Experiences with Domain-Specific Language Embedding in Scala

Anthony M. Sloane

Department of Computing, Macquarie University, Australia
Software Engineering Research Group, Delft University of Technology, The Netherlands
Anthony.Sloane@mq.edu.au

Abstract

Embedding domain-specific languages (DSLs) in general-purpose programming languages offers a simpler path to implementation than developing standalone DSL processors. However, sacrifices must be made, particularly in formal analysis of DSL programs. This paper explores these trade-offs in the context of the Kiama project that is investigating embedding of language processing DSLs in the Scala language. Examples are presented from preliminary experiments with embedding packrat parsing and strategy-based term rewriting. Particular attention is paid to the novel features of Scala that assist with this effort, including case classes, implicit conversions, and advanced support for pattern matching.

1. Introduction

Developers of language processing systems such as compilers, static analysers and software transformation systems are faced with two main implementation methods. On the one hand, they can employ a generative approach and compile specifications of the processing into code. On the other hand, they can embed their specifications in an existing host programming language and use the facilities of that language to obtain an implementation. A hybrid approach is also possible, where generators are used to extend an existing language, resulting in a flexible form of embedding or a staged implementation.

When these approaches are compared, there is no clear winner. A generative approach is the most flexible since every detail can be controlled, whereas an embedding approach must live with the syntax and semantics of the host language. However, building a generator is more work than reusing an existing language implementation.

A traditional argument for generators is that the user does not have to learn a programming language, making the approach suitable for non-programmers who are familiar with the problem domain. However, for language processing systems the target audience is most likely software developers and tool builders. For these users, learning a generator and its input notation is often a disincentive. Thus, embedding is a more attractive approach in this domain than in domains where non-programmers are the end users.

The landscape is changing continuously on both sides of this equation. New meta-tools are being developed to make it easier to implement language processing tools. The capabilities of potential host languages are always being extended and new languages are being developed, many with features designed to support embedding.

Kiama

Kiama is a new project that is investigating the embedding side of this debate. Motivation comes from experience building generators for language specifications in the Eli system [11]. The resulting generators satisfied their goals, but in many cases the specification languages were ad hoc and lacked features that are commonplace in general purpose languages such as support for proper parameterisation, modularisation and extensibility. As language processing systems are used to tackle larger problems and different techniques are combined, support for these kinds of structuring mechanisms becomes increasingly important.

The Kiama thesis is that in the language processing domain it is better to start with a modern general purpose language that embodies prevailing wisdom about how to structure, scale and extend applications, than to expect every generator builder to incorporate this wisdom into their own tools. The initial aim of Kiama is to combine many proven language processing methods into a coherent whole, supported by general facilities from a host language.

Kiama is hosted by the Scala programming language that provides both object-oriented and functional features in a statically-typed combination running on the Java Virtual Machine [21, 22]. Thus, Scala constitutes a powerful base on which to experiment with embedding. Most previous work on embedding languages has been conducted in functional languages such as Haskell or in dynamic languages such as Ruby or Smalltalk. Operating within the constraints of a hybrid language such as Scala offers some advantages but also presents challenges. Exploring these boundaries is an important component of the Kiama project.

The rest of this paper presents a tour through the major features of Kiama as it currently stands, beginning with pro-

gram representations in abstract syntax and pattern matching. Packrat parser combinators are then described, followed by processing of programs using strategy-based term rewriting, and a pointer to current work on attribute grammar-like facilities. Finally, infrastructure obtained from the host language is discussed, particularly for testing. Throughout, the noteworthy features of Scala are outlined and brief comparisons with Haskell are provided, since it is the most widely used host language for embedding.

2. Abstract syntax and pattern matching

Scala provides *case classes* to fill the role of algebraic data types in languages like Haskell and ML. The main differences between a case class and an ordinary class are a simpler constructor invocation and support for pattern matching. For example, Figure 1 shows the abstract syntax of a simple imperative language, encoded in a standard object-oriented way using case classes.

Trees conforming to this abstract syntax can be created easily. For example, the while program

```
{ i = 10; while (i) i = i - 1; }
```

can be written as

```
Seqn (List (
  Asgn ("i", Num (10)),
  While (Var ("i"),
    Asgn ("i", Sub (Var ("i"), Num (1))))))
```

Apart from simplified construction, case classes can also be used in pattern matching. For example, the following code fragment matches against an expression `e` to compute its value, if possible.

```
e match {
    case Num (i)                 => i
    case Add (Num (i), Num (j)) => i + j
    case Sub (Num (i), Num (j)) => ...
}
```

The case class style of Scala is more verbose than the Haskell equivalent, but integrates nicely with the rest of the language. Generalised algebraic data types can be achieved using parameterised types. Case classes can have additional fields and members so their basic functionality can be extended in ways that Haskell data types cannot. Value construction and pattern matching are very similar to Haskell. Case classes automatically get default support for pretty-printing as data types do in Haskell but case classes also get binary serialisation.

3. Parser combinators

Parser combinators have a long history, particularly in functional programming languages (see, for example, [8, 9, 16, 24]). Kiama has a parsing library based around *parsing expression grammars (PEGs)* and the associated *packrat parsing* method [4, 5, 6]. PEGs are similar to context-free grammars but have an ordered choice operator and semantic predicates. Nevertheless, they remain closed under composition which is useful for language extension and combination tasks. Packrat parsing is a linear time recognition method, albeit with non-trivial space usage due to memoisation. Kiama supports arbitrary left recursion using the algorithms used to implement the OMeta language [27, 28].

The Kiama parser combinator notation is based on PEGs but, for compatibility, also on the standard Scala parser combinator library, that follows a more traditional backtracking combinator parsing approach [18]. For example, Figure 2 shows a Kiama parser for the simple imperative language and summarises the combinators used. Some of them are straight-forward functions that combine parsers. Others, such as the Kleene star and plus are methods on Parser class. Scala helps here by permitting operator characters to be used as method names and by allowing the "." for method invocation to be omitted, so that `stmt*` stands for `stmt.*`.

Lazy values are used so that the order of definition does not matter. Types are inferred except for recursive productions and are provided by the user when memoisation is required (for example, for `exp`, `term` and `factor`). Strings are *implicitly converted* to parsers by the library function `literal` a call to which is inserted by the compiler where necessary.

```
implicit def literal
    (s : String) : Parser[String] = ...
```

Semantic actions are incorporated using the `^^` combinator. For example, the following production builds a tree fragment for a while statement.

```
lazy val whileStmt =
    ("while" ~> "(" ~> exp <~ ")") ~ stmt ^^
    { case e ~ b => While (e, b) }
```

The semantic action is written as an *anonymous pattern matching function* that deconstructs the pair returned by the parser and constructs the abstract syntax tree node.

From the user's perspective, Kiama's parsing library is very similar to those in other languages and to Scala's standard library. The main difference is the ability to use left recursion, leading to more natural grammars. Implicit conversions to parsers play a similar role to the type classes in a Haskell parser library, but in a more disciplined way since they are properly scoped. From an implementor's perspective, the library was straight-forward to get running, because the published imperative packrat algorithms can be coded almost directly in Scala. The syntactic rules for method calls mean that an object-oriented design can be maintained but a simpler syntax used in calls, approximating the clean syntax of Haskell. In fact, even prefix methods can be defined, allowing `!x` to be used instead of `not(x)`, for example.

Combining parsers with Scala's implicit conversions provide a dynamic version of concrete syntax support. For ex-

```
abstract class Stmt
case class Null extends Stmt
case class Seqn (ss : Seq[Stmt]) extends Stmt
case class Asgn (s : String, e : Exp) extends Stmt
case class While (e : Exp, b : Stmt) extends Stmt

abstract class Exp
case class Var (s : String) extends Exp
case class Neg (e : Exp) extends Exp
case class Add (l : Exp, r : Exp) extends Exp
case class Sub (l : Exp, r : Exp) extends Exp
case class Mul (l : Exp, r : Exp) extends Exp
case class Div (l : Exp, r : Exp) extends Exp
```

Figure 1. Abstract syntax for a simple imperative language.

```
lazy val stmt : Parser[Stmt]      = ";" | sequence | asgnStmt | whileStmt
lazy val sequence                 = "{" ~> (stmt*) <~ "}"
lazy val asgnStmt                 = idn ~ ("=" ~> exp) <~ ";"
lazy val whileStmt                = ("while" ~> "(" ~> exp <~ ")") ~ stmt
lazy val keyword : Parser[String] = "while"

lazy val exp : MemoParser[Exp]    = exp ~ ("+" ~> term) | exp ~ ("-" ~> term) | term
lazy val term : MemoParser[Exp]   = term ~ ("*" ~> factor) | term ~ ("/" ~> factor) | factor
lazy val factor : MemoParser[Exp] = double | integer | variable | "-" ~> exp | "(" ~> exp <~ ")"

lazy val integer                  = token (digit+)
lazy val double                   = token ((digit+) ~ ("." ~> (digit+)))
lazy val variable                 = idn
lazy val idn                      = not (keyword) ~> token (letter ~ (letterOrDigit*))
```

Combinator	*Meaning*
`x | y`	Try to recognise an `x` and if that fails, try a `y`.
`x ~ y`	Parse an `x` then a `y`, returning the paired result of both parses, if both succeed.
`x ~> y`	Parse an `x` then a `y`, returning the result of the `y` parse, if both succeed.
`x <~ y`	Parse an `x` then a `y`, returning the result of the `x` parse, if both succeed.
`x*`	Parse zero or more copies of whatever `x` parses.
`x+`	Parse one or more copies of whatever `x` parses.
`token (x)`	Recognise what `x` does, preceded by optional layout (e.g., whitespace).
`not (x)`	Succeed if `x` fails at the current position, otherwise fail. Has no effect on the input.

Figure 2. A Kiama parser for the simple imperative language (semantic actions omitted) and the relevant parsing combinators.

ample, an implicit function can be used to automatically parse a literal string when a syntax tree is required.

```
implicit def strToExp (s : String) : Exp =
    ... run exp parser on s ...

val e : Exp = "1 + 3 * 4"
```

Parsers can also be incorporated into pattern matching via Scala's *extractor patterns*. An object can be used in a pattern match and its `unapply` method is invoked to perform the match and bind variables. In this example, the `Assign` object provides transparent access to the assignment statement parser in a pattern matching expression[1].

```
object Assign {
    def unapply (s : String) :
                Option[(String,Exp)] =
        ... use asgnStmt parser on s
        return Some (i,e) on a successful parse
        None on failure ...
}
```

[1] Scala's `Option` type is similar to Haskell's `Maybe`.

The extractor is used as follows:

```
str match {
    case Assign (id, exp) => ...
    ...
}
```

4. Strategy-based term rewriting

Stratego is a successful term rewriting language based around the concept of generic tree traversals [26]. Its semantics [2] is naturally suited to a combinator-style implementation where a strategy is implemented as a function that takes a subject term as argument and returns either a transformed term or a failure indication. For example, the following Kiama function defines a simple transformation of arithmetic expressions.

```
def simplify : Exp => Exp =
    rewrite (everywheretd (rule {
        case Sub (x, y) =>
            simplify (Add (x, Neg (y)))
        case Add (x, y) if x == y =>
            Mul (Num (2), x)
    }))
```

The code delimited by the braces denotes a partial function that performs the transformation on a single expression. `rule` transforms this function into a Stratego-style strategy. `everywheretd` is a combinator that applies its argument recursively to every sub-term of the subject term. Finally, `rewrite` applies its argument strategy to an expression and, if it succeeds, returns the resulting expression, otherwise it returns the original expression.

The Kiama implementation provides a natural definition of the generic strategy combinators that is very similar to Stratego's library definitions. For example, `everywheretd` is defined as follows.

```
def everywheretd (s : => Strategy) : Strategy =
    topdown (attempt (s))
def topdown (s : => Strategy) : Strategy =
    s <* all (topdown (s))
def attempt (s : => Strategy) : Strategy =
    s <+ id
```

The definition of `everywheretd` uses a generic `topdown` combinator and a combinator `attempt` that applies its argument strategy and returns an unchanged term if the argument fails. `<*` is the sequence combinator that executes its right argument only if its left argument succeeds, and `<+` is guarded choice that executes its right argument only if its left argument fails. (`id` is the identity strategy.) As for the parsing library, this rewriting library also benefits from Scala's syntactic rules since something like `s <+ id` is actually `s.<+(id)`.

The Kiama implementation varies from Stratego in a number of ways. For example, the Stratego semantics explicitly manipulates an environment for bound variables in patterns, which Kiama achieves using normal Scala bindings. This design changes the semantics somewhat since in Stratego it is possible to pass bindings from one strategy to another in a sequence. In Kiama this is only possible within a single rule, but this limitation has not been encountered in testing so far. Also, Stratego's strategies are not typed, whereas Kiama can use Scala's type system to provide local checking for rules. Since Kiama strategies can access any Scala code, computations that don't fit naturally into the term rewriting model are easier than in Stratego. For example, arithmetic in Stratego uses special primitive strategies and context-dependent computations such as scoping use dynamic rules, both of which can be directly coded in Kiama rules using standard Scala.

Most of the Stratego syntax carries over to the Scala setting, with sequential combination the most notable exception since Stratego's semicolon operator is already used by Scala's syntax. Kiama uses call-by-name parameters (indicated by `=>` before the parameter type) to avoid infinite expansion, which would be trivially dealt with by lazy evaluation in a Haskell implementation. Scala's anonymous pattern matching functions make some aspects easier to state than in a Haskell implementation that would require separate function definitions.

Kiama's term rewriting library is clearly similar in many ways to numerous Haskell rewriting and generic traversal libraries, including Scrap Your Boilerplate [14, 15] and Uniplate [17]. Kiama follows Stratego more closely than these libraries and currently doesn't attempt to use the type system as fully. TOM extends Java and other languages with transformation features based on pattern matching and strategies [1, 19]. When used with Java, TOM has a similar context to Kiama but is a generator not an embedding, so the emphasis is different.

5. Attribution

Current work on Kiama is investigating analysis of syntax trees using techniques inspired by attribute grammars. The approach being used is a dynamically scheduled evaluator in the style of Jourdan [10] and similar to that used by the JastAdd generator [7]. Details are currently in flux, preventing a proper explanation here, but more information should be available by the workshop.

In recent related work at Delft, Lennart Kats has developed a method for expressing attribute grammar shorthand notations using strategic programming [12]. In this approach, standard attribute grammar equations provide the local computations. Non-local accesses and higher-order attribute patterns are achieved using *decorators* that specify how to obtain attribute values using a generic traversal of the tree. Using decorators, mechanisms such as reaching up the tree or evaluating an attribute until a fixed point is reached can be expressed in a generic way, rather than being embed-

ded in the attribute grammar system implementation. A trial of decorators in Kiama is also underway.

6. Infrastructure

As is the case for all embedded languages, Kiama inherits tool support from its host language. For example, Kiama programs can be debugged using standard JVM-based debuggers such as that provided by Eclipse. For example, it is straight-forward to place breakpoints inside rewriting strategies and to examine the terms to which they are being applied. Of course, this method of debugging exposes library details rather than preserving the DSL interface as DSL-level debugging support could do.[2]

More interestingly, testing frameworks for Scala are very useful in the Kiama setting. For example, the contributed scalacheck library [20], that provides functionality similar to that provided by Quickcheck for Haskell, can be used to write sophisticated tests. For example, the following code provides round-trip testing of parsing and pretty-printing.

```
trait PrettyPrintable {
    def pretty (o : StringBuilder)
}

def roundtrip[T <: PrettyPrintable]
        (parser : Parser[T])
        (implicit arbT : Arbitrary[T]) {
    check ((t : T) => {
        val buffer = new StringBuilder
        t.pretty (buffer)
        expectBool (parser, buffer.toString, t)
    })
}

// Sample invocation
roundtrip (exp)
```

`roundtrip` is generic in the type `T` of the values that are returned by the parser to be tested. The upper bound on `T` ensures that values of that type can be pretty-printed via the `PrettyPrintable` trait. The body of `roundtrip` uses `check` from the scalacheck library to test a predicate on `T` values. `check` requires a way to create arbitrary values of type `T`. This capability is provided by the parameter `arbT`, which if not explicitly given, will be provided implicitly from the current scope. In our example, the tests import `Arbitrary[T]` values for all types in the imperative language abstract syntax. Provided with a test value, the actual test pretty-prints the value into a buffer, parses it back and expects that the parsed value is the same as the test value.[3] The helper function `expectBool` hides the details of running the parser, extracting the value from a successful parser result and so on.

[2] Furthermore, in the current versions of the Eclipse debugging support, no attempt is made to hide the Scala-to-JVM translation, so some manual "resugaring" is necessary at that level as well.

[3] Of course, this test assumes that pretty-printing is working correctly.

7. Conclusion and Open Questions

Kiama is a young library and has not been tried on very large problems. Yet, it is clear that the embedding approach is allowing functionality approaching that of generators to be obtained with a relatively small amount of work: around 3000 lines of code, including tests and documentation comments.

Scala has proven to be a convenient host language since it offers many of the relevant features of languages previously associated with this kind of library, such as high-level pattern matching in Haskell or ML, but also has an imperative base, access to lazy evaluation, a more predictable execution model and access to Java infrastructure. Scala's scalable component features [23] are expected to be of increasing importance as larger problems are attempted.

As in any embedding experiment, some syntactic concessions have been made, but so far at least, these have not been onerous. Potentially more problematic are correctness concerns. A true embedding approach doesn't support analysis of DSL programs beyond that provided by the host language. Embedding therefore encourages a design that uses the host language for static checking of syntax, names and types but does not require more extensive correctness analysis of DSL programs. For example, using a technique for packrat parsing that supports left recursion means that Kiama does not need to check for it. Similarly, Stratego's semantics means that once a strategy has been created, it can be run without requiring any further static checking. It is an open question whether DSLs of this kind exist for all of the language processing tasks that one would wish to do.

A bigger question relates to the guarantees that can be made about the execution of a DSL program. Since DSLs such as the Stratego embedding in Kiama give full access to Scala, there is significant potential for incorrect execution. For example, a rule could cause a non-local control flow via an exception thereby breaking a rewrite traversal that is in progress. Current work is investigating ways to capture the constraints and requirements of DSLs and the host language so that some checking can be performed.

Beyond the general issues of DSL combination, particular combinations are also of interest. In particular, Kiama will provide a base on which to experiment with interactions between rewriting and attribution. JastAdd provides one form of combination where rewrites can be triggered by attribute accesses [3], but there are other possibilities.

Some aspects of Scala deserve to be more fully investigated. For example, encapsulation of parsing, rewriting or analysis within extractor patterns can provide significant abstraction, but more use cases need to be considered to see if this is really useful. Scala's type system needs to be explored more fully to give types to strategies, perhaps along the line of Lämmel's work [13]. As shown earlier, a limited

form of dynamic concrete syntax for languages can be supported by invoking parsers from implicit string conversion, but this does not currently carry across to pattern matching in the sense of systems like Stratego and ASF+SDF[25].

Acknowledgments

The author thanks the Software Engineering Research Group at the Technical University of Delft for hosting the study leave during which this work is being performed. This visit is supported by the Dutch NWO grant 040.11.001, *Combining Attribute Grammars and Term Rewriting for Programming Abstractions*. Eelco Visser, Lennart Kats, Charles Consel, Mark van den Brand, members of IFIP WG 2.11 and the anonymous reviewers have provided useful feedback on this work.

References

[1] BALLAND, E., BRAUNER, P., KOPETZ, R., MOREAU, P., AND REILLES, A. Tom: Piggybacking Rewriting on Java. *Lecture Notes in Computer Science 4533* (2007), 36.

[2] BRAVENBOER, M., VAN DAM, A., OLMOS, K., AND VISSER, E. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae 69* (2005), 1–56.

[3] EKMAN, T., AND HEDIN, G. Rewritable reference attributed grammars. In *Proceedings of the 18th European Conference on Object-Oriented Programming* (2004), vol. 3086, pp. 147–171.

[4] FORD, B. Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming* (2002), 36–47.

[5] FORD, B. Parsing expression grammars: a recognition-based syntactic foundation. *ACM SIGPLAN Notices 39*, 1 (2004), 111–122.

[6] GRIMM, R. Practical packrat parsing. *New York University Technical Report, Dept. of Computer Science, TR2004-854* (2004).

[7] HEDIN, G., AND MAGNUSSON, E. Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program. 47*, 1 (2003), 37–58.

[8] HUTTON, G. Higher-Order Functions for Parsing. *Journal of Functional Programming 2*, 3 (1992), 323–343.

[9] HUTTON, G., AND MEIJER, E. Monadic parser combinators. *Journal of Functional Programming 8*, 4 (1996), 437–444.

[10] JOURDAN, M. An optimal-time recursive evaluator for attribute grammars. In *International Symposium on Programming* (1984), Springer, pp. 167–178.

[11] KASTENS, U., SLOANE, A. M., AND WAITE, W. M. *Generating Software from Specifications*. Jones and Bartlett, Sudbury, MA, 2007.

[12] KATS, L., SLOANE, A., AND VISSER, E. Decorated attribute grammars: Attribute evaluation meets strategic programming. Submitted for publication., October 2008.

[13] LÄMMEL, R. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming 54*, 1 (2003), 1–64.

[14] LÄMMEL, R., AND PEYTON JONES, S. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices 38*, 3 (Mar. 2003), 26–37. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[15] LÄMMEL, R., AND PEYTON JONES, S. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)* (Sept. 2005), ACM Press, pp. 204–215.

[16] LEIJEN, D. Parsec, a fast combinator parser. *URL: http://www. cs. uu. nl/daan/parsec. html* (2000).

[17] MITCHELL, N., AND RUNCIMAN, C. Uniform boilerplate and list processing. *Proceedings of the ACM SIGPLAN workshop on Haskell workshop* (2007), 49–60.

[18] MOORS, A., PIESSENS, F., AND ODERSKY, M. Parser combinators in scala. Tech. Rep. Report CW 491, Katholieke Universiteit Leuven, February 2008.

[19] MOREAU, P., RINGEISSEN, C., AND VITTEK, M. A Pattern Matching Compiler for Multiple Target Languages. *Lecture Notes in Computer Science* (2003), 61–76.

[20] NILSSON, R. scalacheck: a powerful tool for automatic unit testing. http://code.google.com/p/scalacheck/, October 2008.

[21] ODERSKY, M. *Scala language specification, Version 2.7*. Programming Methods Laboratory, EPFL, Switzerland, August 2008.

[22] ODERSKY, M., SPOON, L., AND VENNERS, B. *Programming in Scala*. artima, 2008.

[23] ODERSKY, M., AND ZENGER, M. Scalable component abstractions. In *Proceedings of OOPSLA 2005* (2005).

[24] SWIERSTRA, S., AND DUPONCHEEL, L. Deterministic, error-correcting combinator parsers. *Advanced Functional Programming* (1996), 184–207.

[25] VAN DEN BRAND, M., SCHEERDER, J., VINJU, J., AND VISSER, E. Disambiguation filters for scannerless generalized lr parsers. *Compiler Construction* (2002), 21–44.

[26] VISSER, E. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, C. Lengauer et al., Eds., vol. 3016 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2004, pp. 216–238.

[27] WARTH, A., DOUGLASS, J., AND MILLSTEIN, T. Packrat parsers can support left recursion. *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (2008), 103–110.

[28] WARTH, A., AND PIUMARTA, I. OMeta: an object-oriented language for pattern matching. *Proceedings of the 2007 Symposium on Dynamic languages* (2007), 11–19.