Composition of languages embedded in Scala

Haeri, Hossein; Keir, Paul

Published in:
Proceedings of the 2019 Federated Conference on Computer Science and Information Systems

Accepted/In press: 26/06/2019

Document Version
Peer reviewed version

Composition of Languages Embedded in Scala

Seyed H. HAERI (Hossein)
Université catholique de Louvain, Belgium
hossein.haeri@ucl.ac.be

Paul Keir
University of the West of Scotland, UK
paul.keir@uws.ac.uk

***Abstract*—Composition is amongst the major challenges faced in language engineering. Erdweg et al. offered a taxonomy for language composition. Mernik catalogued the use of the Language Definitional Framework LISA for composition sorts in that taxonomy. We produce a similar catalogue for embedded language engineering in Scala.**

We begin with techniques that are not specific to Scala. They are applicable in any host language with a module system and support for higher order functions. We, then, present two more techniques to examine Scala-specific language engineering. Interestingly enough, even though dealing with embedded languages, in terms of lines of code, our material is of comparable length to its LISA counterpart. Our work lends insight into Scala's serviceability for composition, as a host for embedded language engineering.

I. INTRODUCTION

a) Language composition is a piece of reality!: Everyday, there are new programming languages that are born by combining ideas from older languages. Inspiration aside, that is an act of composition in many cases. For example, roughly put, Scala adds functional programming and ML modules with mixin composition to Java; which, in return, is C++ without pointers; which, in return, is C with OOP.

The taxonomy of Erdweg et al. [8] suggests a terminology and a formalism for describing such compositions. According to them, one can formalise our Scala description as:

$$\text{Scala} \cong \text{C} \lhd \text{C++} \rhd \text{Java} \lhd (\text{MLModule} \uplus \text{Mixin}) \quad (1)$$

b) Observations from Chemistry: Consider the reaction:

$$H_2SO_3 + 2 \times NaOH \longrightarrow 2 \times H_2O + Na_2SO_3 \quad (2)$$

In Chemistry, two key ingredients for success in the study of such equations are: ($\mathbf{CI}_1$) the availability of substances as the *subjects of study*, and, ($\mathbf{CI}_2$) knowledge about *how* to perform a desirable composition. In reaction (2), for instance, both substances H_2SO_3 and $NaOH$ need to be available. One also needs to know how to double $NaOH$ for the equation balance to be right. Also, how to add $NaOH$ to H_2SO_3 (like the rate of addition, proper temperature, etc.) needs to be known.

c) Programmatic Availability & Composition: The study of formulae like equation (1) determines the precise relative position of languages. Using the outcome, one would be able to add, for example, what is missing in equation (1) so that the "$\cong$" can be replaced by an "$=$". One would also gather that the left-out "FP$\uplus$" is necessary right before MLModule for the balance to be right. Such manipulations are similar to adjusting coefficients in reaction (2) to obtain a balance. Similar to Chemistry, two key ingredients become noticeable here: ($\mathbf{PLI}_1$) *programmatic* availability of programming languages themselves and their belongings as the subjects of study, and, ($\mathbf{PLI}_2$) knowledge about how to *programmatically* obtain desirable language compositions.

By the time of this writing, (mainstream) languages are next to inaccessible as programmatic entities. The study of programmatic language composition, nonetheless, can be conducted independently using, say, contrived languages. That is how this paper tries to gain ($\mathbf{PLI}_2$).

d) Contributions: We demonstrate three techniques for composing languages embedded in Scala. The first (Section II) is applicable in any host language with a module system and support for higher order functions. The second (Section III) is based on Lightweight Modular Staging (LMS) [32]. And, the third – which is also a new solution to the Expression Problem (EP) [5], [30], [39] – employs (possibly restricted) abstract types. The trick in our third technique is promoting the cases of Algebraic Data Types (ADTs) into their own ADT-parameterised standalone components. We showcase each technique using the example compositions of Mernik [19]. We, then, compare the three techniques for their success in addressing the EP concerns (Section V). A discussion about the related work also comes at Section VI.

e) Coding Conventions: This paper assumes familiarity with Scala. For each showcase, the syntax and semantics come in separate packings called `syntax` and `semantics`, respectively. Due to space restrictions, in our code, the name of the showcase is only appended as a comment to the end of the first line of the respective `syntax` or `semantics`. For the same reason, our code is also otherwise unusually compressed. Whilst the showcases are referred to in the prose in CamlCase, their respective Scala package (containing the showcase's `syntax` and `semantics`) is named `like_this` or abbreviated as `lt`.

II. SCALA-UNSPECIFIC

Erdweg et al. catalogue five different ways languages can be composed: language extension, language restriction, language unification, self-extension, and extension composition. Mernik offers simple DSLs to showcase those ways in LISA. In this section, we employ Mernik's simple DSLs for the same purpose, albeit in Scala.

A. Language Extension

A base language B is said to be extended to a language E when the description of B is amended with a description fragment to get E. Erdweg et al. denote that by $B \lhd E$. Consider the language Robot below (packaged under the name `robot` in Scala) for a robot arm that takes commands for moving one unit to either of the four 2D directions. The semantics of Robot involves updating the arm's `position` (recorded in terms of the x and y coordinates) based on the commands (lines 11 to 16).

```
1  object syntax {//robot
2    class Command
3    case object Left  extends Command
4    case object Right extends Command
5    case object Up    extends Command
6    case object Down  extends Command
7    case class Commands(s: Seq[Command])}
8  object semantics {import syntax._//robot
9    class Position(var x: Int, var y: Int)
10   object position extends Position(0, 0)
11   def locate: Command => Unit = {
12     case Left  => position.x -= 1
13     case Right => position.x += 1
14     case Up    => position.y += 1
15     case Down  => position.y -= 1}
16   def locate(cs: Commands) = cs.s.foreach(locate) }
```

Robot is extended to RobotTime (the `robot_time` package) by adding to the semantics, i.e., Robot $\lhd$ RobotTime:

```
1  package robot_time
2  import robot._; import syntax._
3  def time(cs: Commands): Int = cs.s.length
```

Assuming that executing each command takes one time unit, the total time required for a set of commands is the size of the set. The method `time` in line 3 above **add**s that piece of semantics to Robot to get RobotTime. whereas `Commands(Right, Down, Down)` in Robot has only got the semantics $x = 1, y = -2$, it **also** has the semantics $t = 3$ in RobotTime. (The coordinates are obtained by `locate` in line 16 of `robot` and the timing by line 3 of `robot_time`.)

Here is a difference between our implementation of RobotTime and that of Mernik: The latter is done in LISA: a Language Definitional Framework (LDF) that combines OOP with Attribute Grammars [17], [28]. As such, LISA's counterpart for `time` has to visit all the grammatical rules in Robot to attribute the new piece of semantics to them. On the contrary, Scala gave us the joy of simply equating `time` by the number of the commands, regardless of the grammatical rules involved.

B. Language Restriction

A base language B is said to be restricted to a language R when certain parts of the B's features are removed upon transition to R. This is denoted by $B \rhd R$. A typical usage of that is when a language is narrowed to a core of it. That is, certain parts of the base syntax are cancelled into combinations of other base syntactic parts that are deemed to be equivalent. For example, both GPH [37] and Utrecht HASKELL [7] are developed like that.

The language RobotPositive below (packaged under `robot_positive`) restricts Robot to only `Up` and `Right` commands. (Technically, the object `syntax` below is not required. Yet, we retain it for completeness.)

```
1  object syntax {//robot_positive
2    import robot.syntax.{Right, Up, Commands}}
3  object semantics {//robot_positive
4    import robot.syntax.{Right, Up, Commands}
5    import robot.semantics.position
6    def locate(cs: Commands) {for(c <- cs.s) c match {
7      case Right => position.x += 1
8      case Up    => position.y += 1 } } }
```

Any attempt to use the expression in the previous section under RobotPositive will fail to compile for the availability of `Down` in it, which is absent in RobotPositive. On the other hand, `Commands(Right, Up, Up)` has the semantics $x = 1, y = 2$ under RobotPositive.

C. Language Unification

Erdweg et al. say two languages L_1 and L_2 are unified to L when both L_1 and L_2 make sense independently from one another and from L (as the composition's outcome). Furthermore, in L, neither L_1 nor L_2 should be dominated by the other so that a concept of equity prevails in the composition. The notation is $L = L_1 \uplus_g L_2$, where g is the so-called glue code required for the composition.

Having seen the language Robot, we now consider the language ExprAdd (packaged under `expr_add`): a simple ADT with two cases for natural numbers and addition.

```
1  object syntax {//expr_add
2    class Expr {//Expr ::= Expr + Term | ...
3      def + (t: Term): Expr = Add(this, t) }
4    class Term extends Expr//Expr ::= ... | Term
5    case class Num(n: Int) extends Term//Term ::= n
6    case class Add(left: Expr, right: Term) extends Expr}
7  object semantics {import syntax._//expr_add
8    def value: Expr => Int = {
9      case Num(n)    => n
10     case Add(e, t) => value(e) + value(t)} }
```

Using `value` in line 8 above, one obtains the semantics 5, 12, and 6 for the expressions `Num(5)`, `Num(10)+ Num(2)`, and `Num(1)+ Num(2)+ Num(3)`, respectively.

The language RobotUniExprAdd below (packaged under `robot_uni_expr_add`) unifies Robot and ExprAdd by allowing the robot arm to take commands for moving as many units to either of the four directions as the corresponding ExprAdd argument evaluates to. As such, `Commands(Right(Num(5)), Up(Num(2)+ Num(10)), Up(Num(2)+ Num(2)+ Num(2)), Down(Num(4)))` has the semantics $x = 5, y = 14$. Check `locate` in line 11 below.

```
1  object syntax {//robot_uni_expr_add
2    import robot.syntax.Command; import expr_add.syntax._
3    case class Left(e: Expr)  extends Command
4    case class Right(e: Expr) extends Command
5    case class Up(e: Expr)    extends Command
6    case class Down(e: Expr)  extends Command}
7  object semantics {//robot_uni_expr_add
8    import robot.{syntax.Commands, semantics.position}
9    import robot_uni_expr_add.syntax._
10   import expr_add.semantics._
11   def locate(cs: Commands) {for(c <- cs.s) c match {
12     case Left(e)  => position.x -= value(e)
13     case Right(e) => position.x += value(e)
14     case Up(e)    => position.y += value(e)
15     case Down(e)  => position.y -= value(e)} } }
```

D. Self Extension

This is the situation when the description of a language L itself is used for extending it. Typically, embedded DSLs self-extend their host language. For example, all the languages we present in this paper self-extend Scala.

Like Mernik, we believe that demonstrating self extension takes much more than the volume of a single research paper. This is because bootstrapping a language L to the level where it can handle self extension is already more involved than that volume. Hence, we too drop demonstration of self extension.

E. Extension Composition

Extension composition is when (both or at least one of) the language descriptions that are to be composed are themselves compositions of other language descriptions. As such, extension composition can be regarded as higher order composition. Six combinations of extension and unification are possible (three distinguished by Mernik):

1) Double-Unification ($\uplus\uplus$): $L_1 \uplus_g (L_2 \uplus_h L_3)$.
2) Double-Extension ($\lhd\lhd$): $B \lhd E_1 \lhd E_2$.
3) Extension by a Unification ($\lhd\,(\uplus)$): $B \lhd (L_1 \uplus L_2)$.
4) Extension of a Unification ($(\uplus)\,\lhd$): $(L_1 \uplus L_2) \lhd E$.
5) Unification with an Extension ($\{\uplus, (\lhd)\}$): $L \uplus (B \lhd E)$ or $(B \lhd E) \uplus L$. Note the symmetry.

We now consider each combination.

1) Double-Unification ($\uplus\uplus$): To that end, we begin by presenting Mernik's language Dec (packaged under `dec`) in Scala. Dec enables the programmer to bind a set of variables to integer constants.

```
1  object syntax {//dec
2    case class ConstDefList(ds: Map[String, Int]) }
```

Unsurprisingly, the (Scala-automatic) semantics of `ConstDefList("a" -> 5, "b" -> 10)` is then $\{a \mapsto 5, b \mapsto 10\}$.

With that, we illustrate the first class of Mernik's extension compositions using RobotUniExprAddUniDec (packaged under `rueaud`). As suggested by its name, this language is (Robot$\uplus$ExprAdd)$\uplus$Dec. The Robot$\uplus$ExprAdd portion is already presented. See `robot_uni_expr_add` in Section II-C. We now show how to obtain the remaining unification.

```
1  import expr_add.syntax.{Expr, Term}
2  object syntax {//rueaud
3    import robot.syntax.Commands; import dec.syntax._
4    implicit class CDLInCs(val cdl:ConstDefList) {
5      def in (s: Commands) = {
6        consts = cdl.ds; new EnvComm(cdl.ds, s) } }
7    class EnvComm(val ds: Map[String, Int],
8                  val cs: Commands)
9    var consts: Map[String, Int] = Map()
10   case class Var(n: String) extends Term}
11 object semantics {//rueaud
12   import syntax._; import robot_uni_expr_add.syntax._
13   import robot.semantics._
14   def value_ext: (Expr, Expr => Int) => Int = {
15     case (Var(n), c) => consts(n)
16     case (e, c)      =>
17       expr_add.ext_semantics.value_ext(e, c)}
18   def value(e: Expr): Int = value_ext(e, value)
19   def locate(r: EnvComm) {r.cs.s.foreach {
20     case Left(e)  => position.x -= value(e)
21     case Right(e) => position.x += value(e)
22     case Up(e)    => position.y += value(e)
23     case Down(e)  => position.y -= value(e)} }
```

`rueaud.syntax` aims at reusing the former language descriptions as they are. To that end, it takes a *pimp my library* approach [22] on trying to implicitly (lines 4 to 8 above) give instances of `dec.ConstDefList` the extra feature of being followed by commands possibly referring to the declarations. Such declarations followed by expressions are then instances of `EnvComm`. The variable `consts` (line 9) is where the processed declarations are stored. The new ADT case `Var` (line 10) is for looking up the value a name is bound to. `rueaud` legitimises commands for moving the robot arm as many units as a pertaining expression evaluates to (lines 20 to 23). Note that, because of `Var`, those expressions can refer to declarations as well. All that together gives `ConstDefList ("a" -> 5, "b" -> 10)in Commands(Right(Var(" a")), Up(Num(2)+ Var("b")), Down(Num(4)))` the semantics $x = 5, y = 8$ in RobotUniExprAddUniDec.

Instead of reusing `expr_add.semantics.value`, the `rueaud.semantics.value` method uses the method `expr_add.ext_semantics.value_ext`, which will be explained shortly. This is because the former is closed on the set of ADT cases it can handle. Hence, we resort to the following *extensible* semantics of ExprAdd:

```
1  object ext_semantics {import syntax._
2    def value_ext: (Expr, Expr => Int) => Int = {
3      case (Num(n), c)    => n
4      case (Add(e, t), c) => c(e) + c(t)}
5    def value(e: Expr): Int = value_ext(e, value)}
```

In the fashion of $\gamma\Phi C_0$ [13], `value_ext` above takes a continuation argument `c` (line 2), which caters postponing the closing time until the appropriately complete shape [16] of the ADT is known (line 5 above for `expr_add` and line 18 for `rueaud`). As such, extending RobotUniExprAdd to RobotUniExprAddUniDec here involves manipulating the former. See Section V for more.

2) Double-Extension ($\lhd\lhd$): The idea in RobotTimeSpeed below (packaged under `robot_time_speed`) is to enable the user to instruct the robot arm with the speed for its subsequent moves, until further notice. It adds a pertaining command to RobotTime to obtain Robot $\lhd$ RobotTime $\lhd$ RobotTimeSpeed.

```
1  object syntax {//robot_time_speed
2    import robot.syntax.Command
3    case class Speed(i: Int) extends Command}
4  object semantics {//robot_time_speed
5    import syntax._; import robot.syntax.{Command, Commands}
6    import robot.semantics.position
7    def locate: Command => Unit = {
8      case Speed(_) => {}
9      case c        => robot.semantics.locate(c)}
10   def locate(cs: Commands) = cs.s.foreach(locate)
11   var speed: Double = 1.0
12   def time(cs: Commands): Double = {
13     var sum: Double = 0.0; for(c <- cs.s) c match {
14       case Speed(i) => speed = i
15       case _        => sum += (1.0 / speed)}
16     sum} }
```

The new command for altering speed is `Speed` in line 3 above. This new command has no impact on the arm's position, as manifested in line 8. It is in the time calculation where, once used, the related variable (i.e., `speed` in line 11) is updated accordingly (line 14) and taken into consideration

for subsequent commands (line 15). `Commands(Up, Speed(2), Right, Left)` has the semantics $x = 1, y = 0, t = 2$ in RobotTimeSpeed.

3) Extension by a Unification ($\triangleleft$ ($\uplus$)): We now demonstrate RobotExtExprAddUniDec = Robot $\triangleleft$ (ExprAdd$\uplus$Dec). We begin by ExprAddUniDec (packaged under `eaud`):

```
1  import expr_add.syntax._
2  object syntax {import dec.syntax._//eaud
3    class EnvExpr(val ds: Map[String, Int], val e: Expr)
4    implicit class CDL2CDLInE(val cdl: ConstDefList) {
5      def in (e: Expr) = {
6        consts = cdl.ds; new EnvExpr(cdl.ds, e)} }
7    var consts: Map[String, Int] = Map()
8    case class Var(n: String) extends Term}
9  object semantics {//eaud
10   import syntax._; import dec.syntax._
11   def value_ext: (Expr, Expr => Int) => Int = {
12     case (Var(n), c) => consts(n)
13     case (e, c) => expr_add.ext_semantics.value_ext(e, c)}
14   def value(e: Expr): Int = value_ext(e, value)
15   def value(ee: EnvExpr): Int = value(ee.e)}
```

`eaud` is similar to `rueaud` in Section II-E1 and we drop further explanation. RobotExtExprAddUniDec below (packaged under `reeaud`) tries to make use of `eaud`.

```
1  object syntax {//reeaud
2    import dec.syntax._; import robot.syntax.Commands
3    class EnvComm(val ds: Map[String, Int],
4                  val cs: Commands)
5    implicit class CDL2CDLInC(val cdl: ConstDefList) {
6      def in (s: Commands) = {
7        consts = cdl.ds; new EnvComm(cdl.ds, s)} }
8    var consts = eaud.syntax.consts}
9  object semantics {import robot.semantics.position//reeaud
10   import robot_uni_expr_add.syntax._
11   import eaud.semantics.value; import syntax._
12   def locate(r: EnvComm) { r.cs.s.foreach {
13     case Left(e)  => position.x -= value(e)
14     case Right(e) => position.x += value(e)
15     case Up(e)    => position.y += value(e)
16     case Down(e)  => position.y -= value(e)} } }
```

Here are the few idiosyncrasies of `reeaud`: Firstly, `reeaud` fails to reuse most of the syntactic facilities of `eaud`. This is because the former employs declarations followed by commands, whereas the latter employs declarations followed by expressions. In line 8, nevertheless, `consts` is reused. Secondly, even though RobotExtExprAddUniDec = Robot $\triangleleft$ ..., in `reeaud.semantics`, we do not reuse `robot.syntax`. On the contrary, in line 10, it reuses the syntax of `robot_uni_expr_add` (for RobotUniExprAdd). This is because, in Robot, it is only possible to move the arm one unit to either direction. The Scala syntax for those two pieces of (embedded) syntax cannot coexist side by side. See Section III-A2 for more.

`reeaud.semantics.locate` is similar to `rueaud.semantics.locate`. In RobotExtExprAddUniDec, `ConstDefList("a" -> 5, "b" -> 10)in Commands(Right(Var("a")), Up(Num(2)+ Var("b")), Down(Num(4)))` has semantics $x = 5, y = 8$.

As pointed out by Mernik, so long as functionality is the only concern, RobotUniExprAddUniDec $\equiv$ RobotExtExprAddUniDec. The difference, both in LISA and Scala, is in the language descriptions, and the combinations by which they are obtained. Unlike its LISA counterpart, nonetheless, obtaining RobotExtExprAddUniDec in Scala involves intermediate material that is not reused in the final product.

4) Extension of a Unification (($\uplus$) $\triangleleft$): RobotUniExprAddExtRobotTime below (packaged under `rueaert`) extends RobotUniExprAdd (Section II) by a timing facility. The time required for carrying out a command of moving in one direction equals what the pertaining expression evaluates to (lines 7 to 10). The method `time` below is a simple fold operation on the given sequence of commands, based on that explanation. RobotUniExprAddExtRobotTime = (Robot $\uplus$ ExprAdd) $\triangleleft$ RobotTime.

```
1  object syntax {//rueaert
2    import robot_uni_expr_add.syntax._ }
3  object semantics {import expr_add.semantics._//rueaert
4    import robot.syntax.Commands
5    import robot_uni_expr_add.syntax._
6    def time(cs: Commands): Int = (0 /: cs.s){
7      case (s, Left(e))  => s + value(e)
8      case (s, Right(e)) => s + value(e)
9      case (s, Up(e))    => s + value(e)
10     case (s, Down(e))  => s + value(e)} }
```

`Commands(Right(Num(5)), Up(Num(2)+ Num(10)), Up(Num(2)+ Num(2)+ Num(2)), Down(Num(4)))` has the semantics $x = 5, y = 14, t = 27$ in `rueaert`.

5) Unification with an Extension ($\{\uplus, (\triangleleft)\}$): Take RobotUniExprMul = Robot $\uplus$ ExprMul, where ExprAdd $\triangleleft$ ExprMul. The language ExprMul extends ExprAdd by a new ADT case for multiplication (`Mul`). What is unique about ExprMul amongst the visited extension combinations is that, upon extension, it changes the syntactic categories of the ADT cases it borrows from ExprAdd. (And, in fact, it also provides a new syntactic category, i.e., `Factor`.) As presented in Section III-B, this can impose a great deal of complexity when language extension is implemented using inheritance. Here is ExprMul (packaged under `expr_mul`).

```
1  import expr_add.syntax.{Expr, Term, Add}
2  object syntax {//expr_mul
3    class Factor extends Term//Term ::= Factor | ...
4    implicit class TermTimesFactor(val t: Term) {
5        def * (f: Factor): Term = Mul(t, f)
6    }//Term ::= ... | Term * Factor
7    case class Num(n: Int) extends Factor//Factor ::= n
8    case class Mul(left: Term, right: Factor) extends Term }
9  object semantics {import syntax._//expr_mul
10   import expr_add.ext_semantics.{value_ext => add_value}
11   def value_ext: (Expr, Expr => Int) => Int = {
12     case (Num(n), c)    => n
13     case (Add(e, t), c) => add_value(Add(e, t), c)
14     case (Mul(t, f), c) => c(t) * c(f)}
15   def value(e: Expr): Int = value_ext(e, value)}
```

In line 1, ExprMul imports the syntactic entities it borrows from ExprAdd: the ADT case `Add` and the syntactic categories `Expr` and `Term`. It then introduces its new syntactic category `Factor` in line 3. Next, in lines 4 to 6, it provides the syntactic sugar for multiplication. Note how it, afterwards, declares numbers to now be of the category `Factor` – as opposed to `Term` in `expr_add.syntax`. The rest of `expr_mul` should be straightforward except for the Scala syntax of line 10. Those lines abbreviate `expr_add.ext_semantics.value_ext` to `add_value` in `expr_mul.semantics`. In line 13, `expr_mul` reuses `add_value` for the solo ADT case that it borrows from `expr_add`, i.e., `Add`.

```
1  object syntax{import expr_mul.syntax._//robot_uni_expr_mul
```

```
    import robot.syntax.Command; import expr_add.syntax.Expr
    case class Left  (e: Expr) extends Command
    case class Right(e: Expr) extends Command
    case class Up    (e: Expr) extends Command
    case class Down  (e: Expr) extends Command}
object semantics {import syntax._//robot_uni_expr_mul
    import robot.{syntax.Commands, semantics.position}
    import expr_mul.semantics._
    def locate(cs: Commands) = cs.s.foreach {
      case Left(e)  => position.x -= value(e)
      case Right(e) => position.x += value(e)
      case Up(e)    => position.y += value(e)
      case Down(e)  => position.y -= value(e)} }
```

The above implementation of RobotUniExprMul (packaged under `robot_uni_expr_mul`) takes tightly after RobotUniExprAdd (in Section II-C). We, therefore, do not provide a dedicated walk-through. `Commands(Right(Num(5)* Num(2)), Down(Num(4)+ Num(2)* Num(3)))` has the semantics $x = 10, y = -10$ in `robot_uni_expr_mul`.

F. Language Specific?

To investigate the extent to which Scala-specific language features impact upon our design, we intend also to compare against realisations in other languages. To this end, we have prepared a C++ implementation which adopts the Scala approach outlined so far. Respecting the dynamic polymorphism of the Scala original, the C++ implementation utilises `shared_ptr` smart pointer to manage the memory allocation and runtime typing of expressions; allowing the `vector` container member object of the `Commands` class to store different expression types. User-defined integral and string literals also allow a notably concise syntax for the `Num` and `Var` instantiations; e.g., `Commands{Right{"a"_s}, Up{2_n + "b"_s}, Down{4_n}}`. Future work will explore this further.

III. LMS-Based

LMS is the technique Scala puts forward for solving EP. The essence of LMS is the use of Scala traits for extensibility and **super** calls for reuse. With their mixin nature, Scala traits can extend one another, enjoying the benefits of inheritance. In particular, an ADT can be inherited upon trait extension. But, the heir trait can also add its own new ADT cases. On top of that, **super** calls enable reusing methods on the cases of the original ADT. Whereas the new cases can be handled by the same method, albeit overridden by the heir trait.

In the package `eaud` below (for ExprAddUniDec), for implementing both the syntax and semantics, traits are used – as opposed to objects in Section II. Instead of **import**ing members from other languages, it now **extends** those other languages to acquire the same members via inheritance. In Scala terms, `eaud.syntax` is, for instance, said to be mixing in `expr_add.syntax` and `dec.syntax`, in line 1 below.

In line 4, then, `eaud.semantics` overrides `value`. In line 5, it handles the new ADT case `eaud.syntax` introduces. All those other ADT cases that `eaud` inherits are, in line 6, relayed to the upper levels of inheritance.

```
1 trait syntax extends expr_add.syntax with dec.syntax {
2     ... /* like eaud.syntax in Section II-E3 */ ...}
3 trait semantics extends syntax with expr_add.semantics {
4   override def value: Expr => Int = {
5     case Var(n) => consts(n)
6     case e      => super.value(e)} ...}
```

This is how LMS facilitates both simplicity and extensibility. (Note that we needed not to resort to `value_ext`.)

LMS has been successfully employed for languages in a multitude of applications. For the benefits of LMS, the reader is invited to consult those works. Given that we did not come to observe new benefits, we will not get into that here. We rather dedicate this section to the difficulties we faced over employing LMS for embedded language composition.

A. Minor Difficulties

The two categories of minor difficulties we faced relate to language restriction (Section III-A1) and clashes occurred between names upon composition (Section III-A2).

1) Language Restriction: Upon extension, the programmer is usually provided with no means for acting selectively on the members to be inherited. When mixing traits too, **all** the (public or protected) members get inherited automatically. Hence, with inheritance being the means for language composition, language restriction is not possible. That enforces **import** as the fallback. With the use of traits, the mechanics is, however, more involved than Section II. Because traits are abstract, one needs to materialise them first (line 2 below), and only then, they can be **import**ed from (line 2).

```
1 trait syntax/* robot_positive */{val robosyn = new robot.
2   syntax {};import robosyn.{Right, Up, Command, Commands}}
```

Even though LISA also employs inheritance for language composition, this difficulty does not arise there. The reason is as follows: Being also an Attribute Grammars system, (subject) language semantics is specified in LISA by traversing the concrete syntax. On the other hand, leveraging its OOP, LISA allows the heir language to override the parent language's concrete syntax. As a result, language restriction is also possible in LISA via inheritance.

One final related comment: In our experience, enforced **import**s like those required for language restriction were not exclusive to that way of language composition. In fact, in a good number of other occasions, the languages do make selective use of one another. That, on its own, was not a knotty problem. It, however, requires increasingly more care when it comes to interplay with hierarchies of languages and the relevant Scala mixins.

Note that **import**ed names (like those in line 2 above) do not get inherited but the respective materialised traits (like `robosyn` in line 2 above) do. Such **import**s can be required on several occasions down the hierarchy. In the case of unification, however, where the multiple inheritance nature of mixins is employed, an extra **override** might also be enforced to disambiguate duplicated names across the meeting two hierarchies. See Section III-B for more.

2) Name Clash: Recall from Section II-E3 that RobotExtExprAddUniDec $=$ Robot $\triangleleft$ (ExprAdd $\uplus$ Dec). In an LMS-based implementation of RobotExtExprAddUniDec, therefore, one would naturally want to implement `rueaud.semantics` as follows:

```
1 trait semantics extends rueaud.syntax with
2   robot.semantics with eaud.semantics {//rueaud
3     ... /* locate like Section II-E1 */ ...}
```

That is, however, not possible. The error message is: "object `Left` is not a case class, nor does it have an `unapply`/`unapplySeq` member." The problem is that, even though `Left` is inherited from `robot`, in `locate`, Scala would not be able to match it using the syntax `Left(e)`. The available constructor and extractor of `Left` take no arguments. Moreover, overloading that syntax is not possible. This is because Scala desugars both case classes and case objects to objects with `unapply` (or `unapplySeq`) methods. Objects, on the other hand, are final, banning any later manipulation. To proceed, one needs to use `robot_uni_expr_add.semantics` in return of `robot.semantics`.

The problem is harder to diagnose for RobotUniExprAddExtRobotTime. Recall from Section II-E4 that RobotUniExprAddExtRobotTime $=$ (Robot $\uplus$ ExprAdd) $\lhd$ RobotTime. For the attempt

```
trait semantics extends rueaert.syntax with
  robot_uni_expr_add.semantics with
  robot_time.semantics {... /* rueaert */ ...}
```

even when one employs `robot_uni_expr_add.semantics` instead of `robot.semantics`, one gets an error – this time, regarding the composition itself: "overriding object `Left` in trait `syntax`; object `Left` in trait `syntax` cannot override final member." The problem here is with `robot_time` being an extension to `robot`, bringing the case object `Left` into the mix with that of `robot_uni_expr_add` that takes an argument.

B. Major Difficulties

The difficulties we spoke about in the previous subsection were not particularly acute in that not many circumvention attempts would fail for them. In this section, we will report a multi-staged combat with an acute difficulty we faced. In short, the combat was against the combination of Scala's path-dependant typing and intervention of concrete syntax.

The contents of this section might look too specific to Scala. They are not. Scala's path-dependant typing is just one way to foster family polymorphism [9] (as opposed to lightweight family polymorphism [33]). The familiar reader will figure out that the same problem is likely to emerge in every host language that embraces family polymorphism.

Given that ExprMul is a direct extension to ExprAdd, one's first guess would be:

```
trait expr_mul.syntax extends expr_add.syntax {...}
```

That is, however, not possible because, then, `Num` cannot be overridden. Recall from Section II-E5 that ExprMul changes the syntactic category of `Num`. But, even an attempt like those in Section III-A1 for the syntax

```
trait syntax {val easyn = new expr_add.syntax {}//expr_mul
  import easyn.{Expr, Term, Add} /* Num, Factor, etc. */}
```

would still cause failure for the semantics.

```
trait expr_mul.semantics extends syntax with
                                 expr_add.semantics {...}
```

Here is the error message: "overriding object `Num` in trait syntax; object `Num` in trait syntax cannot override final member." This is because of the clash between the `Num` of such a `expr_mul.syntax` and `expr_add.semantics`. See Section III-A2 for an explanation on similar error messages.

Now, let us suppose for the sake of argument that the semantics too selectively **`import`**s the ADT cases:

```
1 trait semantics {//expr_mul
2   val emsyn = new expr_mul.syntax {}
3   import emsyn.{Num, Mul, Factor}
4   val easyn = new expr_add.syntax {}
5   import easyn.{Expr, Add, Term}
6   ... /* value or value_ext here */ ...}
```

Recall that ExprMul adds the ADT case `Mul` to ExprAdd. To reuse – *à la* LMS – the ExprAdd semantics whilst also handling the new ADT case, one may (mistakenly) try:

```
1 override def value: Expr => Int = {
2   case Mul(t, f) => value(t) * value(f)  ...}
```

But, that will not type-check because of path-dependant typing interference: `Expr` in `value`'s signature is different from `Expr` that `Mul` inherits from. Here is the error message for line 2 above: "constructor cannot be instantiated to expected type; found: `semantics.this.emsyn.Mul` required: `semantics.this.Expr`." Even worse: An attempt for reusing the semantics of the only ADT case that remains intact over the move from ExprAdd to ExprMul using `value_ext`

```
1 trait semantics {...//expr_mul
2   import easem.{value_ext => add_value}
3   def value_ext: (Expr, Expr => Int) => Int = {
4     case (Num(n), c)    => n
5     case (Add(e, t), c) => add_value(Add(e, t), c)
6     case (Mul(t, f), c) => c(t) * c(f)} }
```

will again fail due to path-dependant typing. The error message for line 5 above is: "type mismatch; found: `semantics.this.easyn.Add` required: `semantics.this.easem.Expr`."

Given that `expr_mul.semantics` is to reuse pattern matching of `expr_add.semantics`, the former is also bound to the types – here , ADT cases – of the latter. In order to prevent the path-dependant clashes, thus, the only way forward seems to be for **both** `expr_mul.syntax` and `expr_mul.semantics` to import types of `expr_add.semantics`. This is, of course, very unnatural for the former.

```
1 trait syntax {//expr_mul
2   val easem = new expr_add.semantics {}
3   import easem.{Expr, Term, Add}; ...}
4 trait semantics extends syntax {//expr_mul
5   import easem.{Expr, Add, value_ext => add_value};...
6   def value_ext: (Expr, Expr => Int) => Int = {
7     case (Num(n), c) => n
8     case (a: Add, c) => add_value(a, c);...} ...}
```

Still, if not done craftily enough, path-dependant typing can be an impediment. Replacing the line 8 above with

`case (a @ Add(_, _), c)=> add_value(a, c)`

will fail to type-check because `a` is considered to be of type `this.Add`; whereas, `add_value` accepts an `easem.Expr`. The unsightly circumvention would be:

`case (a @ Add(_, _), c)=> add_value(a.asInstanceOf[easem.Expr], c.asInstanceOf[easem.Expr => Int]).`

We would like to remind that all the difficulties illustrated in this section were only experienced in the presence of manipulation in the syntactic categories upon extension.

value: Expr => Int
expr_add{Num, Add}, eaud{Num, Add, Var}, expr_mul{Num, Add, Mul}
locate: Command => Unit (without e)
robot{Right, Left, Up, Down}, robot_positive{Right, Up}
locate (with (e))
in reeaud: EnvComm => Unit
in robot_uni_expr_add: Commands => Unit
in rueaud: EnvComm => Unit
in robot_uni_expr_mul: Commands => Unit
EnvComm
reeaud, rueaud

Fig. 1: Repeated Entities in Sections II and III

Syntactic categories are often used for dealing with concrete syntax. Semantics, on the other hand, inputs abstract syntax. The following section presents a solution that disassociates concrete syntax from abstract syntax. It applies the LMS at the abstract syntax level, and, hence, independently of the concrete syntax that varies across languages. That design sets the different languages free on engineering their syntactic categorisation whilst enjoying the benefits of LMS.

IV. Refactoring

The previous two sections were developed as if the guest language implementer was not aware in advance of the next guest languages and the upcoming combinations. We also maintained a backward compatibility policy in that we did not touch the older languages as we proceeded. Refactoring, however, is common in everyday software development.

Refactoring can have a variety of meanings, depending on the target and the methods used. In this paper, we do not plan extensive refactoring. We only focus on avoiding repetition. Fig. 1 lists a number of repetitions in the previous sections.

We notice that the method `value` is repeated in `expr_add`, `eaud`, and `expr_mul`. More precisely, the ADT cases `Num` and `Add` – which are, basically, inherited from `expr_add` – are handled thrice in the codebase. As will be shown in this section, we gave `value` its own abstraction.

We also notice that the method `locate` is present in two sets of language descriptions: in (i) `robot` and `robot_posit-ive` (when the four direction commands do not take arguments); and, in (ii) `reeaud`, `robot_uni_expr_add`, `rueaud`, and `robot_uni_expr_mul` (when the four direction commands do take arguments). Each of those sets constitutes a candidate for refactoring. Finally, `EnvComm` is common between `reeaud` and `rueaert` – constituting yet another refactoring candidate. Although we have indeed refactored the candidates of this paragraph as well, we will not include their demonstration in this paper. The interested reader can look them up in our online codebase.

Let us now focus on refactoring the first row of Fig. 1. (Refactoring the other rows of Fig. 1 is done similarly.) Here is a succinct summary of actions to be taken: The idea is a combination of LMS and Component-Based Mechanisation [12], [11], [13]. We parameterise the ADT cases `Num`, `Add`, `Var`, and `Mul` by the language description and perform their semantics evaluation independently of the language description. We pack the two former cases – namely, `Num` and `Add` that are common between all the items in the first row of Fig. 1 – together in a trait. Then, we extend that trait for `Var` and later for `Mul`, both *à la* LMS. Finally, the concrete language descriptions only get to mix the respective abstract descriptions. The elaboration follows.

```
1  trait na_syntax {//E for Expr, N for Num, A for Add
2    type E; type N <: E; type A <: E
3    def n_extr(n: N): Option[Int]
4    def a_extr(a: A): Option[(E, E)]
5    object N {def unapply(n: N) = n_extr(n)}
6    object A {def unapply(a: A) = a_extr(a)} }
7  trait na_semantics extends na_syntax {
8    def value: E => Int = {
9      case N(n)       => n
10     case A(e1, e2) => value(e1) + value(e2)} }
```

In `na_syntax` above, the abstract type `E` (in line 2) is a language-independent representation for the expression type of a guest language. Such a guest language can be an item in row 1 of Fig. 1 or any similar language with integer arithmetics that at least contains integral literals and addition. Given that ADTs are implemented in Scala using plain inheritance, two more language-independent abstract types have been employed that are announced to be extending `E`. Those are `N` for `Num` and `A` for `Add`, in line 2.

Because `N` and `A` are supposed to later be instantiated to the respective cases of an ADT, they are expected to come with the Scala matching syntax, like those in lines 9 and 10. The Scala machinery for enforcing availability of the desirable matching syntax requires a discipline in coding that is slightly tricky. The discipline involves, for each ADT case abstract type, inclusion of a same-named (singleton) object – called *companion* object – that ships, then, with an *extractor*, i.e., an `unapply` method of the right type signature. The actual duty of the extractor is relayed to an abstract method, to be enforced to every guest language that implements `na_syntax`. For `N`, for instance, that duty is on `n_extr` in line 3. The Scala signature of `n_extr` means that, **if** matching `N` succeeds, it would be initialising an argument of type `Int`. All that wiring enables the method `na_semantics.value` to handle the semantics of `Num` and `Add`.

```
1  trait nam_syntax extends na_syntax {type M <: E
2    def m_extr(m: M): Option[(E, E)]
3    object M {def unapply(m: M) = m_extr(m)}}
4  trait nam_semantics extends nam_syntax with na_semantics {
5    override def value: E => Int = {
6      case M(e1, e2) => value(e1) * value(e2)
7      case e          => super.value(e)} }
```

The trait `nam_syntax` adds the abstract type `M` (in line 1 above), which corresponds to `Mul`. It also provides the Scala matching syntax in lines 2 and 3. The trait `nam_semantics` reuses (*à la* LMS) what is already implemented by `na_semantics` by performing a **`super`** call on the relevant ADT cases (line 7).

```
1  trait expr_add.syntax extends na_syntax {...
2    //like lines 2 to 6 of expr_add.syntax in Section II...
3    //Fix the ADT type, the Num case, and the Add case.
4    type E = Expr; type N = Num; type A = Add
5    //And, fix the extractors.
6    def n_extr(n: Num) = Num.unapply(n)
```

```
7     def a_extr(a: Add) = Add.unapply(a)}
8   trait expr_add.semantics extends
9     expr_add.syntax with na_semantics
```

In addition to working out the Section II concrete syntax, the trait `expr_add.syntax` above, now is required to provide evidence on it indeed having ADT cases for integral literals and addition. That, again involves some slightly tricky discipline consisting of two steps. First, in line 4, the concrete counterparts for the abstract (ADT case) types in `na_syntax` are fixed. Second, in lines 6 and 7 the extractors promised to `na_syntax` are fixed.

Recall from `expr_add.syntax` of Section II that `Num` and `Add` are both case classes. Scala actually desugars case classes to normal classes in addition to companion objects with the right-typed `unapply` methods. That is why we can use `Num.unapply` and `Add.unapply` off-the-shelf.

Nothing more remains for `expr_add.semantics` to do except inheriting its (abstract and concrete) syntax from `expr_add.syntax` and its semantics from `na_semantics`.

```
1   trait expr_mul.syntax extends nam_syntax {
2     val easyn = new expr_add.syntax {}
3     import easyn.{Expr, Term, Add};...
4     //like lines 3 to 8 of expr_mul.syntax in Section II...
5     type E = Expr; type N = Num; type A = Add; type M = Mul
6     def n_extr(n: Num) = Num.unapply(n)
7     def a_extr(a: Add) = Add.unapply(a)
8     def m_extr(m: Mul) = Mul.unapply(m)}
9   trait expr_mul.semantics extends
10    expr_mul.syntax with nam_semantics
```

Implementing ExprMul, in this fashion, is similar, as demonstrated above. It only is that, like in Section III, our use of traits instead of objects in favour of LMS imposes instantiation of the trait `expr_add.syntax` (line 2) before **`import`**ing the desirable concrete syntax items (line 3).

Remarks

From a semantician's point of view, `na_semantics` packs the morphisms of the **category** of all languages with `Num` and `Int` in the syntax. Although not entirely in the fashion of Modular Structural Operational Semantics (MSOS) [20], this is still very close. $\gamma\Phi C_0$ [13] describes that as: "client *na_semantics*<$F \lhd$ *Int* $\oplus$ *Num*>{...}," where F is the *family parameter* of *na_semantics*. In words, that reads: A family Φ to be substituted for F needs at least to have components *Int* and *Num* (or their equivalents) in its mix.

From another language theoretical viewpoint, `na_syntax` and `na_semantics` are both type classes [40]. From that viewpoint, `expr_add.syntax` is an instance of `na_syntax` and `expr_add.semantics` is an instance of `na_semantics`. The evidence for the former is provided in line 4 in `expr_add.syntax`. Interestingly, however, our encoding of type classes in Scala is not the common one [26]. In particular, we do not prescribe the use of **`implicit`**s.

As also announced at the last paragraph of Section III, `na_syntax` and `na_semantics` (and also `nam_syntax` and `nam_semantics`) relate to the abstract syntax only. This is how they leverage LMS and yet do not suffer from the concrete syntactic anomalies discussed in Section III. Moreover, unlike Modular Reifiable Matching [27], the technique we presented in this section is not exclusively targeting two-level types [34]. The reason is that our technique in this section fully disassociates concrete syntax from the abstract syntax so there no longer is an issue of levels in the types. LMS itself comes with no such separation either – suggesting the name *abstract LMS* for our technique.

It is noteworthy that the disassociation of abstract and concrete syntax with the lack of the LMS anomalies discussed in Section III needs not specifically be *à la* LMS. The same impact can also be achieved using a decentralised pattern matching that is integrated at the right time [14]. The difference is that the abstract LMS composes components (that correspond to ADT cases) *additively* [35, §17.3], whilst the latter technique would be composing them *sequentially*.

The connection between this technique and Component-Based Software Engineering (CBSE) [35, §17],[29, §10] is also interesting. From a CBSE standpoint, `nam_syntax` is a component in that: Without binding to a particular implementation, it specifies its so-called 'requires' and 'provides' interfaces. The `nam_syntax` 'requires' interface is its lines 1 and 2 – imposing the following two requirements, respectively: The user of `nam_syntax` needs to provide a type `M`. And, there has to be a way to extract two expressions of type `E` from an instance of `M`. In return, the 'provides' interface of `M` is its line 3, where `M`'s Scala match syntax (used in line 6 of `nam_semantics`) is offered. As such, `nam_syntax` is promoting the ADT case `Mul` to its standalone component.[1] This is an important characteristic of the third technique that relates to the EP. Next section is dedicated to that relationship.

V. Expression Problem

EP is a recurrent problem in the field of Programming Languages, for which a wide range of solutions have thus far been proposed, e.g., [25], [1], [41]. EP is the challenge of finding an implementation for an ADT – defined by its cases and the functions on it – that: (E1) both new cases and functions can be added; (E2) applying a function f on a statically constructed ADT term t should fail to compile when f does not cover all the cases in t; (E3) upon extension, forces no manipulation or duplication to the existing code; and (E4) compiling the extension imposes no requirement for repeating compilation or type checking of existing code.

In Sections II–IV, we presented three techniques for embedded language composition in Scala. All the three techniques satisfy E4. We now reflect on their E1–E3 competence: The first technique clearly satisfies E1. Section III-A2 outlines a scenario where LMS fails to satisfy E1. Whether the third technique satisfies E1 depends on whether it employs trait mixing for composition or not. Note that it needs not. The three techniques all relax E2, although they can be circumvented to work when defaults are available [23]. That is a consequence of Scala performing pattern matching at runtime. LMS too

[1]Two reasons for not promoting `Num` and `Add` to components: 1) that would complicate presentation. 2) the current design in which those two ADT cases are packed together in a single component (i.e., `na_syntax`) demonstrates how to address the Common Reuse Principle of Martin [18].

relaxes E2 and that has thus far been considered an acceptable setting. (For example, MVCs [24] and Torgersen's second solution [36] both have the same issue.) The state of affairs for LMS might change in future though [31].

As witnessed by RobotUniExprAddUniDec in Section II-E1, the Scala-unspecific technique fails to satisfy E3 when new cases are to be added. As detailed in Section III-B, LMS has to fight path-dependant typing to satisfy E3 when syntactic categories are updated upon composition. Whether there always is a winning strategy for LMS in such a situation is not known. The third technique clearly satisfies E3.

We understand that the path-dependant typing difficulties of the LMS-based technique might indeed be a result of our peculiar design. In particular, our choice of giving the syntax and semantics of a language each a trait of their own might be picked as the root cause. We would like to defend that choice of ours, specifically, for the likelihood of engineering (or experimentation with) more than one semantics for the same syntax [15]. In such cases, separation of the syntax and semantics is inevitable.

Finally, one may wonder whether the third technique makes it to a new solution to EP. The answer is indeed yes. At least for EP in presence of defaults [23]. This is the third EP solution of its kind: It promotes ADT cases to their own ADT-parameterised components. See [12], [11] for the first and [14] for the second EP solution of this kind.

VI. Related Work

a) LISA.: As stated earlier, this paper is highly inspired by Mernik [19]. We essentially took his examples for showing how to compose languages embedded in Scala. With LISA being an LDF, even though Scala is famous for its hospitality to embedded languages, we were surprised to end up having less lines-of-code (LoC) in all the three techniques.

Fig. 2 summarises the LoC comparison. In the LoC there, we have also included some syntactic cosmetics that we did not display in this paper. In our experience, the occasions where Scala outperforms LISA by far are those where the task was a ready cake for GPLs. Examples are RobotTime for all the techniques and RobotExtExprAddUniDec for the third technique. For the former, a simple container size query does the job. For the latter, simple trait mixing does.

The first technique generally performs better (in terms of LoC) than LISA. The second is even better usually with its utilisation of trait mixing (dismissing the obvious `import`s) and `super` calls. At last, the third is the best with its a posteriori refactoring. The two occasions when LISA considerably outperforms Scala are RobotUniExprAdd for the first technique and RobotUniExprMul for the third. Those correspond to Sections III-A2 and III-B, respectively.

The factored out code in the third technique is not counted in Fig. 2. Once that too is added, the total LoC reaches 328 – which is 2 more than first technique's LoC. We tend to think the reason is the simplicity in the semantics of Mernik's examples. That caused the number of lines the refactoring saves to be less than the extra overhead the technique requires. For more realistic case studies, we expect the balance to be completely different. That would be well in favour of refactoring due to reasonably more involved semantics.

b) Other Language Composition Catalogues.: Völter [38] proposes a taxonomy of language composition that he showcases in JetBrains MPS. Barrett, Bolz, and Tratt [2] catalogue composition of six Python and Prolog virtual machines. Zhang et al. [42] facilitate composition of languages that are embedded using Object Algebras [10]. Melange [6] is an LDF that is specially equipped for language composition.

c) Components for Language Implementation.: PLanCompS funcons are syntactic constructs that ship with their own fixed static and dynamic semantics (presented in MSOS). The PLanCompS specification of a programming language is developed by merely assembling funcons [21]. Despite their merit, funcons do not constitute CBSE components. In particular, funcons do not ship with their 'requires' interfaces.

MVCs [24] are components for solving an extension to EP. Rather than components in their CBSE sense, however, MVCs are components in a Component-Oriented Programming sense. MVCs rely on the implementation details of **how** a component realises its interfaces. CBSE components, in contrast, are identified by their 'requires' and 'provides' **interface**s.

Haeri and Schupp [12], [14] take a CBSE approach for the implementation of embedded languages. Their approach employs type constraints and multiple inheritance. The third technique here employs (possibly constrained) abstract types instead of type parameters. Although essentially the same, the former can make code terser. In Scala, however, offering the match syntax is apparently not possible for type parameters.

Finally, Cazzola and Vacchi [4] too have taken a CBSE approach. Their components correspond to a DSL's compiler passes. Accordingly, how their work relates to the common semantic formalisms is not clear.

VII. Conclusions and Future Work

In this paper we present three different techniques for composing languages embedded in Scala. The first is Scala-unspecific and works in presence of common module systems and higher order functions (Section II). The second is LMS-based and requires mixin composition and `super` calls (Section III). The third works by promoting ADT cases to ADT-parameterised components (Section IV). We showcase the three techniques using the example compositions of Mernik, which, in return, were designed to exhibit LISA's composition facilities for Erdweg et al.'s taxonomy of composition. We manifest the strengths and weaknesses of each technique. We compare them according to their performance as EP solutions (Section V) and LoC (Section VI-0a).

Systematic study of embedded language composition is a young topic. Numerous paths exist for future research. Examining our third technique against larger testcases is an immediate future work. Type classes are more widely practised in Haskell. It would be interesting to see our third technique in Haskell, where mixins and inheritance are absent. Object Algebras are gaining gravity as a powerful abstraction for

	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9	L_{10}	L_{11}	L_{12}	L_{13}	Sum
LISA	42	23	13	19	19	32	39	41	20	34	23	20	19	344
T_1	32	7	16	26	34	11	40	25	31	34	17	22	31	326
T_2	30	6	15	20	29	10	34	20	26	28	13	23	33	287
T_3	29	5	15	16	16	10	10	20	23	6	13	23	16	202

Columns: L_1 = Robot, L_2 = RobotTime, L_3 = RobotPositive, L_4 = ExprAdd, L_5 = RobotUniExprAdd, L_6 = Dec, L_7 = RobotUniExprAddUniDec, L_8 = RobotTimeSpeed, L_9 = ExprAddUniDec, L_{10} = RobotExtExprAddUniDec, L_{11} = RobotUniExprAddExtRobotTime, L_{12} = ExprMul, L_{13} = RobotUniExprMul Rows: LISA = Mernik's Implementation, T_i = Technique i, for $i \in \{1, 2, 3\}$

Fig. 2: Lines-of-Code Comparison between Mernik's LISA and Our Three Techniques

embedded language development but are heavyweight in both term creation [3] and algebra composition. It is easy to turn `na_syntax` and the like into Object Algebra Interfaces to lower those two weights. Finally, it is important to also produce catalogues like this paper in other host languages than Scala. We are currently working on that.

REFERENCES

[1] P. Bahr and T. Hvitved. Parametric Compositional Data Types. In J. Chapman and P. B. Levy, editors, 4^{th} *MSFP*, volume 76 of *ENTCS*, pages 3–24, February 2012.

[2] E. Barrett, C. F. Bolz, and L. Tratt. Approaches to Interpreter Composition. *Comp. Lang., Sys. & Struct.*, 44:199–217, 2015.

[3] A. P. Black. The Expression Problem, Gracefully. In M. Sakkinen, editor, *MASPEGHI@ECOOP 2015*, pages 1–7. ACM, July 2015.

[4] W. Cazzola and E. Vacchi. Language Components for Modular DSLs using Traits. *ComLan*, 45:16 – 34, 2016.

[5] W. R. Cook. Object-Oriented Programming Versus Abstract Data Types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *FOOL*, volume 489 of *LNCS*, pages 151–178, Holland, June 1990.

[6] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel. Melange: A Meta-Language for Modular and Reusable Development of DSLs. In R. F. Paige, D. Di Ruscio, and M. Völter, editors, 8^{th} *SLE*, pages 25–36, October 2015.

[7] A. Dijkstra, J. Fokker, and S. D. Swierstra. The Architecture of the Utrecht HASKELL Compiler. In S. Weirich, editor, 2^{nd} HASKELL, pages 93–104, Edinburgh, Scotland, 2009. ACM.

[8] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language Composition Untangled. In A. Sloane and S. Andova, editors, 12^{th} *LDTA*, page 7. ACM, March 2012.

[9] E. Ernst. Family Polymorphism. In J. Lindskov Knudsen, editor, 15^{th} *ECOOP*, volume 2072 of *LNCS*, pages 303–326. Springer, June 2001.

[10] Guttag, J. V. and Horning, J. J. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10:27–52, 1978.

[11] S. H. Haeri. *Component-Based Mechanisation of Programming Languages in Embedded Settings*. PhD thesis, STS, TUHH, Germany, December 2014.

[12] S. H. Haeri and S. Schupp. Reusable Components for Lightweight Mechanisation of Programming Languages. In W. Binder, E. Bodden, and W. Löwe, editors, 12^{th} *SC*, volume 8088 of *LNCS*, pages 1–16. Springer, June 2013.

[13] S. H. Haeri and S. Schupp. Expression Compatibility Problem. In J. H. Davenport and F. Ghourabi, editors, 7^{th} *SCSS*, volume 39 of *EPiC Comp.*, pages 55–67. EasyChair, March 2016.

[14] S. H. Haeri and S. Schupp. Integration of a Decentralised Pattern Matching: Venue for a New Paradigm Intermarriage. In M. Mosbah and M. Rusinowitch, editors, 8^{th} *SCSS*, volume 45 of *EPiC Comp.*, pages 16–28. EasyChair, April 2017.

[15] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic Embedding of DSLs. In Y. Smaragdakis and J. G. Siek, editors, 7^{th} *GPCE*, pages 137–148, Nashville, TN, USA, October 2008. ACM.

[16] J. Jeuring, S. Leather, J. P. Magalhães, and A. R. Yakushev. Libraries for Generic Programming in HASKELL. In P. W. M. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *Adv. Func. Prog.*, 6^{th} *Int. School, AFP*, volume 5832 of *LNCS*, pages 165–229. Springer, May 2008.

[17] D. E. Knuth. Semantics of Context-Free Languages. *Math. Sys. Theo.*, 2(2):127–145, 1968.

[18] R. C. Martin. Design Principles and Design Patterns, 2000. online article available from the ObjectMentor website.

[19] M. Mernik. An Object-Oriented Approach to Language Compositions for Software Language Engineering. *J. Sys. & Soft.*, 86(9):2451–2464, 2013.

[20] P. D. Mosses. Modular Structural Operational Semantics. *JLAP*, 60–61:195–228, 2004.

[21] P. D. Mosses. Component-Based Description of Programming Languages. In E. Gelenbe, S. Abramsky, and V. Sassone, editors, *BCS Int. Acad. Conf.*, pages 275–286. Brit. Comp. Soc., 2008.

[22] M. Odersky. Pimp my Library. *Artima Developer Blog*, 9, October 2006.

[23] M. Odersky and M. Zenger. Independently Extensible Solutions to the Expression Problem. In *FOOL*, January 2005.

[24] B. C. d. S. Oliveira. Modular Visitor Components. In 23^{rd} *ECOOP*, volume 5653 of *LNCS*, pages 269–293. Springer, 2009.

[25] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the Masses – Practical Extensibility with Object Algebras. In 26^{th} *ECOOP*, volume 7313 of *LNCS*, pages 2–27. Springer, 2012.

[26] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type Classes as Objects and Implicits. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, 25^{th} *OOPSLA*, pages 341–360. ACM, October 2010.

[27] B. C. d. S. Oliveira, S.-C. Mu, and S.-H. You. Modular Reifiable Matching: A List-of-Functors Approach to Two-Level Types. In B. Lippmeier, editor, 8^{th} HASKELL, pages 82–93. ACM, September 2015.

[28] J. Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Comp. Surv.*, 27(2):196–255, 1995.

[29] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 7^{th} edition, 2009.

[30] J. C. Reynolds. User-Defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction. In S. A. Schuman, editor, *New Direc. Algo. Lang.*, pages 157–168. INRIA, 1975.

[31] T. Rompf. Reflections on LMS: Exploring Front-End Alternatives. In A. Biboudis, M. Jonnalagedda, S. Stucki, and V. Ureche, editors, 7^{th} *SIGPLAN Symp. Scala*, pages 41–50. ACM, November 2016.

[32] T. Rompf and M. Odersky. Lightweight Modular Staging: a Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In 9^{th} *GPCE*, pages 127–136, Eindhoven, Holland, 2010. ACM.

[33] C. Saito, A. Igarashi, and M. Viroli. Lightweight Family Polymorphism. *J. Func. Prog.*, 18(3):285–331, 2008.

[34] T. Sheard and E. Pasalic. Two-Level Types and Parameterized Modules. *JFP*, 14(5):547–587, 2004.

[35] I. Sommerville. *Software Engineering*. Addison-Wesley, 9^{th} edition, 2011.

[36] M. Torgersen. The Expression Problem Revisited. In M. Odersky, editor, 18^{th} *ECOOP*, volume 3086 of *LNCS*, pages 123–143, Oslo (Norway), June 2004.

[37] P.W. Trinder, K. Hammond, H-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *JFP*, 8(1):23–60, January 1998.

[38] M. Völter. Language and IDE Modularization and Composition with MPS. *GTTSE*, 7680:383–430, 2011.

[39] P. Wadler. The Expression Problem. Java Genericity Mailing List, November 1998.

[40] P. Wadler and S. Blott. How to Make ad-hoc Polymorphism Less ad-hoc. In 16^{th} *POPL*, pages 60–76. ACM Press, January 1989.

[41] Y. Wang and B. C. d. S. Oliveira. The Expression Problem, Trivially! In 15^{th} *Modularity*, pages 37–41, New York, NY, USA, 2016. ACM.

[42] H. Zhang, Z. Chu, B. C. d. S. Oliveira, and T. van der Storm. Scrap Your Boilerplate with Object Algebras. In J. Aldrich and P. Eugster, editors, 29^{th} *OOPSLA*, pages 127–146, October 2015.