Language Components for Modular DSLs using Traits

Walter Cazzola[a,*], Edoardo Vacchi[a]

[a] *Department of Computer Science, Università degli Studi di Milano, Italy.*

Abstract

Recent advances in tooling and modern programming languages have progressively brought back the practice of developing domain-specific languages as a means to improve software development. Consequently, the problem of making composition between languages easier by emphasizing code reuse and componentized programming is a topic of increasing interest in research. In fact, it is not uncommon for different languages to share common features, and, because in the same project different DSLs may coexist to model concepts from different problem areas, it is interesting to study ways to develop modular, extensible languages. Earlier work has shown that traits can be used to modularize the semantics of a language implementation; a lot of attention is often spent on embedded DSLs; even when external DSLs are discussed, the main focus is on modularizing the semantics. In this paper we will show a complete trait-based approach to modularize not only the semantics but also the syntax of external DSLs, thereby simplifying extension and therefore evolution of a language implementation. We show the benefits of implementing these techniques using the Scala programming language.

1. Introduction

In recent years, the practice of developing domain-specific languages (DSL) to deal with domain-specific problems has started to regain interest among researchers and practitioners as demonstrated by surveys and books [1, 2] and a more recent study about research trends and applications of DSLs [3]. Support tooling is becoming more and more powerful, flexible and convenient, with the introduction of new frameworks and platforms; modern API design is progressively converging to a style that resembles an *embedded language*, to the point where the very distinction between a DSL and a general purpose language (GPL) is becoming thinner; a style that Martin Fowler and Eric Evans dubbed *fluent interface* [4], and that languages such as Scala [5], Smalltalk, Ruby and Groovy actively promote: the flexible parser and syntax of such languages allow users to even omit some punctuation, making it simple to simulate the embedding of a foreign language. For instance, Listing 1, shows a Scala internal DSL (punctuation that can be omitted has been dimmed).

However, language embedding as a fluent interface has still to obey to the host language limitations; inevitably, *external* DSLs, provide an unrivaled level of syntactic flexibility

*Corresponding author, E-mail: cazzola@di.unimi.it

```
// multiple init
val opened, closed = new State;
// builder pattern
val open  = Transition.from(opened).to.(closed);
val close = Transition.from(closed).to.(opened);
// abstract factory
val door = StateMachine (
  // named parameters
  states      = List(opened, closed),
  transitions = List(open, close)
);
```

Listing 1: A State Machine language as a Scala embedded DSL

(for instance, compare the state machine in Listing 1 with the equivalent written in an external DSL and reported in Listing 3), at the cost of requiring developers to write their own parsing routines, and to implement the semantics of each single construct. Nevertheless, modern libraries and languages today provide programmers with tools that make developing their own external DSL within their reach. For instance, *parser combinator* libraries [6–8] make it possible to define an executable parser in such a way that the code that describes it closely resemble the structure and looks of its formal grammar. We are getting closer and closer to a full componentization of language implementations, where domain specific languages could be implemented as the combination of features concretized as reusable *assets*.

Modular language development is a research branch that investigates tools and techniques to componentize the design and implementation of languages, with particular attention to DSLs, where each feature may be easy to represent as a distinct code unit, making a language implementation very close to a combination of a selection of such components, realizing a form of *feature-oriented* language composition [9–12]. In general a *language implementation* can be described by i) a *parser* for a concrete syntax, yielding ii) an *abstract syntax tree* that puts in relation the concrete syntax of an input program with an *abstract syntax* representation, and iii) the semantics that can be associated with the nodes of the abstract syntax tree [13, 14]. These three parts of a language can be decomposed into a collection of *language components* [9, 15], that is, a sort of *bundle* that includes the *syntax*, and the *semantics* of that construct; the semantics may be represented as a *sequence* of *evaluation phases* (*e.g.,* type checking, evaluation, etc.) that pertain to that particular construct (Fig. 1). A language component can be shared and distributed as a whole across different language implementations, possibly as a binary, pre-compiled package; the final objective is to provide the features of the language as self-contained bundle of components that can be just combined together. The contribution of this work is to synthesize a collection of patterns and techniques that can be used to implement language components, using *traits*, lightweight entities of code reuse that are often contrasted to single and multiple inheritance [16, 17], and that have been already shown (*e.g.,* [12, 18, 19]) to be especially good to achieve language componentization. To this end, we will show how to

- separate the syntactic concern from the construction of the abstract representation of the language;
- separate the abstract representation of the language from the implementation of its semantics;

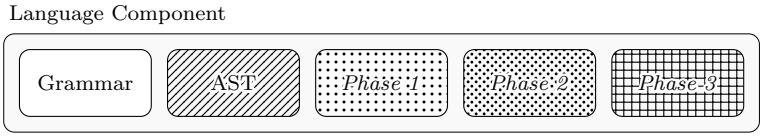

Figure 1: A Language Component.

- modularize the implementation of the semantics in distinct phases;
- decouple the abstract representation from the semantics of each phase, possibly expressing dependencies between phases.

We will use traits to *componentize the parser implementation* in such a way that the code resembles a grammar, thereby making it easier to understand and develop, and to *implement the interpreter pattern* separating different concerns of the semantics of the language constructs.

The benefit of representing language concepts through traits is an improved modularization, thereby simplifying code sharing across language implementations. Moreover, since traits in most languages can be written as separate code units, employing them in the modularization of a language make it possible to compile each language component *separately* and *independently* from the others, allowing them to be shared as binary assets, that, nevertheless, can still be combined together post-compilation.

The approach that we present has been influenced by many sources of inspiration: first of all, Scala's parser combinator library bundles *traits* with predefined combinators for commonly used literals and regex patterns, that users can mix-in to their classes; then, our experience with the implementation of the Neverlang framework [15, 20–22] for componentized language development, with which the trait-based model that we will present shares a few commonalities; finally, the previous work on modularizing the semantics of an interpreter (*e.g.,* [12, 18, 19]). The objective of this work is to present a complete solution, including syntax and semantic composition, to realize the implementation of *language components*. The final goal will be, in the future, to be able to implement languages in a feature-oriented way, possibly using feature modeling techniques to present the variability in a language family; such an experience has been already carried out —see [9, 23, 24]— using our own programming language framework, that provides first-class support for language components (known as *slices*); in this work we want to show that, although a dedicated tool simplifies a componentized model of language development, a similar degree of code reuse can be reached by employing constructs and features that are already available in many modern GPLs.

For this work we chose to use Scala's trait implementation, since it completes Schärli's original prototype [16] with the additional guarantees of correctness that a static type system provides. Nevertheless, the approach should be portable to any language that supports trait-like composition and a library for parser combinators, such as Smalltalk, Ruby, Groovy, etc.

A simple state machine DSL. As our running example we will use a simple State Machine language similar to the one from Tratt's paper [25] (see Listing 2). Similarly to Tratt, we will also show how to extend the basic state machine DSL with guards and action language; but in our case the extended DSL will be the result of composing together

```
// StateMachine
SMachine    → "state" "machine" Ident Initial "{" Body "}" ;
Initial     → "initial" "(" Ident ")"
// Transition types
Transition  → SimpleTransition ;
SimpleTransition → "transition" "from" Ident "to" Ident ":" Ident ;
// State types
State       → SimpleState ;
SimpleState → "state" Ident;
// Body Definition
Body        → State* Transition* ;
```

Listing 2: State Machine DSL Grammar.

```
state machine Door initial closed {
  state opened state closed
  transition from opened to closed: close
  transition from closed to opened: open
}
```

Listing 3: Door State Machine for the grammar in Listing 2.

traits from the basic state machine language and a separate *action language*.

The rest of this paper is structured as follows. Section 2 describes the background. Then the paper is divided into two parts: in Sect. 3 we will draw a parallel between grammars and traits and we will show that it is possible to modularize a parser implementation in the same way we will partition the set of rules of a formal grammar. In this section we will use Scala's traits and its parser combinator library. In Sect. 4 we will show how to implement the semantics of our DSLs using traits to decouple the semantic implementation of the interpreter from the abstract representation of language concepts. Section 5 expands the running example of state machines in a full case study, by extending the basic state machine language with support for an *action language* and *guard expressions*. Section 6 compares our solution to some related work, and in Sect. 7 we draw our conclusions.

2. Background

We will give a few details on the technical background that is required to understand the rest of this paper. We first briefly recap formal grammars, then we define the concept of trait as found in [16, 17], and finally we describe the peculiarities of Scala's trait implementation, with respect to the features we will use here.

2.1. Formal Grammars

In the following we will assume that the reader has some confidence with language theory, please refer to a book on the topic for general definitions (*e.g.,* [26]). In short, a formal grammar is a tuple $G = \langle \Sigma, N, P, S \rangle$, where Σ is an alphabet of *terminal symbols*, N is an alphabet of *nonterminal symbols*, P is a set of *production rules* and $S \in N$ is the *start symbol*. A production rule (or simply a *production*) is written as $A \to \omega$ where $A \in N$,

```
package sm.lang

trait StateMachine extends RegexParsers {
  // provided
  def statemachine = "state" ~> "machine" ~> ident ~ initial ~ ( "{" ~> body <~ "}" )
  def initial = "initial" ~> ("(" ~> ident <~ ")")
  // required
  type TBody
  def body : Parser[TBody]
  def ident: Parser[String]
}

trait SimpleTransition extends RegexParsers {
  def simpleTransition = ("transition" ~ "from") ~> ident ~ ("to" ~> ident) ~ (":" ~> ident)
  def ident: Parser[String]
}

trait SimpleState extends RegexParsers {
  def simpleState = "state" ~> ident
  def ident: Parser[String]
}

trait Transition extends RegexParsers {
  type TTransition
  def simpleTransition: Parser[TTransition]
  def transition = simpleTransition
}

trait State extends RegexParsers {
  type TState
  def simpleState: Parser[TState]
  def state = simpleState
}

trait Body extends RegexParsers {
  type TTransition; type TState
  def state: Parser[TState]
  def transition: Parser[TTransition]
  def body = state.* ~ transition.*
}
```

Listing 4: A trait-based parser for grammar in Listing 2.

and $\omega \in (\Sigma \cup N)^*$, with $(\Sigma \cup N)^*$ being the transitive closure of set $\Sigma \cup N$ with respect to symbol juxtaposition. The generated language $L(G)$ of a grammar is the set of all the words that can be *derived* from a starting nonterminal S for the grammar G. A language for a grammar G is said to be *empty* if $L(G) = \varnothing$ and, conversely, non-empty when it contains at least one sentence. In other words, there exists at least one *sentence*, (or *word*, or *program*) that can be expressed using the language represented by G. In the following, we will assume grammars that generate non-empty languages, and, for simplicity, we will make the assumption that our grammars do not contain the empty word ε[1].

[1] As well-known this is not a real limitation since any context-free grammar with ε can be transformed without losing any information in a grammar without any empty word. This only affects the grammar size.

2.2. Parser Combinators

A parser can be seen as a function that takes a stream of characters as input and produces a *parse tree*. Higher-order functions known as *combinators* can be composed together to construct grammar structures such as sequencing, repetitions, optionality and choice. If the host language supports infix operator notation, then a grammar rule written using parser combinators resembles an EBNF production. The biggest advantage of parsing with combinators is improved composability; larger parsers are generated by taking simple, primitive parser and *composing* them functionally. There are many object-oriented frameworks for parser combinators, for instance jParsec[2] and Scala parser combinators [7]. All implementations use the host language to build an object model of parsers.

Scala's parser combinator library provides a hierarchy of *traits*, the base trait `Parsers` provides the basic combinators, and its descendants provide utility combinators, like pre-defined tokenizers; *e.g.,* `JavaTokenParsers` defines combinator for Java-style identifiers, numbers, string literals, etc. Each combinator is represented as an instance of a `Parser[T]`, a function `Input` ⇒ `ParseResult[T]`. `Parser[T]` provides methods that combine the `Parser[T]` on which they are invoked with the combinator that they are given as an argument. For instance `p.~(q)` is the *sequence* combinator, `p.*` is the *repetition* combinator, etc. The `^^` attaches an action to the combinator it is invoked onto. Because of Scala's ability to emulate *infix operators* through methods, combined with implicit conversions (*e.g.,* a quoted identifier such as `"state"` is implicitly converted into the combinator `literal("state")`), Scala's parser combinators closely resemble the EBNF representation of the syntax of the language. Listing 4, that we will describe with more detail in Sect. 3, shows the parser combinators that implement the grammar in Listing 2.

2.3. Traits

Formally, a *trait* [16, 17] is a function $t : \mathcal{N} \to \mathcal{B}^*$, mapping the set $\mathcal{N}$ of method *names* into the set $\mathcal{B}^*$ of method bodies; the set $\mathcal{B}^*$ includes the *undefined* method ($\bot$) and the *overspecified* method ($\top$), that represent *required* methods and *conflicting* methods, respectively. A trait is *free of conflicts* if, for each method name $n \in \mathcal{N}$ it must always be the case that $t(n) \neq \top$. Traits are composed with the *sum* operation $+ : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$; given two traits $t_1, t_2 \in \mathcal{T}$ the sum trait $t = t_1 + t_2$ is the respective union of all the provided and required methods in t_1 and t_2. In case of *conflict*, it is possible to provide a *method dictionary* d with an alternative definition of the conflicting methods in t; this is defined as a function $d \triangleright t : \mathcal{N} \to \mathcal{B}^*$:

$$(d \triangleright t)(l) \triangleq \begin{cases} t(l) & d(l) = \bot \\ d(l) & \text{otherwise} \end{cases}$$

where a *method dictionary* is again, a mapping $d : \mathcal{N} \to \mathcal{B}^*$.

Traits in Scala. Although there is sometimes debate about whether Scala's traits do really follow closely Schärli's original formulation [16], but Schärli compared Scala's trait implementation with his work [27]; the main differences are that they are modeled as

[2] `jparsec.codehaus.org`

a "special form of an abstract class" [28] that does not encapsulate state; they cannot only be composed but can also be inherited; they support generics; they do not support aliasing and exclusion. Otherwise, Scala's trait implementation is pretty close to the original definition, while the main limitations can be ascribed to the fact that Scala is a *statically typed* language whereas Smalltalk (the original implementation target) is a dynamic language. For instance, when two traits provide different, alternative, *conflicting* implementations of the same member, Scala is able to resolve automatically the conflict, by considering the *order* of composition of the traits (this is one way to tackle the *diamond problem*) otherwise the conflict has to be resolved *manually*; for more details, see [28]. Being Scala's traits *statically typed*, mixing in a trait with another requires to pay extra care to the type signatures of the members; however, besides *methods* Scala's traits include *abstract type definition*s to *require* that a concrete implementation *provides* a valid definition of a type at the moment of instantiation. For instance, in Listing 4 each trait *provides* the definition of concrete members (*e.g.,* `statemachine`), it *requires* the definition of abstract members (*e.g.,* `body`) and it *declares* abstract return types for its abstract members (*e.g.,* `TBody`).

For a given trait t, in Scala it is not possible to override members of t with *any* method dictionary d. For instance, arbitrary redefinitions of a concrete member's return type are clearly forbidden: in particular, overriding methods may be *contravariant* in their arguments and *covariant* in their return types. For instance, let $t(n)$, with $n \in \mathcal{N}$ be a function $f : A \to B$ for two arbitrary sets A, B, and let d be a dictionary of overriding members for t: we can define $t' = t \triangleright d$ if and only if $d(n)$ is a function $g : A' \to B'$ and both $A' \supseteq A$ and $B' \subseteq B$.

3. Trait-Based Grammar Modularization

A parser is usually defined as a single, self-contained entity, but reasoning by analogy with language grammar, a parser may be easily componentized. A grammar can be partitioned into a collection of interdependent *sets of productions*. Using parser combinators and traits, we provide a construction method to represent such sets and their dependencies as pluggable, shareable and reusable components. The resulting traits implement parser *components* that can be easily combined together, unplugged for language restriction, and shared with other languages for extension (Sect. 5). Similar results could be achieved by using inheritance [29] but sacrificing some flexibility when the language does not support multiple inheritance.

For a given grammar $G = \langle \Sigma, N, P, S \rangle$, the set of its productions P can be thought of as a collection of disjoint sets of rules $P_0, P_1, \ldots, P_{n-1}$ such that $P \equiv P_0 \cup P_1 \cup \cdots \cup P_{n-1}$, each of which may represent a different syntactic *feature* of the language. In other words, for a set of productions P, we can define an n-size partition $\mathbf{P} = \{P_0, P_1, \ldots, P_{n-1}\}$, and each set $P_k \in \mathbf{P}$ would represent a different syntactic *feature* or *concern* of a language.

Example. Consider the state machine grammar in Listing 2. Productions have been grouped into logical sets of features: i) the *container* for the body of the state machine, specifying an *initial* state, ii) the *body* of the state machine, as a sequence of *states* plus a sequence of *transitions*, iii) the definition of a simple *state* as an identifier, and iv) the definition of a simple *transition* as pair of state identifiers, plus a name.

Now, let G be a grammar such that $L(G) \neq \varnothing$. Then there is at least some $w \in L(G)$ and consequently there must be some sequence of derivations

$$S \Rightarrow \gamma \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_m \Rightarrow w.$$

By the definition of *derivation*, we write $\gamma \Rightarrow \gamma'$ if and only if there are α, β, $\omega \in (\Sigma \cup N)^*$, $A \in N$ such that $(\gamma = \alpha A \beta) \wedge (A \to \omega \in P) \wedge (\gamma' = \alpha\omega\beta)$. Thus, for each derivation step there must be some rule $A \to \omega \in P$. In particular, if we consider a set $\mathbf{P}$ of $n > 0$ partitions of P, then, by definition of partition, there is exactly one $P_i \in \mathbf{P}$ such that $0 \leq i < n$ and $A \to \omega \in P_i$. Then, for each partition P_k of productions, we can define a set $r(P_k) \subseteq N$ of *required* nonterminals and a set $p(P_k) \subseteq N$ of *provided* nonterminals. Then, for $j = 0, 1, \ldots, m$, with $m = |P_k|$, each production is of the form:

$$X_{j0} \to X_{j1} X_{j2} \cdots X_{j\mathbf{n}_j}$$

where $\mathbf{n}_j$ is the number of right-hand symbols in the j-th production (thus $\mathbf{n}_j + 1$ is the number of symbols of that production).

- The *provide* set $p(P_k)$ is the set of all the *left-hand* nonterminals of all the productions in P_k:
$$p(P_k) = \{X_{j0} \mid j = 0, 1, \ldots, m\},$$
- The *require* set is the set of all the *right-hand* nonterminals of all the productions in P_k, that are *not* also in $p(P_k)$:
$$r(P_k) = \{X_{ji} \mid j = 0, 1, \ldots, m,\ 0 < i \leq n_{\mathbf{j}},\ X_{ji} \notin p(P_k)\}$$

For instance, the set $\{\texttt{Body} \to \texttt{State} * \texttt{Transition}*\}$ in Listing 2 *provides* `Body` and *requires* `State` and `Transition`.

Grammar Composition. Because of the definition of partition, it is clear that any grammar $G = \langle \Sigma, N, P, S \rangle$, can be seen as the union $P_0 \cup P_1 \cup \cdots \cup P_{n-1}$. But also, for each language with a grammar G, it is always possible to define a grammar $G' = \langle \Sigma, N, P', S' \rangle$ where $S' \in N$ and P' is

$$P' = \bigcup_{P_i \in \mathbf{P}' \subset \mathbf{P}} P_i.$$

In other words, every grammar G' can be seen as the union of a collection of sets of productions $\mathbf{P}'$, which is itself a subset of a *universe* of sets of productions $\mathbf{P}$, with some axiom $S' \in N$ (not necessarily such that $S = S'$), and it is easy to see that $L(G') \subseteq L(G)$ for all $\mathbf{P}' \subset \mathbf{P}$. In particular, in order to guarantee that $L(G') \neq \varnothing$, we might want to pose the following restrictions:

$$S' \in p(P_k), P_k \in \mathbf{P}' \tag{1}$$
$$X \in r(P_k), P_k \in \mathbf{P}' \implies \exists P_j \in \mathbf{P}' : X \in p(P_k) \tag{2}$$

That is, the axiom for G' is a *provided* nonterminal for some set $P_k \in \mathbf{P}'$. Moreover, for each *required* nonterminal X in a production set P_k there *must* exist a set of rules P_j such that it *provides* X. Please notice that (2) is a strong requirement: in fact, although

Traits	**Production Sets**
A trait t *provides* the set of methods $\{n \in \mathcal{N} \mid t(N) \neq \bot\}$;	P_k *provides* a set of rewrite rules for the nonterminals $X \in p(P_k)$
A trait t *requires* the set of methods $\{n \in \mathcal{N} \mid t(N) = \bot\}$;	For all $X \in r(P_k)$, it *requires* that there exist $X \to \omega \in P_j$, with $P_j \in \mathbf{P}$.
For all pairs of traits $t_i, t_j \in \mathcal{T}$ it is possible to define the *sum* $t_i + t_j$; in case of conflicts in $t = t_i + t_j$ a function $d \triangleright t : \mathcal{N} \to \mathcal{B}^*$ must be provided to resolve them;	For all pairs $P_k, P_j \in \mathbf{P}$ it is possible to define the union set $P_k \cup P_j$. In grammars there is no notion of *conflict*, but two productions with the same left-hand side represent an *alternate choice* in a derivation step. For instance, consider some derivation $\gamma \Rightarrow \gamma'$ with $\gamma = \alpha A \beta$, and suppose that $A \to \omega, A \to \omega' \in P$, then γ' might be either $\alpha\omega\beta$ or $\alpha\omega'\beta$. This follows from the definition of derivation step and does not require an explicit resolution.
Assuming that there are no conflicts, the composition of $t_i + t_j$ is equivalent to the *flattened* trait t that contains all of the methods defined either in t_i or t_j.	The union $P_j \cup P_k$ is the collection of the productions either in P_j or in P_k; in other words, the set $P_j \cup P_k \equiv \{p \in P_j \vee p \in P_k\}$.

Table 1: Analogies between traits and sets of productions.

for each set $\mathbf{P}'$ for which (1) and (2) hold, it will be $L(G) \neq \varnothing$, it is easy to see that there might be cases when $L(G) \neq \varnothing$ even when (2) is not satisfied.[3] For instance consider the sets:

$$P_0 = \{S \to x\}, \quad P_1 = \{A \to B\}$$

then grammar $G = \langle \Sigma, N, P_0 \cup P_1, S \rangle$ generates a non-empty language, although there is $B \in r(P_k)$ and $B \notin p(P_0), B \notin p(P_1)$.

In the following we propose a construction method to represent a production set P_k as a trait t_k and we show how the union operation $P_k \cup P_j$, for $P_k, P_j \in \mathbf{P}$ relates to the trait composition $t_k + t_j$. We will see that, with this method, such restrictions play an important role.

3.1. Trait Construction

If we consider a partition $\mathbf{P}$ of the set P of productions of a grammar G, each set of the partition shares many commonalities with a trait; for brevity we summarize them in Table 1. A *recursive descent parser* would implement each production of a grammar as a *function*, and one way to implement such functions is to employ *parser combinators* (Sect. 2.2). A *grouping* of these functions could represent a *set of productions* in a partition, with all of its dependencies (that is, its *require* set). *Traits* may represent this grouping, thereby implementing a *modular parser* for a given grammar G.

Let be $P_k \in \mathbf{P}$, then the trait $t_k \in \mathcal{T}$ is defined as follows:

- $X \in r(P_k) \implies X \in N$, with $t(X) = \bot$, that is, each required nonterminal X is a required method of t
- $X \in p(P_k) \implies X \in N$ and it will be $t(X) \neq \bot$; in particular, let $p_j \in P_k$ be the j-th rule in P_k of the form $X \to \omega$, such that:

$$p_j = X \to X_{j1} X_{j2} \cdots X_{j\mathbf{n}_j}$$

[3] For a more extensive discussion on the extension and restriction of grammars and parsers, see also [30].

```
object SimpleStateMachine extends StateMachine
  with Body with SimpleState with State
  with SimpleTransition with Transition with JavaTokenParsers {
    type TState = String
    type TTransition = ~[~[String,String],String]
    type TBody = ~[List[TState],List[TTransition]]
    def program = SMachine
}
```

Listing 5: Composition of the traits and type refinement.

then the body of the method named X is a function of all the *right*-hand nonterminals of all the productions with *left*-hand X; and in particular, using the notation in [31], the method named X will be:

$$\begin{aligned} t(X) = & X_{11} \bullet X_{22} \bullet \cdots \bullet X_{1\mathbf{n}_1} \\ & \cup X_{21} \bullet X_{22} \bullet \cdots \bullet X_{2\mathbf{n}_2} \\ & \cup \quad \vdots \\ & \cup X_{m1} \bullet X_{m2} \bullet \cdots \bullet X_{m\mathbf{n}_m} \end{aligned}$$

where " $\bullet$ " denotes the parser combinator for *sequence*, "$\cup$" denotes the parser combinator for *alternative* choice (*union*), and each $X_{ij} \in \mathcal{N}$ when it is a *nonterminal* (that is, it is also $X_{ij} \in N$); if $X_{ij} \in \Sigma^+$, then it is represented using the parser combinator that matches the character sequence X_{ij}. Finally, assuming EBNF grammars, the grammar formalism will also include the quantifiers '?', '$*$' and '+'. In this case, for each quantifier in the grammar, there will be an equivalent combinator in the method body.

Example. We previously showed (Listing 2) the grammar for our simple state machine language. In Listing 4 we translated the four sets of production sets into *traits*: `StateMachine`, `State`, `Transition` and the `Body` of a `StateMachine`. Scala's traits require to i) declare the undefined members, and ii) declare the return types of these members. *Abstract types* make it possible to declare and use a type that will be defined only when the trait will be mixed-in to a concrete class. This feature will particularly come in handy when we will later introduce the semantics of the language implementation. As a convention, abstract types in our code will always start with `T`. The composition of the first version of the language (parser-only) is shown in Listing 5. At this stage, we finally *have* to define the abstract types `TBody` as a list of `State`s and a list of `TTransition`s where `TState` is defined as a `String` and `TTransition` as a triplet of `String`s[4]. Notice how we also compose the library-provided `JavaTokenParser` trait, which provides a definition for `ident`: a parser combinator that matches an identifier, and returns it as a `String`.

[4]This strange type signature is the result of how Scala represents the return type of the "~" parser combinator; we will not discuss these details here; for more information see [5].

Conflicts. Suppose that some grammar provides an alternative definition for some concepts. For instance, in the previous example there might be the rules:

$$\texttt{Transition} \rightarrow \texttt{ShortTransition},$$
$$\texttt{ShortTransition} \rightarrow \texttt{Ident ":" Ident "[" Ident "]"}$$

and some trait t would represent them. Now, suppose that the grammar has been partitioned as such:

$$P_k = \{\texttt{Transition} \rightarrow \texttt{ShortTransition}\}$$
$$P_j = \{\texttt{Transition} \rightarrow \texttt{SimpleTransition}\}$$

then we would have two traits t_k, t_j with two alternative, conflicting definitions of `Transition`. This conflict can be resolved by providing a *resolution* (the function $d \triangleright t$ in Sect. 2.3); this can be done at the moment of the composition by overriding the method `transition`:

```
override def transition = simpleTransition | shortTransition
```

where "`|`" is the *alternative* combinator.

4. Trait-Based Semantics Composition

In a typical interpreter or compiler implementation, the *concrete syntax* of a language is mapped onto an *abstract* representation, the *abstract syntax tree* (AST). In a functional programming language, we would usually define an `eval(AstNode)` function that would pattern match on the type of these nodes. In Scala we could write:

```
def eval(n: AstNode): TResult = n match {
  case StateMachine(name, initial, body) ⇒ ...
  case Transition(from, to, name) ⇒ ...
  case State(name) ⇒ ...
}
```

This solution has the limit to *centralize* the implementation of the semantics of our interpreter in the code of such function, thus making the interpreter less modular and configurable, especially with respect to data type extensions: adding a new type of `Transition` requires to modify the actual body of the `eval` function. In an object-oriented context, the usual solution to the problem is to delegate the role of pattern matching to *polymorphism.* In the *interpreter* pattern a language construct is represented as an `AstNode` type with an abstract `eval` method. Each `AstNode` subtype defines a concrete `eval` method, that implements the semantics only for that specific subtype. This, on the one hand, makes it easier to add new node types to the language implementation, but each concrete `AstNode` subtype is then forced to comply with the interface of the base type. On the other hand, we could compose the semantics onto the nodes using *traits.* The object-oriented approach, in terms of required lines of code, is indeed more *verbose* than the functional version, but it leads to an implementation that is more *configurable.* For instance, different traits may provide an implementation of a different *evaluation phase*, and these traits could be independently shared and distributed among different language implementations.

In this section we show how to realize a *modular interpreter* using traits. The key idea is to represent the bare AST node as a pure data structure and then inject

```
package sm.ast

class StateMachine[+T](val name:String, val initial:String, val transitions: List[T])
class Transition(val from:String,val to:String, val name:String)
class State(val name:String)

trait SMVerify {
  def initial:String; def transitions:List[Transition]
  def verify: Boolean = ...
}

trait SMEval { this: StateMachineVerify =>
  def states:List[State]
  def transitions:List[Transition]
  // evaluate and return the final state iff the sm is well-formed
  def eval: Option[State] =
    if (verify) Some(doEval) else None
  // find the first fireable transition leaving the initial state and evaluate it;
  // then return the final state name
  def eval: State = {
    val nxtT = transitions.find(t ⇒ t.from == initial && t.fireable)
    val finalStateName = nxtT.get.eval(transitions)
    states.find(s ⇒ s.name == finalStateName).get
  }
  def verify: Boolean
}
```

Listing 6: State Machine AST with *evaluation* and, optionally, *verification* (optional parts are in violet).

the methods implementing the semantics, thereby decoupling the semantics that these methods implement from the abstract data representation. Note that the construction of each AST node and its composition with the traits that implement its semantics (the evaluation phases) can be delegated to a *factory method* (Fig. 2). In Sect. 6 we will describe how this technique compares to the related work [12, 18, 19].

4.1. A Modular Interpreter Pattern

In the interpreter pattern, the collection of the AST node types is represented as a collection of classes $c_0, c_1, \ldots, c_m$; an instance of one of these classes represents a typed node; each node class c_j implements an `eval` method; if a node of type c' is supposed to be a child of a node of type c, then the `eval` method of c is supposed to invoke the `eval` method of its child c'. Using traits, we can decouple the representation of the semantics of the evaluation from the representation of the nodes, by factoring out the `eval` method into a trait t. Thus, each node c may mix-in a trait t with a method `eval`, implementing the semantics for a particular evaluation strategy of the node c. Using traits, it is possible to *configure* each node with a different evaluation strategy, independently. Moreover, assuming that the semantics of a language may be represented by several *phases*, i.e., *visits* of the AST, and that, in the interpreter pattern, such phases could be represented by different methods, for each class node c it is possible to compose *many* traits, each implementing a different phase. In other words, let c be a class representing an AST node, and let $t_0, t_1, \ldots, t_{n-1}$ be a sequence of n evaluation phases, represented as traits, respectively providing the methods $\texttt{eval}_0, \texttt{eval}_1, \ldots, \texttt{eval}_{n-1}$. The AST node can be then

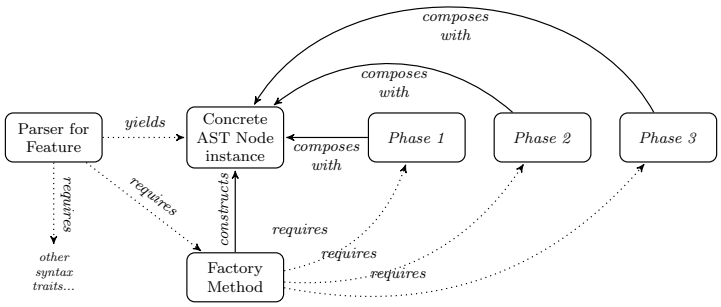

Figure 2: AST Node composition

composed using the trait composition operation "+":

$$t_0 + t_1 + \cdots + t_{n-1}$$

- Each trait t_k may *require* members of c. For instance, if c declares a member n, then it might be that $t_k(n) = \bot$. The members of c that each t_k may require represent values in the AST node class, prevalently children of that AST node. In a statically typed context, traits would also impose constraints on the typing of such children.
- A trait t_k may also *depend* on a particular evaluation phase, by *requiring* a method $\texttt{eval}_j$, effectively imposing that a trait t_j such that $t_j(\texttt{eval}_j) \neq \bot$ will be eventually mixed-in to c.

The evaluation of each AST node will then consist in invoking in sequence the $\texttt{eval}_k$ methods that have been composed onto the node object instance.

Example. In Listing 6 we present the AST types that we use in the state machine language. The AST classes are not required to expose methods to evaluate the language semantics. Instead, the semantics can be implemented separately in traits that will be mixed-in at construction-time. A state machine might be *compiled*, *interpreted*, *verified* for correctness (*e.g.,* check that all states and transitions are reachable), etc. All these concerns can be easily represented as separate traits. The semantics of the nodes can then be composed onto the AST nodes at construction time. For instance, if we assume that the *evaluation* of `StateMachine` is implemented by the `SMEval` trait, and the *verification* is in the `SMVerify` trait, then we might write:

```
new StateMachine(smName,start,trans) with SMVerify with SMEval
```

The evaluation of the abstract tree is then similar to the application of a standard *interpreter* pattern, by invoking the $\texttt{eval}_k$ method, for each implemented phase; alternatively a *main* phase might be defined that invokes in sequence the required phases.

```
def eval = { eval_0(args_0); eval_1(args_1); ...; eval_{n-1}(args_{n-1}) }
```

Each trait will have visibility only to the members of the node that are relevant to the particular evaluation phase that they implement, by *requiring* a member to be available

(*e.g.,* `transitions`, see Listing 6). A trait may also declare dependency on a particular *evaluation phase*: this can be primarily expressed using abstract member declarations (*e.g.,* `verify`) and, in Scala, possibly using explicitly-typed self references (commonly referred to as the *cake pattern* [28]): traits may specify that their self-type should be an instance of a particular trait. For instance, if trait `SMEval` were to declare its willing to depend on a *verification* phase (in violet, in Listing 6), it may require the `verify: Boolean` method and/or declare the *self-type* `SMVerify`. In the first case the trait is only requiring that a member `verify` with that signature will be available at construction time, while the self-type requires that *exactly* a particular interface has been composed onto the base class at construction time. Because the state machine is inconsistent when `verify` is `false`, the `eval` method may return an `Option` that is `None` when the verification step has failed. In the online version of this example[5] we implemented an error reporting mechanism using the `Either` monad.

4.2. AST Configuration

Now, only one detail has been left aside, namely, how the parser should construct the node instances, by composing the phase-specific traits to the semantic-agnostic AST representation. We can impose that each trait that yields an AST node *requires* a *factory method* for that node to be present. For instance, if the method n should return an AST node of type N, then t' should *require* a factory method `newN` that returns a new node instance of type N. In the previous section, we showed a construction to represent a partition $\mathbf{P}$ of the production set P of a grammar G as a collection of traits. Using *abstract types*, *type refinements* and *abstract members* we were able to represent the parser as a collection of components, that could be recomposed at will. At this point, we would like to refine the implementation so that the parser returns AST nodes instead of a parse tree. However, in Scala it is not possible to extend the previous traits and arbitrarily *override* their members, *changing* their return and argument types (Sect. 2.3): like in Java, argument types can be contravariant and return types can be covariant. Nevertheless, it is possible to *substitute* a trait t with a *refined* trait t' that includes alternative member implementations, with the desired signatures.

In general, if t is a trait with a member n, R is the return type of the method n, and we want to override it with a member with return type R', incompatible with R, then we can define a new t' such that the its method n is a function that returns a value of type R. In our case, the method n for which we would like to change the return type will require an adequate factory method to be present.

Example. The method `sm.lang.Transition#transition` yields a `Parser[TTransition]` (Listing 4), and it is defined in terms of the parser `simpleTransition` with type `~[~[String, String], String]`, that is, a triplet of `String`s. Now consider the function `Transition` in Listing 7. We can define the new trait `AstTransition` with a different `transition` method defined as the combination of `simpleTransition` with a function that returns a `Transition` node. The function is defined as a *factory method*, whose implementation is provided by the trait `AstFactory` (Listing 7). By applying the same pattern, we are also able to attach a function to the `statemachine` method to return a `Parser[StateMachine]`.

[5] `neverlang.di.unimi.it/tblc.tgz`.

```
trait AstFactory {
 ...
 type TTransition = sm.ast.Transition
 def Transition(frm:String,to:String,nm:String)=
   new sm.ast.Transition(frm,to,nm)
}
trait AstTransition {
  ...
  def transition = super.simpleTransition ^^ {
    case from ~ to ~ name ⇒ Transition(from, to, name)
  }
}
```

Listing 7: Snippet from the new version of the state machine language, where the parser yields an AST. In blue, the factory method.

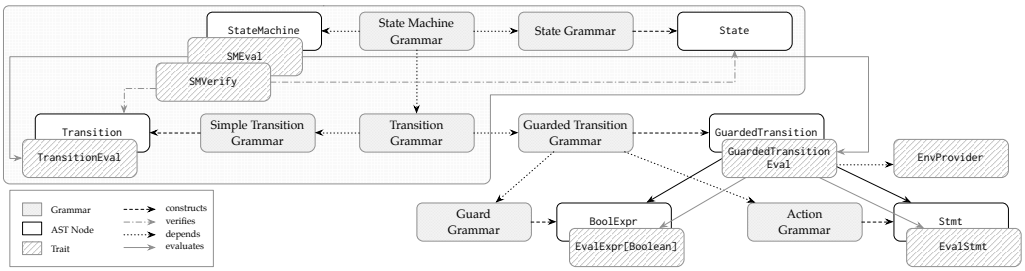

Figure 3: Relations between components in the State Machine case study. The *constructs* relation indicates usage of a factory method.

4.3. Semantic Dependencies

The factory method mechanism can be generalized to express further *semantic dependencies*. For instance a method may *require* that a specific service is available to the language implementation, such as a support library providing utility functions, or a data structure containing shared state information. Suppose that a method n in a trait t uses a method n' defined by some interface i'. Then, t may *require* that a valid instance of i' is available to t by declaring a member m in t such that m returns one such instance of i'. For instance, consider again our state machine example, and suppose that some phase `CodeGen` defines a method **`def`** `compile(File):Boolean`; that compiles the state machine to disk and returns a boolean that indicates if the operation succeeded. The trait `CodeGen` may then require a member **`def`** `fsLibProvider: FsLib` for some interface `FsLib` that declares methods for file system interaction. The desired `FsLib` instance could then be provided by a concrete implementation of the `fsLibProvider` method, possibly defined in a different trait (similarly to what we saw for the `AstFactory`), that would act as a singleton provider.

5. Case Study

The main point of a DSL is to describe the solution of a domain problem *concisely*, and the purpose of componentizing a language implementation is to reuse part of its

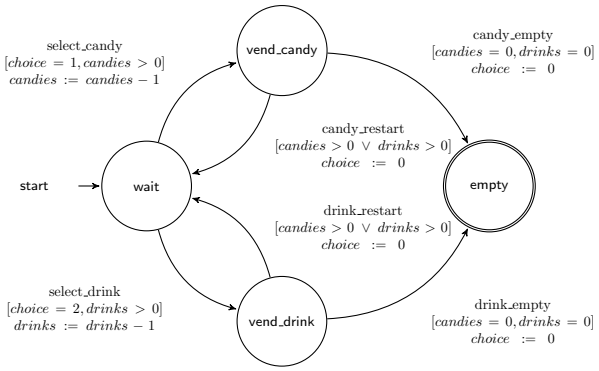

Figure 4: `Vending Machine` state machine.

features in different language implementations. This is particularly useful when put in the perspective of *evolving* a DSL implementation. In the previous sections we introduced the state machine DSL as our running example. In Sect. 3 we showed that, given a partition over the grammar of a language, it is possible to *componentize* the parser for that language if we represent as *traits* the sets of that partition, and we gave instructions to realize this representation. Such a modularization of the parser makes it possible to atomically *extend* and *restrict* the syntax of a language implementation, by mixing-in or excluding traits from the language implementation. Likewise, the trait-based interpreter pattern presented in Sect. 4, makes it simpler to extend a language implementation with new *data types*, and new *evaluation phases*. In this section we will expand further on our running example, making it our *case study*. We will take the state machine DSL and *evolve* it by plugging in a new feature, borrowing part of its implementation from a distinct language: we will substitute the simple transitions of the running example with an alternative implementation, with support for *guards* and *actions*. This example is an adaptation from [25], but, in contrast to this work, in our case the language for guards and actions is implemented as a standalone language. We want to show that using our approach, it is easy to plug the trait-based implementation of the syntax and the semantics of this separate language into the initial state machine language, and to *substitute* part of the original implementation with new components. The advantage of this approach is that all of the extensions and updates that we apply to the original implementation, do not require to actually act on its code; rather, each new feature is implemented as a separate component. In Fig. 3 we show the situation of the state machine language so far, and the extension that we will describe in this section. The part on the left, in the grey box, represents the components that have been already described, while the part on the right will be presented in this section. In Fig. 4, we show the state chart for the vending machine that we will use as an example usage of the extended state machine language (code in Listing 8). For the sake of clarity, we have not included the complete source code for this example in this paper, but it is available from the online version.

```
state machine VendingMachine initial wait {
  state wait state vend_candy
  state vend_drink state empty
  transition from wait to vend_candy: select_candy
   [ choice = 1 && candies > 0 ]{ candies:=candies-1; }
  transition from vend_candy to wait : candy_restart
   [ candies > 0 || drinks > 0 ]{ choice := 0; }
  transition from vend_candy to empty : candy_empty
   [ candies = 0 && drinks = 0 ]{ choice := 0; }
  ...
}
```

Listing 8: Code for the `Vending Machine` in Fig. 4. Code for `drinks` is omitted, since it mirrors the `candies` side.

5.1. Action Language

Executable UML models include the specification of an *action language* [32] that can be used for many purposes, such as expressing actions and guards in a state machine model. In this example, let us suppose that we already have an implementation of a suitable language for this purpose, that is a simple, imperative programming language with support for variables and assignment statements. The parser and the AST types for this language may come as a *bundle* of self-contained components that may be reused within other DSLs. In Listing 9 we show the data types for a part of this language. The semantics of a suitable *interpreter* could be implemented with the same technique that we employed in the previous sections, using traits to represent features of the language as components. As in typical small programming languages, the evaluation of an expression language requires the definition of some `Env` type, representing the environment; two base traits (`EvalExpr[T]` and `EvalStmt`, Listing 9) represent the contract for the behavior of expressions and statements of the language. An *expression* such as `Sum` would yield a result of some type `T`, a *boolean expression* such as `Eq` would be a comparison between objects of type `T` yielding a truth value, and a statement (such as variable assignment) would produce a change in the environment. The environment may be represented as a map `env : String → Int`, that is, a binding between variable names and their current value. For instance, the `VarDef` statement assigns (*e.g.,* `x :=1 + y`) the result of an (integer) expression to a variable identifier. Each trait represents only one small aspect of the semantics of the language, and each trait may require only those members of the node that are required for that particular piece of semantics. The composition of semantics onto the AST nodes happens using the *factory method* technique described in Sect. 4.2.

5.2. Guards and Actions

In Fig. 4 and Listing 8 we are showing the state chart of a *vending machine*. The machine vends `drinks` and `candies`, depending on an initial `choice`, which is an integer value—that is, `1` for candies, `2` for drinks, and `0` for neither. Once a candy or a drink has been vended, the machine resets the `choice` to `0`, and it goes back to the initial *waiting* state, unless both candies and drinks are unavailable, in which case the machine goes to the *empty* state. The example requires us to introduce the concepts of *variable*, *guard* and *action* to transitions: the *guard* is a *boolean expression* that causes a transition to fire only when it evaluates to true, an *action* is a sequence of statements of the action language

```
trait IntExpr
class Sum[T](val x:T, val y:T) extends IntExpr
trait BoolExpr
class Eq[T](val x:T, val y:T) extends BoolExpr
trait Stmt[+T]
class VarDef[T](val x:String,val n: T) extends Stmt[T]

trait Eval {
  type Env = Map[String,Int]
}
trait EvalExpr[T] extends Eval {
  def eval(env:Env): T
}
trait EvalStmt extends Eval {
  def eval(env:Env): Env
}
    ...
trait SumEval extends EvalExpr[Int] {
 def x:EvalExpr[Int]; def y:EvalExpr[Int]
 def eval(env:Env) = x.eval(env) + y.eval(env)
}
trait VarDefEval extends EvalStmt {
 def x: String; def n: EvalExpr[Int]
 def eval(env:Env) = env.updated(x, n.eval(env))
}
trait AstFactory {
    ...
 def Sum(x:IntExpr, y:IntExpr)   = new Sum(x,y) with SumEval
 def VarDef(x:String, n:IntExpr) = new VarDef(x,n) with VarDefEval
}
```

Listing 9: Part of the Action Language AST and Traits for Action Language Evaluation. In blue, the factory methods.

that are executed when a transition fires, and a *variable* is an identifier that is associated with an integer value. In Fig. 3 we show the relations between the new *guarded transition* components, the affected components of the original state machine implementation, and the traits of the action language that are plugged into the guarded transition components to implement the guards and action concerns. These components substitute the transition-related components of the running example (*e.g.,* `GuardedTransition`), and bridge the basic state machine language with the components from the action language (*e.g.,* `BoolExpr` and `Stmt`).

Syntax. *Guarded transitions* may optionally specify a *guard* and an *action* (Fig. 4), thus it subsumes `SimpleTransition` (Listing 4). Because of the modular implementation, it is possible to *unplug* the old component from the parser implementation and introduce the new `GuardedTransitionSyntax` trait (Listing 10), which *extends* `SimpleTransition`. This trait requires a valid definition of the `TAction` and `TGuard` abstract types: these types can be concretized so that the action language that we described in the previous section can be plugged in. For instance we can introduce the `ActionLang` trait, containing the full implementation of the action language for simplicity; of course, each component may be also introduced individually. In Fig. 3 we only show the `BoolExpr` and `Stmt` data types, with the related traits. At this point, the parser would be able to recognize the full

```
trait GuardedTransitionSyntax extends sm.lang.SimpleTransition {
  type TGuard; type TAction; type TTransition
  def guardedTransition: Parser[TTransition] =
    super.simpleTransition ~ guard.? ~ action.? ^^ {
      case from~to~id~g~a ⇒ GuardedTransition(from,to,id,g,a)
  }
  def guard = "[" ~> boolExpr <~ "]"
  def action = "{" ~> stmtList <~ "}"
  // requires:
  def boolExpr:Parser[TGuard]
  def stmtList:Parser[TAction]
  // factory method
  def GuardedTransition(f:String,t:String,id:String,g:Option[TGuard],a:Option[TAction]): TTransition
}
```

Listing 10: Guarded Transition trait. In red, the *syntactic* dependencies (nonterminals of the grammar); in blue, the *semantic* dependencies (factory methods for the AST nodes).

description of a *transition with guards*. Let us then generate the new data types with their semantics.

Semantics. We need to define the `GuardedTransition` data type (Listing 10); this AST node differs from `Transition` data type found in the basic state machine (Listing 6), in that it optionally contains a *guard* and an *action*. Had we defined the interpreter using pattern matching (see Sect. 4), we would have had to *rewrite* the `eval(AstNode)` function. Using the trait-based approach we can implement the new `GuardedTransitionEval` (see Listing 11) *separately* and mix it in the new language implementation, leaving the other parts of the language untouched. For instance, a new `GuardAstFactory` should be provided so that the new `GuardedTransition` node type will mix-in the new trait:

```
new GuardedTransition(from,to,id,guard,action) with GuardedTransitionEval
```

The semantics for *guard* and *action* is implemented respectively by the `EvalExpr[Boolean]` and `EvalStmt` traits (Fig. 3) from the *action language*, which provides `eval(env:Env)`, where `Env` represents an *environment*. Thus, the `GuardedTransitionEval` trait should be able to invoke these methods with suitable arguments. We can express the dependency on this concept using a *semantic dependency* (Sect. 4.3). The `GuardedTransitionEval` trait will *require* a method `envProvider` of type `EnvDef` (Listing 11). The `EnvDef` interface includes methods to *retrieve* and *update* an instance of type `Env`. The `eval` method of the new transition type, retrieves the `env` instance with `envProvider.env` and passes it down to the `guard`, implemented as an `Option[StmtEval]` and the `action` (represented as an `Option[EvalExpr[Boolean]]`): once the computation of the action has been performed, it then updates `env` instance. Using this technique the only part of the code that changed is the *junction point* between the state machine language and the action language; that is, only the traits that pertain to the guarded transition component. The interface of the `StateMachine` node has not changed, and the composed *evaluation* trait is still the same, with the same `eval` method. The guarded transition implementation only relies on the presence of an `EnvProvider` instance. This provider can be initialized to drive the execution of the state machine. In the vending machine example, the wait state is the initial state. The `eval` method of the guarded transition fires the next transition and

```
trait EnvDef      {  def env:Env; def env_=(e:Env) }
trait EnvProvider {  def envProvider:EnvDef = ...  }
trait GuardedTransitionEval extends TransitionEval with EnvProvider {
  def from: String; def to: String;
  def action: Option[StmtEval];
  def guard: Option[EvalExpr[Boolean]]
  // describe whether the transition is fireable;
  // the transition is always fireable when the guard is None
  def fireable = guard.map(_.eval(envProvider.env)).getOrElse(true)
  // returns the name of the final state
  def eval(transitions:List[TransitionEval]): String = {
    println("entering state " + from)
    // update the environment
    val env = envProvider.env
    // if action is None, env does not change
    val newEnv = action.map(stmt ⇒ stmt.eval(env)).getOrElse(env)
    envProvider.env = newEnv
    val next = transitions.find(t ⇒ t.from == to && t.fireable)
    if (next.isEmpty) this.to
    else next.get.eval(transitions)
  }
}
```

Listing 11: Evaluating a Transition with guards.

executes the corresponding optional action when a specified guard evaluates to true (or unconditionally if no guard is specified). For instance, by initializing the `envProvider` as follows:

```
envProvider.env = Map("choice" -> 1, "candies" -> 1, "drinks" -> 0)
```

The final state will be empty (Fig. 4), and the environment will result in `choice` $\mapsto 0, $`candies` $\mapsto 0, $`drinks` $\mapsto 0$.

6. Evaluation and Comparison with Related Work

Many component-based language development frameworks have been proposed over the years (e.g., [11, 33–35]). These frameworks emphasize the separation of the concepts of a language as pluggable and composable units, but do not rely on a particular host language; rather, they provide a programmable *platform* to implement external DSLs; some of them, even provide IDEs and generate IDEs for the implemented languages. For this work we took inspiration from Neverlang [15, 20, 21], where each unit usually represents a syntactic feature of the language (a keyword, or a construct) along with the implementation of its semantics. These units are called *modules* and *slices*. Modules declare the syntax of a feature, that is a portion of a grammar of the language in BNF, and may provide the definition of several `roles`, i.e., the implementation of an evaluation phase, with respect to that part of the syntax. All the roles together represent the *semantics* of the construct. A slice then selects the *syntax* definition and the *semantic* roles and composes them together. The composition of all the slices yields the full implementation of a language interpreter or compiler. Modules and slices can be compiled separately and distributed in pre-compiled form. The final objective is being able to

represent a language as a selection of heterogeneous pre-packaged features [9, 23, 24]. In the approach that we presented, syntax definitions are represented as grammar-like traits using parser combinators (Sect. 3), *roles* are represented by the evaluation *phases* encoded by traits, and slices can be roughly seen as the equivalent of *factory methods* (Sect. 4.2) that describe the configuration of an AST node. In Neverlang, a slice may also describe the mapping between the concrete syntax and the abstract syntax (in fact, in a Neverlang DSL, the two may often overlap, making the mapping implicit). With this work we wanted to show that the Neverlang model of modular language implementation can be easily reproduced and implemented and used within other frameworks achieving the same degree of language decomposition and language component reusability.

In general, the problem of componentizing GPLs and DSLs is a long-standing issue that over the years has been explored far and wide. In the functional domain, many authors have described techniques to represent interpreters for programming languages in a modular way. We have compared the proposed technique with some of work that can be found in the literature. Since most work focuses on the implementation of the *semantics* of a language, rather than the syntax definition, we will assume that in every case it is possible to write a parser using parser combinators, following the technique described in Sect. 3. We will therefore focus on the modular implementation of the semantics of a DSL. In particular, we will consider the case of the *action language* that we used in state machine DSL, with particular attention to the *expression* part, since it is small but it is known to be challenging enough, and most works address the implementation of an expression language as a running example. In order for the comparison to be as fair as possible, we considered the Scala implementation of each technique.

Monad Transformers. Wadler's original paper on *monads* [13] in Haskell describes a way to represent an interpreter for a simple expression language in a purely functional context. Within this context, monads are useful to represent stateful computations or computations that may fail. Liang *et al.* [14] showed how *monad transformers* can be employed in Haskell as *building blocks* of an interpreter. Both Wadler's and Liang *et al.*'s approaches minimize the amount of code that is required to extend the interpreter with new logic. Later, Martin Grabmüller [36] wrote a step-by-step tutorial that describes the technique for modern-day Haskell. These techniques at a first glance might look *opposed* to the work we have presented, but they are actually *orthogonal*. Being Scala a hybrid between an object-oriented and a functional programming language, many functional patterns can be easily *translated* into Scala. For instance, Debasish Gosh, author of the book DSLs in action [37], has adapted Liang *et al.*'s technique for Scala using the Scalaz library[6]. The code at the GitHub repository[7] is a full implementation of an expression language using Liang *et al.*'s technique. Monad transformers make it easier to separate evaluation phases (Sect. 4). Using monad transformers it is possible to *compose* the function with new behavior, while respecting the original type signature of the function. However, the composition of the new behavior actually *requires* to reformulate the implementation of the function from scratch. Moreover, employing pattern matching has the downside of limiting extensibility on the side of data type cases.

[6] `debasishg.blogspot.it/2011/07/monad-transformers-in-scala.html`.

[7] `github.com/debasishg/monad-trans`.

Modular Visitor and Object Algebras. Oliveira *et al.* have proposed [12, 19] solutions to the expression problem [38] in two forms. The first is an *extensible visitor* pattern where both the dimensions of data-type and evaluation phase extensibility are considered. The visitor pattern is generally regarded as the object-oriented rendition of pattern matching; it has therefore the same downsides of pattern matching in functional programming languages, that is, it is possible to add new language processors, but not to extend the data types. This rendition of the technique uses traits for composing the semantics. In *object algebras*, the semantics of language constructs can be built up using trait composition, and instance creation can be abstracted using factories. The technique implements an extension of the factory-based mechanism that we presented in Sect. 4 to *compose* behavior of the language interpreter. Instead of explicitly invoking the constructor of the corresponding instance, the authors describe *combinators* to *compose* the traits. These traits can be defined inline, through anonymous classes, or defined separately to improve reuse, similarly to what we described in Sect. 4. The most important feature of this technique is the use of combinators to abstract the composition of the behavior of the interpreter. Without these combinators, the technique is similar to the one we describe: on the other hand, relying on combinators raises the bar of the language requirements, making it less easy to implement in simpler programming languages. For instance, lack of higher-kinded types would rule out Java as an implementation language.

Trait-Based Composition Through Shadowing. The technique described by Zenger and Odersky [18] combines trait composition with *member shadowing*, a feature of the Scala programming language which makes it possible to hide members of a class or trait from the inheritors. This feature is different from *overriding* since it involves inner type definitions such as inner classes and inner traits, for which speaking of overriding would be incorrect. The usage of this peculiar feature of the Scala programming language makes the technique quite tailored to the choice of the programming language, limiting its use.

Summary. Overall, there are quite a few works that deal with the problem of language extensibility, some of which use traits. Traits guarantee a better reuse of features, and make it possibile to better modularize the implementation both along the dimension of language constructs and that of evaluation phases. Besides monad transformers, which in their purest form *do not* use trait-based composition, but pattern matching, all the other techniques that we evaluated are on par with the features they provide in terms of reusability and composition (Table 3), because the basic objectives of separate compilation and composability are the same. There are however differences in the requirements that each of these features imposes on the host language in order to support them. Most of the techniques rely on features of the host language that are often unavailable in most mainstream programming languages (Table 2) such as *higher-order polymorphism* and *pattern matching*. This makes adopting these techniques, in part or completely, less practical: some programming languages are not even powerful enough to express them.

On the other hand, even though we chose Scala for our implementation, the technique that we describe should be easily portable to any language, provided i) that it supports traits (either as a library, or as a native construct) and ii) that there is a parser combinator library. These are not really restrictive constraints, as today many mainstream programming languages are implementing traits or trait-like constructs. For instance, even Java 8 has introduced *default* implementations for members of an interface, which makes them

	Traits	Higher-Order Polymorphism	Pattern Matching	Member Shadowing	Monads
Modular Interpreter (Sect. 4)	✓				
Monad Transformers		✓	✓		✓
Object Algebras	✓	✓			
Modular Visitor	✓	✓			
Traits+Member Shadowing	✓			✓	

Table 2: Comparison between features in the host language, by technique. More requirements mean that the technique may be harder to implement in simpler programming languages.

	Parser Integration	Sep. Compilations	Data-Type Ext.	Eval. Phases
Modular Interpreter (Sect. 4)	✓	✓	✓	✓
Monad Transformers				✓
Object Algebras		✓	✓	✓
Modular Visitor		✓	✓	✓
Traits+Member Shadowing		✓	✓	✓

Table 3: Comparison between features of the technique.

quite close to traits, and there are combinator libraries that would be suitable for the same purpose. In this work we chose to use Scala's trait implementation because a static type system can be used to guarantee that the composition of the language is correct at *compile time*; nonetheless, the original implementation of traits [27] was developed for Squeak Smalltalk. Thus, similar results can be achieved using Smalltalk, equipped with a parsing library such as PetitParser [8]. Other dynamic languages, such as Python, Ruby, Perl and JavaScript support parser combinators through libraries, and traits or trait-like constructs natively or as add-ons; thus, even in such cases, the same technique can be implemented.

This work is geared towards the modularization of a language implementation over two dimensions: the *syntactic* dimension and the *semantic* dimension. With respect to the first dimension, our approach makes it possible to define *modular parser* implementations, by reasoning only on the structure of the language grammar. In particular, in our work we are giving guidelines on how to componentize a grammar, and how the specification of the parser, in the form of a trait-based grammar decomposition, may then construct the abstract representation of the interpreter through composition of factory methods. In Sect. 5 we showed that this approach simplifies the extension of the syntax of the language, possibly reusing components from different languages.

From the *semantic* standpoint, our approach has a number of benefits, but it does involve quite a bit of boilerplate. In particular, for every data type, it is necessary to define a trait for each *evaluation phase* and a *factory method* to configure the composition between the AST data types and the traits that implement their evaluation phases. Nevertheless boilerplate is a necessary evil to address the modularity concern: a bit of boilerplate is in fact required in all of the other techniques. Of course, in a comparison that involved only the count of lines of code, pattern-matching would win hands down: in this case, an evaluation function would just need to specify the case matches for each

data type. On the other hand, the known problem of this approach is that the pattern matching approach is harder to extend with new data types (*e.g.,* see [18]). The challenge is to find an approach where i) the AST implementation should be *extensible* with new subtypes ii) it is possible to add new processing phases, iii) existing code is not modified or duplicated, iv) it is possible to compile each extension separately and v) to combine together independently developed extensions. However, much of the required boilerplate could be auto-generated using naming conventions. For instance, for each AST data type `T` the trait for the evaluation phase might be called `TEval`. On the other hand, the conciseness that is lost with this approach is traded for a much greater extent of extensibility and reuse: separate compilations and reuse make it possible to mechanize language composition, to the point where a language implementation may be reduced to configuring a selection of components (cf. [9, 23, 24]).

Looking instead at Erdweg *et al.*'s taxonomy in [10], the proposed approach clearly supports language extensions and restrictions. Self-extension and unification are not considered even if feasible in this work because, in our view, they do not represent the common case for language evolution. The unification consists of merging two languages in one so that elements of the first are integrated to those of the other and vice versa; no host language is present. In our approach the *unification* of two languages decomposed as suggested consists only on adjusting some grammar rules so that the final grammar result fully connected; all the language features are at the end at the same level and interpreted by the same interpreter. This approach is quite different from some recent approaches (such as [39, 40]) to language unifications that let every original language element be interpreted by the original interpreter and only their interactions are dealt with separately. Self-extension or language embedding occurs when a language is embedded into another by simply using the language constructs already available (such as [41, 42]); the advantage of this approach is that the original interpreter remains unchanged. This basically is what our approach proposes in order to achieve language extension.

7. Conclusions

DSL development is an aspect that modern GPLs have been emphasizing more and more. In this work, we exploited well-known patterns, techniques and constructs to implement external DSLs with a high degree of flexibility and modularity. The final objective is being able to implement DSLs by combining components together, maximizing code reuse and minimizing duplication. The approach revolves around the use of *traits* both for the realization of the *parser* of the DSL and for the implementation of the semantics of the *interpreter*, and because of the guarantees of correctness that static typing provides, we chose to employ Scala's trait implementation. Nevertheless, the assumptions and reasoning we made to pursue our results in Scala should make our approach reproducible in other mainstream programming languages that provide trait-like composition capabilities. Our final objective is to be able to pursue a model of componentization that simplifies the implementation of *feature-oriented* programming languages, possibly using variability modeling techniques to present choices to end users, following up to our previous experiences on the subject matter [9, 23, 24].

References

[1] M. Mernik, J. Heering, A. M. Sloane, When and How to Develop Domain Specific Languages, ACM Comput. Surv. 37 (4) (2005) 316–344.

[2] M. Fowler, R. Parsons, Domain Specific Languages, Addison Wesley, 2010.

[3] T. Kosar, S. Bohra, M. Mernik, Domain Specific Languages: A Systematic Mapping Study, Information and Software Technology 71 (2016) 77–91.

[4] M. Fowler, Fluent Interface, Martin Fowler's Blog (May 2005).
URL http://martinfowler.com/bliki/FluentInterface.html

[5] M. Odersky, L. Spoon, B. Venners, Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition, Artima Inc., 2011.

[6] D. J. P. Leijen, E. Meijer, Parsec: Direct Style Monadic Parser Combinators for the Real World, Technical Report UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands (2001).

[7] A. Moors, F. Piessens, M. Odersky, Parser Combinators in Scala, CW Report 491, Katholieke Universiteit Leuven, Leuven, Belgium (Feb. 2008).

[8] L. Renggli, S. Ducasse, T. Gîrba, O. Nierstrasz, Practical Dynamic Grammars for Dynamic Languages, in: Proceedings of the 4$^{\text{th}}$ Workshop on Dynamic Languages and Applications (DYLA'10), Málaga, Spain, 2010.

[9] E. Vacchi, W. Cazzola, S. Pillay, B. Combemale, Variability Support in Domain-Specific Language Development, in: M. Erwig, R. F. Paige, E. Van Wyk (Eds.), Proceedings of 6$^{\text{th}}$ International Conference on Software Language Engineering (SLE'13), Lecture Notes on Computer Science 8225, Springer, Indianapolis, USA, 2013, pp. 76–95.

[10] S. Erdweg, P. G. Giarrusso, T. Rendel, Language Composition Untangled, in: A. Sloane, S. Andova (Eds.), Proceedings of the 12th Workshop on Language Description, Tools, and Applications (LDTA'12), ACM, Tallinn, Estonia, 2012.

[11] H. Krahn, B. Rumpe, S. Völkel, MontiCore: A Framework for Compositional Development of Domain Specific Languages, International Journal on Software Tools for Technology Transfer 12 (5) (2010) 353–372.

[12] B. C. d. S. Oliveira, T. van der Storm, A. Loh, W. R. Cook, Feature-Oriented Programming with Object Algebras, in: G. Castagna (Ed.), Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13), Lecture Notes in Computer Science 7920, Springer, Montpellier, France, 2013, pp. 27–51.

[13] P. Wadler, Monads for Functional Programming, in: J. Jeuring, E. Meijer (Eds.), Advanced Functional Programming, LNCS 925, Springer, Bøastad, Sweden, 1995, pp. 24–52.

[14] S. Liang, P. Hudak, M. Jones, Monad Transformers and Modular Interpreters, in: R. K. Cytron, P. Lee (Eds.), Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95), ACM, San Francisco, CA, USA, 1995, pp. 333–343.

[15] E. Vacchi, W. Cazzola, Neverlang: A Framework for Feature-Oriented Language Development, Computer Languages, Systems & Structures 43 (3) (2015) 1–40. doi:10.1016/j.cl.2015.02.001.

[16] N. Schärli, S. Ducasse, O. Nierstrasz, A. P. Black, Traits: Composable Units of Behaviour, in: L. Cardelli (Ed.), Proceedings of the 17$^{\text{th}}$ European Conference on Object-Oriented Programming (ECOOP'03), Lecture Notes in Computer Science 2743, Springer, Darmstadt, Germany, 2003, pp. 248–274.

[17] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. P. Black, Traits: A Mechanism for Fine-Grained Reuse, ACM Transactions on Programming Languages and Systems 28 (2) (2006) 331–388.

[18] M. Zenger, M. Odersky, Independently Extensible Solutions to the Expression Problem, in: Proceedings of the 12th International Workshop on Foundations of Object-Oriented Languages (FOOL'12), Long Beach, CA, USA, 2005.

[19] B. C. d. S. Oliveira, Modular Visitor Components: A Practical Solution to the Expression Families Problem, in: S. Drossopoulou (Ed.), Proceedings of the 23$^{\text{rd}}$ European Conference on Object-Oriented Programming (ECOOP'09), Lecture Notes in Computer Science 5653, Springer, Genoa, Italy, 2009, pp. 269–293.

[20] W. Cazzola, Domain-Specific Languages in Few Steps: The Neverlang Approach, in: T. Gschwind, F. De Paoli, V. Gruhn, M. Book (Eds.), Proceedings of the 11$^{\text{th}}$ International Conference on Software Composition (SC'12), Lecture Notes in Computer Science 7306, Springer, Prague, Czech Republic, 2012, pp. 162–177.

[21] W. Cazzola, E. Vacchi, Neverlang 2: Componentised Language Development for the JVM, in: W. Binder, E. Bodden, W. Löwe (Eds.), Proceedings of the 12$^{\text{th}}$ International Conference on

Software Composition (SC'13), Lecture Notes in Computer Science 8088, Springer, Budapest, Hungary, 2013, pp. 17–32.
[22] E. Vacchi, D. M. Olivares, A. Shaqiri, W. Cazzola, Neverlang 2: A Framework for Modular Language Implementation, in: Proceedings of the 13th International Conference on Modularity (Modularity'14), ACM, Lugano, Switzerland, 2014, pp. 23–26.
[23] E. Vacchi, W. Cazzola, B. Combemale, M. Acher, Automating Variability Model Inference for Component-Based Language Implementations, in: P. Heymans, J. Rubin (Eds.), Proceedings of the 18th International Software Product Line Conference (SPLC'14), ACM, Florence, Italy, 2014, pp. 167–176.
[24] T. Kühn, W. Cazzola, D. M. Olivares, Choosy and Picky: Configuration of Language Product Lines, in: G. Botterweck, J. White (Eds.), Proceedings of the 19th International Software Product Line Conference (SPLC'15), ACM, Nashville, TN, USA, 2015, pp. 71–80.
[25] L. Tratt, Evolving a DSL Implementation, in: R. Lämmel, J. Visser, J. Saraiva (Eds.), Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering II (GTTSE'07), LNCS 5235, Springer, Braga, Portugal, 2008, pp. 425–441.
[26] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, Reading, Massachusetts, 1986.
[27] N. Schärli, Traits — Composing Classes from Behavioral Building Blocks, Phd thesis, Universität Bern, Bern, Switzerland (Feb. 2005).
[28] M. Odersky, M. Zenger, Scalable Component Abstractions, in: R. P. Gabriel (Ed.), Proceedings of 19$^{\text{th}}$ ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'05), ACM Press, San Diego, CA, USA, 2005, pp. 41–57.
[29] M. Mernik, An Object-Oriented Approach to Language Compositions for Software Language Engineering, Journal of Systems and Software 86 (9) (2013) 2451–2464.
[30] W. Cazzola, E. Vacchi, On the Incremental Growth and Shrinkage of LR Goto-Graphs, Acta Inf. 51 (7) (2014) 419–447. doi:10.1007/s00236-014-0201-2.
[31] M. Might, D. Darais, Yacc is dead, CoRR abs/1010.5023.
[32] S. J. Mellor, S. Tockey, R. Arthaud, P. Leblanc, An Action Language for UML: Proposal for a Precise Execution Semantics, in: J. Bézivin, P.-A. Muller (Eds.), Proceedings of the first Workshop on The Unified Modeling Language («UML»'98), LNCS 1618, Springer, Mulhouse, France, 1998, pp. 307–318.
[33] E. Visser, WebDSL: A Case Study in Domain-Specific Language Engineering, in: R. Lämmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering II, LNCS 5235, Springer, 2008, pp. 291–373.
[34] T. Ekman, G. Hedin, The JastAdd System — Modular Extensible Compiler Construction, Science of Computer Programming 69 (1-3) (2007) 14–26.
[35] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an Extensible Attribute Grammar System, Electronic Notes on Theoretical Computer Science 203 (2) (2008) 103–116.
[36] M. Grabmüller, Monad Transformers Step by Step, draft paper (October 2006).
[37] D. Ghosh, DSL for the Uninitiated, Commun. ACM 54 (7) (2011) 44–50.
[38] P. Wadler, The Expression Problem, Java Genericity Mailing List (Nov. 1998).
[39] E. Barrett, C. F. Bolz, L. Tratt, Approaches to Interpreter Composition, Computer Languages, Systems & Structures 44 (Part C) (2015) 199–217.
[40] M. Grimmer, C. Seaton, T. Würthinger, H. Mössenböck, Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages, in: G. Leavens (Ed.), Proceedings of the 14th International Conference on Modularity (Modularity'15), ACM, Fort Collins, CO, USA, 2015, pp. 1–13.
[41] V. Karakoidas, D. Mitropoulos, P. Louridas, D. Spinellis, A Type-Safe Embedding of SQL into Java Using the Extensible Compiler Framework, Computer Languages, Systems & Structures 41 (2015) 1–20. doi:10.1016/j.cl.2015.01.001.
[42] P. G. Giarrusso, K. Ostermann, M. Eichberg, R. Mitschke, T. Rendel, C. Kästner, Reify Your Collection Queries for Modularity and Speed!, in: J. Kienzle (Ed.), Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development (AOSD'13), ACM, Fukuoka, Japan, 2013, pp. 1–12.