implicitly[Show[Person]]
println(inst.show(em))
// prints: Employee(Dave, 35, 80000)
println(inst.show(ff))
// prints: Firefighter(Jim, 40, 2004)
```

Note that we are using the same Show instance to convert both objects to strings.

Generation Concrete instances of a classtype, such as Person in Figure 7, in general have subtypes (dynamically). One approach to account for subtypes is by building the logic for all possible subtypes into the type class instance for the supertype, like is shown in Figure 2 in Section 2.3. However, such an approach does not support open class hierarchies, where new subclasses can be added in additional compilation units.

To support open class hierarchies, the generation of type class instances for open classes adds a *dispatch step*. For a class like Person in Figure 7, a dynamic dispatch is generated to select a specific type class instance based on the runtime classtype of the object that the type class is applied to (visitee):[5]

```scala
implicit object PersonInst extends Show[Person] {
  def show(visitee: Person): String =
    visitee match {
      case v1: Employee =>
        implicitly[Show[Employee]].show(v1)
      case v2: Firefighter =>
        implicitly[Show[Firefighter]].show(v2)
    }
}
```

5.1.2 Separate Compilation

To support subtyping polymorphism not only across different compilation units, but also across separately-compiled modules,[6] self-assembly provides *dynamic instance registries*. In the case of separately-compiled modules, subclasses for which we would like to generate instances are in general only discovered at link time. To be able to discover such subclasses, self-assembly allows registering generated instances with an *instance registry* at runtime. A reference to such an instance registry can then be shared across separately-compiled modules.

For example, module A could create a registry and populate it with a number of instances:

```scala
implicit val reg = new SimpleRegistry[Show]
reg.register(classOf[Employee],
             implicitly[Show[Employee]])
reg.register(classOf[Firefighter],
             implicitly[Show[Firefighter]])
...
```

Note that the registry reg is defined as an *implicit value;* as we explain in the following, this is required to enable registry look-ups when dispatching to type class instances based on runtime types.

With the instance registry set up in this way, another separately-compiled module B is then able to dispatch to instances registered by module A:

```scala
implicit val localReg = getRegistryFrom(moduleA)
localReg.register(classOf[Judge],
                  implicitly[Show[Judge]])
...
```

Importantly, when module B invokes the show method of an instance instP of type Show[Person], passing an object with dynamic type Employee, the generated instance instP dispatches to the correct type class instance of type Show[Employee] through a look-up in registry localReg.

[5] Simplified; handling of null values is omitted for simplicity.

[6] The Scala ecosystem distributes modules in separate "JAR files" typically.

Generation To enable registry look-ups, we augment the dispatch logic with a default case:[7]

```
case _ => {
  val reg$1 = implicitly[Registry[Show]]
  val lookup$2: Option[Show[_]] = reg$1.get(clazz)
  lookup$2.get.asInstanceOf[Show[Person]]
            .show(visitee)
}
```

5.2 Object Identity

In object-oriented languages like Scala, it is important to take *object identity* into account. Simple datatypes such as case classes already permit cycles in object graphs via re-assignable fields (using the `var` modifier). It is therefore important to keep track of objects that have already been visited to avoid infinite recursion.

To enable the detection of cycles in object graphs, we keep track of all "visited" objects during the object graph traversal performed by a type class instance. However, it is not sufficient to maintain a single, global set of visited objects, since implementations of one type class might depend on other type classes; different type class instances could therefore interfere with each other when accessing the same global set (yielding nonsensical results). Thus, it is preferable to pass this set of visited objects on the call stack. With the mechanics introduced so far, this is not possible.

To enable passing an additional context (the set of visited objects) on the call stack, we require type classes to extend `Queryable[T, R]`:

```
trait Queryable[T, R] {
  def apply(visitee: T, visited: Set[Any]): R
}
```

The `Queryable[T, R]` trait declares an `apply` method with an additional `visited` parameter (compared to the trait of the type class), which is passed the set of visited objects. This extra method allows us to distinguish between top-level invocations of type class methods and inner invocations (of `apply`). The only downside is that custom type class instances are slightly more verbose to define, although the implementation of `apply` can typically be a trivial forwarder.

For example, consider the `Show[T]` type class, now extending `Queryable[T, String]`:

```
trait Show[T] extends Queryable[T, String] {
  def show(visitee: T): String
}
```

A type class instance for integers can be implemented as follows:

```
implicit val intHasShow = new Show[Int] {
  def show(visitee: Int): String = "" + x
  def apply(visitee: Int, visited: Set[Any]) =
    show(visitee)
}
```

Note that the implementation of `apply` is trivial.

Generation To enable the detection of cycles in object graphs it is necessary to adapt the implementation of the implicit object as follows.

```
implicit object CShowInstance extends Show[C] {
  def show(visitee: C): String =
    apply(visitee, Set[Any]())
  def apply(visitee: C, visited: Set[Any]) =
    ...
}
```

Note that an invocation of `show` is treated as a *top-level invocation* forwarding to `apply` passing an empty set of visited objects. Crucially, when applying the type class instances for the class parameters of `C`, instead of invoking `show` directly, we invoke `apply` passing the `visited` set extended with the current object (`visitee`).

```
var result: String = ""
if (!visited(visitee.p_1)) {
  val inst_1 = implicitly[Show[D_1]]
  result = result +
   inst_1.apply(visitee.p_1, visited + visitee)
}
...
if (!visited(visitee.p_n)) {
  val inst_n = implicitly[Show[D_n]]
  result = result +
   inst_n.apply(visitee.p_n, visited + visitee)
}
```

6. Transformations

The library provides a set of traits for expressing generic functions that are either (a) queries or (b) transformations. Basically, a query generates type class instances that traverse an object graph and return a single result of a possibly different type. In contrast, a transformation generates type class instances that perform a deep copy of an object graph, applying transformations to objects of selected types. While Sections 4-5 were focussed on generic queries, this section provides an overview of generic transformations.

Example Suppose we would like to express a generic transformation, which clones object graphs, except for subobjects of a certain type, which are transformed. An example for such a transformation is a generic "scale" function that scales all integers in an object graph by a given factor. The `self-assembly` library lets us write the "scale" function in two steps: first, the definition of a suitable type class; second, the implementation of a subclass of the library-provided `Transform` class. A suitable type class is easily defined:

```
trait Scale[T] extends Queryable[T, T] {
  def scale(visitee: T): T
}
```

Note that the input and output types of `Queryable` are the same in this case, since `scale` transforms any input object into an object of the same type. The actual transformation is defined as follows:

```
object Scale extends Transform {
  def mkTrees[C <: SContext](c: C) = new Trees(c)

  class Trees[C <: SContext](override val c: C)
    extends super.Trees(c)

  implicit def generate[T]: Scale[T] =
    macro genTransform[T, this.type]
}
```

This transformation is not very interesting yet: it simply creates a deep clone of the input object. To specify how, in our case, integers are scaled, it is necessary to define a custom type class instance:

```
def intScale(factor: Int) = new Scale[Int] {
  def scale(x: Int) = x * factor
  def apply(x: Int, visited: Set[Any]) = scale(x)
}
implicit val intInst = intScale(myFactor)
```

For convenience, we can introduce a generic `gscale` function:

```
def gscale[T](obj: T)(implicit inst: Scale[T]): T =
  inst.scale(obj)
```

`gscale` is then invoked as follows:

```
implicit val inst = intScale(10)
val scaled = gscale(obj)
```

Transformations in `self-assembly` The `genTransform` macro is based on traversals similar to those of generic queries. However, the crucial difference is that the macro generates code to *clone* visited objects (based on techniques used in Scala/Pickling [21]). Interestingly, the implementations of queries and transformations share a substantial number of generic building blocks.

[7] Minimally simplified; the actual code also keeps track of object identities as discussed further below.

7. Generic Properties: Lightweight Pluggable Type System Extensions

In this section we show how our approach supports the definition of lightweight pluggable type system extensions that go beyond object-oriented DGP as discussed in the previous sections. In particular, the `self-assembly` library allows defining generic type-based properties that can be checked by the existing Scala type checker.

The key to support both object-oriented DGP and type properties is the fact that our approach is based on generic programming *at compile time*. In addition to having access to query and transformation facilities provided by the library, users also have (a) access to full static type information and (b) Scala's meta-programming API, enabling one to generatively define such generic type properties.

The enabled language extensions are lightweight in the sense that they cannot extend the existing syntax or change Scala's existing type-checking. Instead, they can be thought of as pluggable type system extensions [4] in that without changing the existing typechecker, additional properties can be checked. As a result, our approach supports extensions such as (transitive) type-based immutability checking, which goes beyond standard DGP.

In the following Section 7.1, we first provide a more precise definition of the supported generic properties. Section 7.2 presents a complete example of a non-trivial generic property, immutable types. Finally, in Section 7.3, we discuss key aspects of our implementation in the `self-assembly` library.

7.1 Generic Properties: Definition

The generic properties supported in `self-assembly` are unary type relations. Oliveira et al. [26] show how to define custom type relations in Scala using implicits (see Section 2.1). However, unary type relations defined using implicits are incapable of expressing properties that depend on structural type information that's inaccessible through simple type bounds. Our approach builds on Oliveira et al.'s foundation, and extends it to deep structural type information using type-safe meta-programming.

In the following, we summarize the definition of type relations using implicits and present a high-level overview of our extensions. We then show how `self-assembly` is augmented with meta-programming facilities in order to enable the definition of deeper structural properties.

Defining Unary Type Relations via Type Classes Using implicits a unary type relation can be defined in Scala using an arbitrary generic type constructor, say, `TC`. A type `T` can be declared to be an element of this relation, by defining an *implicit* of type `TC[T]`:

```
implicit val tct = new TC[T] {}
```

This way, an arbitrary *bounded* unary type relation can be defined. The membership of a type `U` in the relation `TC` can be checked by requiring evidence for it using an implicit parameter:

```
def m[U](implicit ev: TC[U]): ...
```

(Classes, and thereby constructors, can also have such implicit parameters.) Only if there exists an implicit value of type `TC[U]` can an invocation of method `m[U]` be type-checked.

Polymorphic implicit methods allow defining a certain class of unbounded type relations by returning values of type `TC[V]` for an arbitrary type `V` that satisfies given type bounds. For example, the following implicit method declares all types that are equal to or subtypes of type `Person` to be elements of relation `TC`:

```
implicit def belowPerson[S <: Person]: TC[S] =
  new TC[S] {}
```

However, without meta-programming the domain of the relation can only be restricted using type bounds; this is not enough for rich properties such as immutability since it requires deep checking to determine whether fields are re-assignable or not.

More Powerful Type Relations via Type-Safe Meta-Programming We extend the above-described type class-based approach so as to be able to define relations that take deep structural type information into account. Our approach provides the following benefits for library authors defining new type relations (such as the immutable property):

1. Library authors are provided with a safe, read-only view of the static type info corresponding to types we test for membership in the relation. The provided type information is not restricted to subtyping tests, rather, all functionality for analyzing type information is provided by Scala's meta-programming API.
2. Boilerplate for library authors is minimized using the generation approach that we outlined in Section 4.2.2. Analogous to queries and transformations, the `self-assembly` library provides a set of reusable abstractions, in turn making the generation mechanism easily accessible to library authors.

Safety Static meta-programming has a reputation for being ad-hoc, untyped, and "anything-goes." However, in our approach the use of macros is fairly restricted. First, we restrict ourselves to a type-safe subset of Scala's macro system (except for a small trusted core), and macro implementations are guaranteed to conform to their type signatures. As a result, these macros are easy to reason about and are well-behaved citizens in the tooling ecosystem. Second, and perhaps most importantly, the `self-assembly` library encapsulates all code generation capabilities internally; library authors defining new generic properties are provided with only a very restricted API. The API is limited to a read-only view of static type information and the possibility to define a predicate on this information controlling type class instance generation.

7.2 Example: Immutable Types

This section presents a complete example of a generic property as defined by a library author using `self-assembly`: a type property for deep immutability. The implementation of this property is shown in Figure 8.

The goal of the defined generic property is to traverse the full structure of a given type, and to ensure (a) that there are no re-assignable fields and (b) that all field types satisfy this property recursively. Therefore, the property is guaranteed *transitively* (all reachable objects are immutable). To guard against subclasses with re-assignable fields, the implementation assumes references of non-final class type potentially refer to mutable objects.

Elements like trait `Property` and the `genQuery` macro are provided by the library. The idea is that when the `genQuery` macro derives an instance of `Immutable[T]` it (a) creates an instance of class `Trees` at compile time, and (b) uses this to check that type T (accessible at compile time as `tpe`) does not contain re-assignable fields (`vars`) and it is possible to derive `Immutable` instances for all its fields (in turn guaranteeing that they are all deeply immutable).

The example also shows that it is possible to add custom type class instances manually (in the example, for types `Int` and `String`). In general, this means that the checks of the generic property can be overridden for specific types. While providing an escape hatch (*e.g.,* in situations where lightweight static checking is not powerful enough to prove a desired property for some type), this capability can also be used to subvert the checking of the generic property, of course. However, existing type checking of the Scala compiler remains unaffected in all cases.

7.3 Generic Properties as Implemented in `self-assembly`

The `self-assembly` library implements generic properties as extensions of generic queries. Note that library authors defining new type properties are not exposed to the implementation discussed in the following.

```
trait Immutable[T] {}

object Immutable extends Property[Unit] {
  def mkTrees[C <: Context with Singleton](c: C) =
    new Trees(c)

  class Trees[C <: Context with Singleton]
    (override val c: C) extends super.Trees(c) {
    def check(tpe: c.Type): Unit = {
      import c.universe._

      if (tpe.typeSymbol.isClass &&
          !tpe.typeSymbol.asClass.isFinal &&
          !tpe.typeSymbol.asClass.isCaseClass) {
        c.abort(c.enclosingPosition, """instances
of non-final or non-case class not
guaranteed to be immutable""")
      } else {
        // if tpe has var, abort
        val allAccessors =
          tpe.decls collect {
            case sym: MethodSymbol
              if sym.isAccessor ||
              sym.isParamAccessor => sym }
        val varGetters =
          allAccessors collect {
            case sym if sym.isGetter &&
              sym.accessed != NoSymbol &&
              sym.accessed.asTerm.isVar => sym }
        if (varGetters.nonEmpty)
          c.abort(c.enclosingPosition,
                  "not immutable")
      }
    }
  }

  implicit def generate[T]: Immutable[T] =
    macro genQuery[T, this.type]

  implicit val intIsImm: Immutable[Int] =
    new Immutable[Int] {}

  implicit val stringIsImm: Immutable[String] =
    new Immutable[String] {}
}
```

Figure 8. Deep immutability checking using `self-assembly`

Let us consider a sketch of `self-assembly`'s implementation of the simple generic `Property` trait used in the previous example:

```
trait Property[R] extends AcyclicQuery[R] {
  abstract class Trees[C <: SContext]
    (override val c: C) extends super.Trees(c) {
    def check(tpe: c.Type): Unit
    override def delimit(tpe: c.Type) = {
      check(tpe)
      (reify({}), reify({}), reify({}))
    }
    ...
  } }
```

The trait introduces a new abstract `check` method that must be implemented by the library author who wishes to define concrete properties such as `Immutable[T]` above. Moreover, the `delimit` method that the generic query invokes for all types encountered in a traversal is overridden to invoke the user-defined `check` method. Otherwise, `delimit` only returns trivial expression trees, since they are (essentially) unused.

8. Implementation and Case Study

We have implemented our approach in the `self-assembly` Scala library.[8] The library has been developed and tested using the current stable release of Scala version 2.11. No extension of the Scala language or compiler is required by the library. The library is comprised of $\approx 1,150$ LOC.

[8] See `https://github.com/phaller/selfassembly`.

Case Study: Scala Pickling To evaluate both expressivity and performance, we have ported an industrial- strength serialization framework, called Scala/Pickling [21], to `self-assembly`[9].

Scala/Pickling is a popular open-source project; on the social code hosting platform GitHub, the project has more than 360 "stars". To achieve its high performance, Scala/Pickling leverages macros for compile-time code generation. Our port of Scala/Pickling to `self-assembly` supports already about 90% of the features of the original; notably, subtyping, object identity, separate compilation, and pluggable pickle formats. Currently, the port lacks picklers based on run-time reflection.

Framework	Performance Change	LOC reduction
Scala/Pickling	$< 1\%$	56%

Table 1. Results of porting Scala/Pickling to `self-assembly`

In terms of efficiency, `self-assembly` compares favorably to the original library: execution time of the "Evactor" benchmark [21] remains within 1% of Scala/Pickling. At the same time, the `self-assembly`-based code is significantly simpler, shorter, and more maintainable. The use of `self-assembly` reduced the code size for macro-based type class instance generation by about 56%.

9. Related Work

DGP in Functional Languages The idea of DGP originated in the Functional Programming community. There are several approaches for writing datatype-generic programs. Early approaches were based on programming languages with built-in support for DGP. These approaches include PolyP [16], and Generic Haskell [9]. Later approaches were based on small language extensions for general purpose languages like Haskell. Examples include Scrap Your Boilerplate [18], Template Haskell [31] and Generic Clean [2].

More recently, researchers have realized that by using advanced type system features DGP could be implemented directly as libraries. Extensive surveys of various approaches to DGP in Haskell (mostly focused on libraries) document various approaches [15, 29]. A large majority of these library based approaches use *run-time* type representations, as well as, isomorphisms that convert between specific datatypes and generic type representations. Without further optimizations this has a significant impact on performance. To improve performance several approaches use techniques such as partial-evaluation [3] or inlining [20]. Approaches based on partial-evaluation require language support, which makes them more difficult to adopt. Inlining is simpler to adopt since it is readily available in many compilers. Good results optimizing some generic functions have been reported in the GHC compiler. However inlining is not very predictable and some generic functions do not optimize well.

Approaches that use meta-programming techniques like Template Haskell (TH) [1] to do DGP are closest to our work. The use of TH is very often motivated by performance considerations, to avoid the costs of run-time type representations. However, published proposals using TH are based on its *untyped* macro system. (TH itself has recently been upgraded to allow type-safe macros.) Although type errors are still detected at compile time even using the untyped system, they are given in terms of the generated code instead of the macro code. In `self-assembly` we do not need to make such a trade-off, because we only use the type-safe subset of Scala's macros (apart from a small, internal trusted core, as is common in DGP approaches).

In contrast to `self-assembly` none of the functional DGP approaches deal with OO features like subtyping or object identity.

[9] `https://github.com/phaller/selfassembly/tree/master/src/main/scala/selfassembly/examples/pickling`

DGP in OO Languages Adaptive Object-Oriented Programming (AOOP) [19] can be considered a DGP approach. In AOOP there is a domain-specific language for selecting parts of a structure that should be visited. This is useful to do traversals on complex structures and focus only on the interesting parts of the structure relevant for computing the final output. DJ is an implementation of AOOP for Java using reflection [27]. More recently, inspired by AOOP, DemeterF [7] improved on previous approaches by providing support for safe traversals, generics and data-generic function generation. Compared to `self-assembly` most AOOP approaches are not type-safe. Only in DemeterF a custom type system was designed to ensure type-safety of generic functions. However DemeterF requires a new language and it is unclear wether issues like object identity are considered, since they take a more functional approach than other AOOP approaches. DemeterF is a language approach to DGP (much like Generic Haskell, for example); whereas we view `self-assembly` as a library based approach.

There has also been some work porting existing functional DGP approaches to Scala. Moors et al. [22] did a port of "origami"-based DGP [14]. Oliveira and Gibbons [25] picked up on this line of work and have shown how several other DGP approaches can be ported and improved in Scala. In particular they have shown some approaches that for doing DGP with type classes, which has a similar flavour to `self-assembly`. However none of these ports attempt to deal with OO features like subtyping or object identity. Moreover all approaches are based on run-time type representations, which is in contrast to our compile-time approach.

Pluggable Type Systems and Language Extensions There are several approaches for providing pluggable type system extensions for statically-typed OO languages [8, 10, 28], but unlike `self-assembly`, they do not provide DGP capabilities. Furthermore, `self-assembly` provides LLEs, which can cannot extend program syntax (like, *e.g.,* SugarJ [12]) or change Scala's built-in type checking.

Our approach is in some sense complementary to staging for embedded DSLs (*e.g.,* , LMS [30]): however, rather than providing staged expressions that are type-checked by the host language, we piggy-back on a macro system for the definition of new type relations. Implicit macros generate type class instances, which, in turn, refine type-checking of *unstaged* programs in the host language. Furthermore, `self-assembly` doesn't require any extensions to the host language.

10. Conclusion

This paper shows a general mechanism, called `self-assembly`, for lightweight language extensions. This mechanism has the extensibility and customization advantages of type classes; and it has the automatic implementation advantages of mechanisms like Java's serialization mechanism. The key idea is to provide automatic implementations of type classes using type-safe macros. This allows programmers to define their own generic functionality, such as serialization, pretty printing, or equality; and it also allows the definition of generic properties such as immutability checking. To demonstrate the usefulness of `self-assembly` in practice, we implemented an industry-ready serialization framework for Scala.

References

[1] M. D. Adams and T. M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In *Haskell'12*, 2012.

[2] A. Alimarine and M. J. Plasmeijer. A generic programming extension for clean. In *IFL '02*, 2002.

[3] A. Alimarine and S. Smetsers. Efficient generic functional programming. Technical report NIII-R0425, Nijmegen Institute for Computing and Information Sciences, University of Nijmegen, 2004.

[4] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[5] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Scala'13*, 2013.

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

[7] B. Chadwick and K. Lieberherr. Weaving generic programming and traversal performance. In *AOSD'10*, 2010.

[8] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI'05*, pages 85–95, 2005.

[9] D. Clarke and A. Löh. Generic haskell, specifically. In J. Gibbons and J. Jeuring, editors, *Generic Programming*, IFIP, pages 21–47. Kluwer Academic Publishers, 2003.

[10] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller. Building and using pluggable type-checkers. In *ICSE'11*, pages 681–690, 2011.

[11] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP'07*, 2007.

[12] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOSPLA'11*, 2011.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesly, Reading, MA, 1995.

[14] J. Gibbons. Design patterns as higher-order datatype-generic programs. In *WGP '06*, 2006.

[15] R. Hinze, J. Jeuring, and A. Loeh. Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*, volume 4719. Springer Berlin/Heidelberg, 2007.

[16] P. Jansson and J. Jeuring. Polyp - a polytypic programming language extension. In *POPL '97*, 1997.

[17] R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *GPCE '06*, 2006.

[18] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *TLDI'03*, 2003.

[19] K. J. Lieberherr. *Adaptive Object Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing, 1996.

[20] J. P. Magalhães, S. Holdermans, J. Jeuring, and A. Löh. Optimizing generics is easy! In *PEPM '10*, 2010.

[21] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *OOPSLA'13*, 2013.

[22] A. Moors, F. Piessens, and W. Joosen. An object-oriented approach to datatype-generic programming. In *WGP '06*, 2006.

[23] D. R. Musser and A. A. Stepanov. Generic programming. In *ISAAC '88*, 1989.

[24] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *IARCS'09*, 2009.

[25] B. C. d. S. Oliveira and J. Gibbons. Scala for generic programmers. *Journal of Functional Programming*, 20(3,4):303–352, 2010.

[26] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA'10*, 2010.

[27] D. Orleans and K. J. Lieberherr. DJ: Dynamic adaptive programming in Java. Springer-Verlag, 2001.

[28] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA'08*, 2008.

[29] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell'08*, 2008.

[30] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6), 2012.

[31] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In *Haskell '02*, 2002.

[32] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci*, 248(1-2):211–242, 2000.

[33] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, 1989.