Under consideration for publication in J. Functional Programming

parboiled2: a macro-based approach for effective generators of parsing expressions grammars in Scala

ALEXANDER A. MYLTSEV
IP Myltsev A. A., Chelyabinsk, Russia

Abstract

Parsing is everywhere today in the computer era. Developers parse logs, queries to databases and websites, programming and natural languages, etc. When Java ecosystem maturity, concise syntax, and runtime speed matters developers choose `parboiled2` that generates grammars for parsing expression grammars (PEG). The library uses a wide range of Scala facilities to provide the decent functionality. We explain in details the extensions to PEGs. We show how it is naturally implemented in the intuitive syntax and semantics of the internal Scala DSL. We demonstrate how `parboiled2` extensively uses Scala typing to verify DSL integrity. We then show the connections to inner structures of `parboiled2`. The understanding gives to a developer a better understanding of how to compose more effective grammars. Finally, we expose how a grammar is expanded with Scala Macros to effective runtime code.

Contents

1 Introduction

Computer specialists have been parsing programming languages and protocols since the beginning of computer era. They used Chomsky's generative system of grammars, context-

free grammars (CFGs) and regular expressions (REs), to encode syntax of programming languages and protocols. One of generative grammars purposes was to model natural languages and hence inherit ambiguity in its design. The uncertainty brings unnecessary complexity to parsing of machine languages that are explicit by design. There are several alternatives to CFGs for formally specifying syntax.

Parser combinators [1, 2] are popular for readability, modularity, easiness in maintain. Alas, they cannot be fully used in production. The first reason is that naive implementations do not handle left-recursive grammars. R. Frost et al. gives a solution at [3]. The second one lies in the expressive power that causes runtime inefficiency because of the composition overhead and intermediate data structures creation. M. Jonnalagedda and E. Béguet [4] achieved significant performance speedup by removing overheads and intermediate data representations. They used meta-programming techniques such as macros [5] and staging [6]

Parsing Expression Grammars (PEGs) are another alternative. The difference to CFGs is that PEGs eliminate the ambiguity by *prioritized choice* in recognition-based syntax describing [7]. Practically PEGs make a suitable replacement for REs [8]. PEGs work as fast as REs based parsers (even quicker in some edge cases). The benefit is that PEGs allow natural parsing of sequences that defined recursively (XML, JSON, programming languages, etc.). Finally, PEGs are much easier to read and maintain than REs. recognize left recursion [9], support backtracking [10], and semantic actions [11].

We describe in details the implementation of *parboiled2* library in the paper. *parboiled2* is an implementation of PEGs parsers generators in Scala programming language [12]. The paper contributes in fields as follows: a) give intuition on how to use PEGs with *parboiled2* DSL, b) expose inner structure of the library, c) explain how sophisticated type system tightly connects inner parts, d) describe how macro generates fast runtime code, e) list current limitations of the library.

The paper is structured as follows. Section 2 describes by the example core parts of a simple *parboiled2* grammar.

In Section 4 we give intuition into parsing process and describe its semantics in details. In Section 5 we introduce how *parboiled2* produces side effects with Value Stack. In Section

We introduce high-level domain specific language (DSL) to describe recognition rules. Scala type checker strongly verifies every rule and that composition. And finally expand at compile-time to efficient runtime code with Scala macros [5]. *parboiled2* is assembled as a regular Java Virtual Machine (JVM) library. Any JVM-oriented development environment, profiler, debugger, tracer, etc. can use the library.

2 Implementation of Inner Abstractions

A particular parser should be derived from `Parser` base class to inherit all necessary facilities to parse input string. The `Parser` inheritor expects the `input` of type `ParserInput` in the constructor. *parboiled2* provides implicit conversions from three types to `ParserInput` type: `String` by default, `Array[Char]`, and `Array[Byte]`.

Consider a PEG that recognizes mathematical formulas of four basics operations with precedence to non-negative integers (Figure 1). Corresponding *parboiled2* parser is shown

$$
\begin{array}{lcl}
\mathit{expression} & \leftarrow & \mathit{term}\ (('+'\ /\ '-')\ \mathit{term})* \\
\mathit{term} & \leftarrow & \mathit{factor}\ (('*'\ /\ '/')\ \mathit{factor})* \\
\mathit{factor} & \leftarrow & \mathit{number}\ /\ ('('\ \mathit{expression}\ ')') \\
\mathit{number} & \leftarrow & [0-9]+
\end{array}
$$

Fig. 1. PEG for mathematical formulas of four operations to non-negative integers

```
sealed trait Expr
case class Val(value: String)        extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr
case class Mul(lhs: Expr, rhs: Expr) extends Expr
case class Div(lhs: Expr, rhs: Expr) extends Expr
```

Fig. 2. AST for mathematical formulas grammar

in Figure 3. `CalculatorParser` is a Scala class. It contains composition of rules that determine parsing process. Note that `Expression` (Fig. 3, line 3) has explicit type since it recursively used in `Factor` (Fig. 3, line 9). All rules bodies start with `rule` method call. The call body contains the composition of built-in rules from the DSL and calls to other `CalculatorParser` rules in the scope.

The grammar in Fig. 1 only recognizes if an input string is arithmetical expression. To be useful in practice a parser performs semantic actions such as computing an expression or emitting AST nodes. With underlined code parts on lines 4, 7, 10 (Fig. 3) `CalculatorParser` captures the input parts and produces AST nodes listed in Figure 2.

`CalculatorParser` successfully parses the input string to the result that contains AST nodes in the Scala interpreter as follows:

```
scala> new CalculatorParser("1+(2-3*4)/5").InputLine.run()
res0: scala.util.Try[Expr] =
  Success(Add(Val(1),Mul(Val(2),Val(3))))
```

If parsing fails, it returns `Failure` of `ParseError`. `ParseError` contains all necessary information to print comprehensive string message that describes why parsing failed. `CalculatorParser` gets the parsing error message easily with calls in the fashion:

```
scala> val parser = new CalculatorParser("1+2!3")
```

```
1  class CalculatorParser(val input: ParserInput) extends Parser {
2    def InputLine = rule { Expression ~ EOI }
3    def Expression: Rule1[Expr] = rule {
4      Term ~ ('+' ~ Term ~> Add | '-' ~ Term ~> Sub).*
5    }
6    def Term = rule {
7      Factor ~ ('*' ~ Factor ~> Mul | '/' ~ Factor ~> Div).*
8    }
9    def Factor = rule { Number | '(' ~ Expression ~ ')' }
10   def Number = rule { capture(CharPredicate.Digit.+) ~> Val }
11 }
```

Fig. 3. parboiled2 rules for mathematical formulas

```
scala> val Failure(e: ParseError) = parser.InputLine.run()
e: org.parboiled2.ParseError =
  ParseError(Position(3,1,4), Position(3,1,4), <6 traces>)
scala> parser.formatError(e)
res1: String =
Invalid input '!', expected '/', '+', '*', 'EOI', '-'
or Digit (line 1, column 4):
1+2!3
```

Since PEGs are recognition-based, a parser should define a rule `Expression`∼`EOI` that would force the parser to go to the end of an input string. Otherwise, a parser successfully parses arithmetical expression *"1+2"* until it meets unexpected char *'!'*:

```
scala> new CalculatorParser("1+2!3").Expression.run()
res2: scala.util.Try[Expr] = Success(Add(Val(1),Val(2)))
```

3 Rules DSL

`CalculatorParser` contains composition of elementary rules that are listed in Table 1. Rules are naturally grouped into three categories: `basic`, `combinators`, `semantic actions`. Those categories are defined in corresponding Scala traits `RuleDSLBasics`, `RuleDSLCombinators`, `RuleDSLActions`. `basic` and `combinators` rules are derived from original definition of PEGs [7]. `semantic actions` allow a parser to produce some useful result (like AST of parsed expression).

The first defense frontier against usage errors of the library is the `rule` macro. We designed every *parboiled2* rule call to exist only within `rule` macro scope. If a rule is called somewhere outside of the macro, Scala compiler fails with an error. Practically every rule has an annotation that prevents it to exist at compile-time. `rule` macro erases rule calls by expanding their composition to a runnable code.

The second defense frontier is the type system that helps to verify if a `rule` can actually be run against the input. For example, `Expression` has a type stating it returns an AST node of type `Expr`. Hence, the entire rule composition of `Expression` body should be of type `Expr`. We will show more sophisticated examples in Section 5.

4 Semantics of Parsing

PEG parsers are recursive-descent parsers with backtracking. Most parsers produced by traditional parser generators like ANTLR have two parsing phases. PEGs have only one parsing phase. PEGs do not require any look-ahead and perform quite well in most real-world scenarios. Although certain pathological languages implemented in PEGs and inputs exhibit exponential runtime.

When the runner executes a rule against the current position in an input buffer, the rule applies its specific matching logic to the input. When a `Parser` calls `rule` method, it creates an instance of `ParserState` class that stores reference to the input and `cursor` of type `Int`. The `cursor` points to the next unmatched input character. In success case of parsing by a rule, the parser advances the cursor and potentially executes the following

Table 1. *parboiled2 rules*

Rule category	Rule	PEG operator	Description
Basic rules	ch	' '	Literal character
	str	" "	Literal string
	CharPredicate[a]	[ ]	Character class
	ANY[b]	.	Any character
	( )	(e)	Grouping
Combinator rules		$e_1 \sim e_2$	Sequence
		$e_1 \mid e_2$	Prioritized Choice (First Of)
		optional(e)	Optional
		zeroOrMore(e)	Zero-or-more
		oneOrMore(e)	One-or-more
Semantic actions		$\rightsquigarrow_{\mathbf{f}}$	Action operator
		push	Push to value stack
		pop	Pop from value stack

[a] An efficient implementation composable of character sets. It comes with a number pre-defined character classes like *CharPredicate.Digit* or *CharPredicate.LowerHexLetter*

[b] Generalized to $anyOf(chars : String)$ that matches any char of provided ones. $noneOf(chars : String)$ is an inversion of $anyOf$ – fails on any of provided chars

rule. Otherwise, when the rule fails, the cursor is reset to the last successful match. And the parser backtracks in search of another parsing alternative that might succeed.

For example, consider the simple *parboiled2* rule:

```
def foo = rule { 'a' ~ ('b' ~ 'c' | 'b' ~ 'd') }
```

When the rule attempts to match against the input `"abd"`, the parser does steps as follows:

1. Rule `foo` starts executing, which calls its first sub-rule `'a'`. The cursor is at position 0.
2. Rule `'a'` is executed against input at position 0, matches (succeeds), and the cursor advances to position 1.
3. Rule `'b'~'c'|'b'~'d'` starts executing, which calls its first sub-rule `'b'~'c'`.
4. Rule `'b'~'c'` starts executing, which calls its first sub-rule `'b'`.
5. Rule `'b'` is executed against input position 1, matches (succeeds), and the cursor advances to position 2.
6. Rule `'c'` is executed against input position 2 and mismatches (fails).
7. Rule `'b'~'c'|'b'~'d'` notices that its first sub-rule has failed, resets the cursor to position 1 and calls its second sub-rule `'b'~'d'`.
8. Rule `'b'~'d'` starts executing, which calls its first sub-rule `'b'`.
9. Rule `'b'` is executed against input position 1, matches, and the cursor advances to position 2.
10. Rule `'d'` is executed against input position 2, matches, and the cursor advances to position 3.
11. Rule `'b'~'d'` completes successfully, as its last sub-rule has succeeded.
12. Rule `'b'~'c'|'b'~'d'` completes successfully, as one of its sub-rules has succeeded.

1. (8) `push` accepts one or more arguments and immediately pushes to `ValueStack`
2. (9) `capture` accepts a rule as the single argument. If the provided rule is succeeded to match then captured part of the input is pushed to `ValueStack`.
3. (10) $\xrightarrow{\mathbf{f}_n}$ ("action expression") of arity n, pops values $v_1, v_2, \ldots, v_n$ (v_n is assigned to first popped value, v_{n-1} – to the next popped, etc.) from `ValueStack`, applies a given function $\mathbf{f}_n(v_1, v_2, \ldots, v_n)$, and pushes the result back to `ValueStack`.
   Note the inversive order of sequential pops. This is done because `HList` type grows from right to left, and `parboiled2` is extensively built on it. This makes it easier to keep in mind a memo: last value on `ValueStack` is passed as a parameter to last argument of a function **f**.
4. (11) `zero-or-more(e,` $\xrightarrow{\mathbf{f}_2}$`)` might be defined using binary "action expression" becoming "reduction expression". While `e` matches the input, reduction `zero-or-more` pops one value from the `ValueStack` after every successful match of `e`, applies $\mathbf{f}_2$ to it, and puts the result back to the `ValueStack`.

Fig. 4. Extension to PEG: Semantic actions

13. Rule `foo` completes execution successfully, as its last sub-rule has succeeded. The whole input `"abd"` was matched, and the cursor is left at position 3 (after the last-matched character).

5 Parsing Actions and Value Stack

parboiled2 parses same class of inputs as Scala combinator parsers. The primary difference lies in the way they produce the result of parsing. Every Scala combinator parsers grammar is a composition of functions: they always produce a result that is then passed as an argument to another parsing functions. The problem is that parsing produces plenty intermediate and mostly redundant data structures that cause extra calls of memory allocations and garbage collections in JVM. L. Haoyi et al. [13] solve it by writing effective runtime Scala code. M. Jonnalagedda [14] uses compile-time staging to eliminate redundant data structures creation. *parboiled2* introduces a particular data structure called `ValueStack`: the library user should decide how to manipulate intermediate parsing structures.

`ValueStack` is a mutable extension of `Iterable[Any]` that implements untyped stack of values. `Parser` creates fresh new instance of `ValueStack` upon every start rule run. It is a private member of `ParserState` of `Parser's` internals and is not intended to be used directly. We extend the list of parsing expressions proposed in [7] with `semantic actions` to operate on `ValueStack` in Figure 4. We extend relation $\Rightarrow_G$ in Figure 5: from triples of the form (e, x, S) to triples of the form (n, o, S'), where e, x, n, o are defined in [7]. S indicates the state of the `ValueStack` before a matching attempt. S' the state after a matching attempt.

6 Type System

parboiled2 uses Scala type system in two neat ways:

1. *parboiled2* start rule might return a result of one of three types based on imported `DeliveryScheme` implicits.
2. *parboiled2* verifies access to `ValueStack` based on types of rules. It allows to avoid most of the inconsistent states of `ValueStack`.

1. **Standard expression**: $(e,x,S) \Rightarrow (n,o,S)$. Any standard expression doesn't change the state of `ValueStack`.
2. **Push**: $(push(v),x,S) \Rightarrow (1,o,S.push(v))$
3. **Capture (success case)**: If $(e,x,S) \Rightarrow (n,o,S)$ then $(capture(e),x,S) \Rightarrow (n+1,o,S.push(o))$
4. **Capture (failure case)**: If $(e,x,S) \Rightarrow (n,f,S)$ then $(capture(e),x,S) \Rightarrow (n+1,f,S)$
5. **Action expression (success case)**: If $(e,x,S) \Rightarrow (n,o,S)$ then n values are popped from `ValueStack`: $v_n = S.pop(), v_{n-1} = S.pop(), \ldots v_1 = S.pop()$ and $(e \rightsquigarrow_{\mathbf{f}},x,S) \Rightarrow (n+1,o,S.push(\mathbf{f}(v_1,v_2,\ldots v_n))$
6. **Action expression (failure case)**: If $(e,x,S) \Rightarrow (n,f,S)$ then $(e \rightsquigarrow_{\mathbf{f}},x,S) \Rightarrow (n+1,f,S)$
7. **Zero-or-more reduction (repetition case)** If $(e,x_1x_2y,S) \Rightarrow (n_1,x_1,S.push(x_1))$ and $(e^{*(\xrightarrow{\mathbf{f}_2})},x_1x_2y,S) \Rightarrow (n_2,x_1x_2,S.push(x_1).push(x_2))$ then $(e^{*(\xrightarrow{\mathbf{f}_2})},x_1x_2y,S) \Rightarrow (n_1+n_2+1,x_1x_2,S.push(\mathbf{f}_2(S.pop(),x_1)).push(\mathbf{f}_2(S.pop(),x_2)))$
8. **Zero-or-more reduction (termination case)** if $(e,x,S) \Rightarrow (n_1,f,S)$ then $(e^{*(\xrightarrow{\mathbf{f}_2})},x,S) \Rightarrow (n_1+1,\varepsilon,S)$

Fig. 5. Extension to relation $\Rightarrow_G$

6.1 DeliveryScheme of Parsing Result

`run` method launches the start rule of a `Parser` against the provided input string. Parsing then might end in one of three possible ways:

- `success` if a parser successfully matches input. In that case, parsing result should hold an instance subclass of shapeless HList
- `parseError` if parser fails to match against given input. In that case parsing should return `parboiled2.ParserError` that contains information on why parsing failed
- `error` if parser failed for internal an reason (division by zero, index out of range, etc.). In that case parsing returns an instance of *scala.Throwable* subclass

parboiled2 supports three ways to deliver `success/failure` result: `scala.util.Try`, `scala.Either`, and simply throw an exception. We abstract it to `Result` (embedded in `DeliveryScheme`) that has three instances – one per type of result. Fig. 6 shows implementation of `DeliveryScheme` for `scala.util.Try` result type.

`run` method implicitly accepts the particular instance of `DeliveryScheme` available in the scope of calling. It then internally wraps success or failure result by calling `scheme` instance methods.

6.2 Rule Types

Parsing process changes `ValueStack` as a side effect. Naive parsing might lead `ValueStack` to an inconsistent state. For example, a rule might try to pop a value from an empty stack, or cast popped value to a wrong type. Scala type system prevents many invalid operations at the type-checking phase of compilation.

We attach extra type information to `Rule` that keeps track on how it intends to change the `ValueStack`. `Rule` is isomorphic to Scala function: it accepts input of particular type from `ValueStack` values and produces an output of another type that pushes to `ValueStack`. From that perspective `Rule` is defined in the same way as a regular function:
`class` `Rule[-I <:HList, +O <:HList]`, where `I` and `O` are types of input and output correspondingly. For example, parser rules of the type `Rule[Int::String::HNil,`

```
trait DeliveryScheme[L <: HList] {
  type Result
  def success(result: L): Result
  def parseError(error: ParseError): Result
  def failure(error: Throwable): Result
}

implicit def Try[L <: HList] = new DeliveryScheme[L] {
  type Result = Try[L]
  def success(result: L) = Success(result)
  def parseError(error: ParseError) = Failure(error)
  def failure(error: Throwable) = Failure(error)
}

def run()(implicit scheme: Parser.DeliveryScheme[L]): scheme.Result = {
  val result: HList = // ...
  scheme.success(result)
}
```

Fig. 6. Varying result type of `run` configurable by implicitly provided deliver scheme

`String::HNil]` are only allowed to pop from `ValueStack` value of type `Int`, then of type `String` (note the order: `Int` is first), and push a value only of type `String`.

Basic rules are not intended to change the `ValueStack` (Fig. 1). They have **`type`** `Rule0 = Rule[HNil, HNil]`.

Action rules change the *ValueStack* in a straightforward way. `capture` and `push` can only push values to `ValueStack`. `push` rule pushes a value of any type unconditionally. `capture` rule expects that provided inner rule matches, and only then it pushes matched string. It is the moment where type-level computation happens: both `capture` and `push` either decrease `I` if it's not `HNil`, or append to the output type `O` of a parent rule.

The complimentary `drop` rule unconditionally pops and throws away one or more values from `ValueStack`. It either decreases `O` type if it's not `HNil`, or extend the input type `I` of a parent rule.

Action operator $\rightsquigarrow_{\mathbf{f}}$ might either decrease or prepend additional type to either `I`, or `O`. It depends on the relation of how many arguments it simultaneously intends to pop and push to `ValueStack`.

More sophisticated type-level computations stands behind *rule combinators*. `Sequence` combinator should check whether type O_{LHS} of left-hand-side (LHS) rule is compatible with I_{RHS} of right-hand-side (RHS) rule, i.e., check if LHS rule pushes values of types that RHS expects to pop from the `ValueStack`. If O_{LHS} and I_{RHS} are of different sizes, the `sequence combinator` then checks either I_{LHS} or O_{RHS}, if it can handle the "larger" rule.

The special case of *rule combinators* is `reduction rule`. Consider a common scenario of reducing the input string to a single value on the `ValueStack`. The rule `Factor` from Fig. 3 is extended to handle multiplication operation:

```
(Factor: Rule1[Int]) ~
  zeroOrMore('*' ~ Factor ~> ((a: Int, b) => a * b))
```

`zeroOrMore` hosts two operations inside:

```
def capture[I <: HList, O <: HList](r: Rule[I, O])
  (implicit p: Prepend[O, String :: HNil]): Rule[I, p.Out]

def push[T](value: T)(implicit h: HListable[T]): Rule[HNil, h.Out]

object HListable extends LowerPriorityHListable {
  implicit def fromUnit: HListable[Unit] { type Out = HNil } = `n/a`
  implicit def fromHList[T <: HList]: HListable[T] { type Out = T } = `n/a`
}
abstract class LowerPriorityHListable {
  implicit def fromAnyRef[T]: HListable[T] { type Out = T :: HNil } = `n/a`
}
```

Fig. 7. Type signature of basic rules

1. it matches `Factor` expression that pushes the value of type `Int` according to its type
2. then precisely in the same iteration of `zeroOrMore` it pops **two** values from the `ValueStack`, and pushes those values multiplication back to it

The type of inner rule is `Rule[Int::Int::HNil, Int::HNil]`. Since `Int::HNil` is nested to `Int::Int::HNil`, type of `zeroOrMore` is computed to `Rule[Int::HNil, Int::HNil]`

In total, starting from empty `ValueStack` and intending to leave it empty or push some values to it, a custom rules composition of `Rule` types mutually fulfill constraints:

- parsing ends with no values on `ValueStack`, i.e., the grammar recognizes an input. Or parsing stops with one or more values on `ValueStack`
- a rule that pops values of some types from `ValueStack` provides handling function of the same types
- none of the rules tries to pop a value if `ValueStack` is empty

Worth to mention that Scala erasures all type information during compilation. It means there is no overhead of any sophisticated types casts at runtime.

Next, we describe some rules in details. To keep the size of the paper reasonable, we do not cover type signatures of all basic rules. We describe a few simple ones to give the intuition on how to read rest of rules.

6.2.1 capture

`capture` (Figure 7) pushes the matched string on `ValueStack`. `capture` takes any valid rule `r: Rule[I, O]` as argument. `capture` result input type is the same as *r*'s input type. `capture` prepends `String` to `r`'s output type: `O :: String :: HNil` in pseudonotation. *shapeless*' higher kinded `Prepend` type of argument types *O* and *String* :: *HNil* computes `capture`'s output type and put it to *p.Out*.

6.2.2 push

`push` (Figure 7) doesn't depend on any inner rule. It pushes a value of arbitrary type `T`. The complication arises from cases of what type *T* might be:

```
@tailrec def rec(L, LI, T, TI, R, RI) =
  if (TI <: L) R
  else if (LI <: T) RI.reverse ::: R
  else if (LI <: HNil) rec(L, HNil, T, TI.tail, R, RI)
  else if (TI <: HNil) rec(L, LI.tail, T, HNil, R, LI.head :: RI)
  else rec(L, LI.tail, T, TI.tail, R, LI.head :: RI)
rec(L, L, T, T, R, HNil)
```

Fig. 8. Type-level implementation of `sequence` output type computation

- in case of $Unit$ nothing is pushed. *push* that tries to handle value of $Unit$ type is equivalent to calling *run*
- $T <: HList$. Then all values of HList are pushed as individual elements
- a single value of any other type T is pushed as is

This pattern match on type level is implemented in `parboiled2.support.HListable` type as follows. `parboiled2` defines three implicits with appropriate *Out* types for each case: $Unit$, $T <: HList$ and low-priority $AnyRef$. Depending on type T and implied type $HListable[T]$, corresponding implicit with *Out* type would be given to value h of *push*. Defining $fromAnyRef$ as $LowerPriorityHListable$ prevents its being given as implicit for h of any type.

6.2.3 sequence

`sequence` matches when both left-hand-side (LHS) and right-hand-side (RHS) rules are matched. It implies that LHS and RHS rules on `ValueStack` should be compatible on the type level. There are three possible cases:

- when both rules pop nothing from Value Stack. No matter what they push to it (even no values) the result would be a concatenation of O_{LHS} and O_{RHS}. In types (using abbreviated HList pseudonotation) it is encoded: $Rule[,A] \sim Rule[,B] = Rule[,A:B]$
- $Rule[A:B:C,D:E:F] \sim Rule[F,G:H] = Rule[A:B:C,D:E:G:H]$ type describes the case when LHS rule pushes enough values to be popped by RHS rule, no matter what LHS rule actually pops and RHS pushes. The result type should be as it pops LHS values, and rightmost values of LHS rules that are equivalent to popped values of RHS rule wiped out and replaced by pushed values of RHS rule
- and the last case is when RHS rule pops more values than LHS has pushed. In that case, the result rule demands missing values to pop and leaves pushed values as is. The encoded type is $Rule[A,B:C] \sim Rule[D:B:C,E:F] = Rule[D:A,E:F]$.

Type-level implementation of the algorithm is listed at Figure 8.

7 Code Generation

When the Scala compiler ensures that rules composition has valid types, it expands `rule` macros to the code that would be run at runtime. Next we'll describe all steps from rule definition to its code generation.

Consider the rule:

```
val arule = rule { "ab" }
```

rule is a Scala method defined as:

```
def rule[I <: HList, O <: HList](r: Rule[I, O]): Rule[I, O] =
  macro ParserMacros.ruleImpl[I, O]
```

"*ab*" of type *String* is neither instance nor subtype of type $Rule[I,O]$. Scala compiler succeeds in finding the implicit: for that purpose `parboiled2` defines implicit conversion from *String* type to *Rule*0 in *Parser* class:

```
@compileTimeOnly("Calls to ‘str‘ must be inside ‘rule‘ macro")
implicit def str(s: String): Rule0 = ‘n/a‘
```

After being applied, the implicit turns *arule* to:

```
val arule: Rule0 = rule { SimpleParser.this.str("ab"): Rule0 }
```

Both `@compileTimeOnly` and ‘`n/a`‘ (a method that throws `IllegalStateException`) of *rule* method guards runtime execution from leaking of *str* method: if execution path somehow riches *str* at runtime, it would throw an exception. Macro definition of *rule* method evaporates all definitions of *str* method.

rule method calls $ParserMacros.ruleImpl[I,O]$. *ruleImpl* is a special kind of method: it takes Scala AST as input, transforms it, and returns the transformed Scala AST:

```
def ruleImpl[I <: HList: ctx.WeakTypeTag,
             O <: HList: ctx.WeakTypeTag]
  (ctx:ParserContext)
  (r: ctx.Expr[Rule[I, O]]): ctx.Expr[Rule[I, O]] = {
    // ...
    val opTreeCtx = new OpTreeContext[ctx.type] {
      val c: ctx.type = ctx
    }
    opTreeCtx.OpTree(r.tree)
    // ...
  }
```

The dedicated trait *OpTree* handles all possible rules transformations. *ruleImpl* creates its instance and passes the AST *r.tree* to it. *opTreePF* method transforms a definition in the grammar to actual code:

```
val opTreePF: PartialFunction[Tree, Tree] = {
  // ...
  case q"$a.this.str($s)" =>
    q"""
      val matched =
        input.sliceString(cursor, cursor + $s.length) == $s
      if (matched) cursor += $s.length
      matched
    """
```

```
  // ...
}
```

When case pattern is applied to the expression *SimpleParser.this.str*("*ab*"), the values of *a* and *s* on the right hand side would respectively be *SimpleParser* and "*ab*". A naïve implementation should do three things:

1. compare input slice to the string "*ab*"
2. if the input matches then advance the *cursor* (Chapter 3) further to the length of "*ab*"
3. return *Boolean* result of the match

opTreePF matches not only primitives, but complex rules operands as well. Consider *firstOf* rule that is naturally coded as follows:

```
val opTreePF: PartialFunction[Tree, Tree] = {
  case q"$lhs.|[$a, $b]($rhs)" =>
    q"""
      val cursorCurrent = cursor
      if (lhs()) true
      else {
        cursor = cursorCurrent
        if (rhs()) true
        else {
          cursor = cursorCurrent
          false
        }
      }
    """
}
```

lhs and *rhs* are rules that might be composed of primitives, other combinators, and other rule calls. In the end, they are callable and return *Boolean*. For example, in case of

```
("a" ~ "b") | arule
```

the values of *lhs* and *rhs* would be `("a" ~ "b")` and `arule` respectively.

7.1 Optimizations

Naïve implementation generates a string slice on every match attempt. A possible optimization towards efficient implementation would be char-by-char comparison in imperative style:

```
case q"$a.this.str($s)" => q"""
  var ix = 0
  while (ix < $s.length && cursorChar == $s.charAt(ix)) {
    ix += 1
    cursor += 1
  }
  ix == $s.length"""
```

The next optimization step comes from the observation that in most cases a grammar contains domain specific string literals known at compile time. A string literal, say "*abc*", is unrolled to the nested stack of if/else-s as follows:

```
if (cursorChar == 'a') {
    cursor += 1
    if (cursorChar == 'b') {
        cursor += 1
        cursorChar == 'c'
    } else false
} else false
```

Speaking generally, *opTreePF* in first turn checks if *s* is literal string, and then apply *unroll* function that generates if/else-s cascade:

```
def unroll(s: String, ix: Int = 0): Tree =
  if (ix < s.length) q"""
    if (cursorChar == ${s charAt ix}) {
      cursor += 1
      ${unroll(s, ix + 1)}
    } else false
  """ else q"true"

case q"$a.this.str($s)" => s match {
  case Literal(Constant(sc: String)) => unroll(sc)
  case _ => // imperative code for general string match
```

Note how *unroll* mixes code generation logic with assertions of what *s* is at compile-time.

parboiled2 applies a few more of optimizations as follows:

- flatten a tree of *sequence* rules series
- same technique for $firstOf$ rules series
- character sets (*CharPredicate*). They allow effectively determine if in an input character is in a set. parboiled2 comes with plenty of predefined sets (like *CharPredicate.Digit* and *CharPredicate.Alpha*), and allows to define it from a function $Char \rightarrow Boolean$

7.2 Code Generation Limitation

The general limitation in wider spread of effective code generation and optimizations lies in the nature of Scala macros: *rule* macro is only able to analyze the scope of a single method. Consider the grammar:

```
val arule  = rule { "a" }
```

Theoretical obvious optimization of *aarule* is to inline *arule* and squash sequence of two "*a*"s to the single string "*aa*". But practically *opTreePF* only sees the *arule* call without any non-hackable way to get the AST of *arule* body.

8 Catching Parsing Errors

An important part of the parsing process is error reporting: to tell why parsing failed and at what position. Scalac generates code only once during the compilation. The exact same code should parse input and tell whether it fails to parse and why. The process distinguishes two major phases:

- the parsing phase:

  ```
  if (phase0_initialRun())
    scheme.success(valueStack.toHList[L]())
  ```

  If it successfully finishes, *run* returns the top value on the *ValueStack* wrapped in successful result of delivery scheme (Chapter 6.1)
- if it fails, next phases upon run determine error index of the input and collect rule traces. Each phase respects rules that are marked as quiet. Finally, *parserError* is wrapped in error result of delivery scheme:

  ```
  else {
    val principalErrorIndex: Int =
      phase1_establishPrincipalErrorIndex()
    val parseError: ParseError = // rest phases
    scheme.parseError(parseError)
  }
  ```

A rule should return the tracing information when execution path reaches it. There is no code yet that preserves tracing information (Chapter 7). A rule code generation is encapsulated in a reciprocal class. The class has two versions of code rendering: for parsing phase and for error collecting phase. Consider *CharMatch* class for the basic char rule:

```
case class CharMatch(charTree: Tree) extends TerminalOpTree {
  def ruleTraceTerminal =
    q"org.parboiled2.RuleTrace.CharMatch($charTree)"
  def renderInner(wrapped: Boolean): Tree = {
    val unwrappedTree = q"cursorChar == $charTree && __advance()"
    if (wrapped)
      q"$unwrappedTree && __updateMaxCursor() || __registerMismatch()"
    else unwrappedTree
  }
}
```

The *unwrappedTree* has kind of code described in Chapter 7. The addendum is that *CharMatch* renders based on *wrapped* flag. And the wrapped version should either update the max cursor if it matches, or register a mismatch. *TerminalOpTree* implements mismatch registration and error tracing information.

9 Further Work

Obstacles to wider optimizations have the root in narrow scope of rule macro application, as we mentioned in Section 7.2. Particularly, it blocks cross-rule optimizations and indi-

rectly increases code base. For example, B. Ford [7] theoretically showed that *oneOrMore*, *option* and *and − predicate* operators are "syntactic sugar", i.e. combination of other operators can substitute them. Staging and compilation techniques [6] might evaporate intermediate data creation. But they require much wider scope. And `parboiled2` should explicitly implement "syntactic sugar" operators individually for the sake of runtime effectiveness.

Single method code generation by macro also limits a code block to handle all facilities (like debugging and tracing). This would potentially blow up a method code size (limited by JVM), complicates code base support, and lessens `parboiled2` versions back compatibility.

Another problem arises from the fact that *rule* macro depends on a context it doesn't control. For example, a rule might be assigned either to *val* or *def*. Both approaches have pros and cons. But we should make design decisions that define inner implementation and library usages. That is another point where backward compatibility suffers.

Creating higher-ordered rules (a method that takes another rule as a parameter) is also impossible with the current version of Scala Macros.

The origins of *ValueStack* arise from the inefficiency of combinator approaches – they produce too many intermediate data structures. First bad thing is `parboiled2` shifts side-effect result composition too broad on a developer's shoulders. Hence, again we're tightened to hard with the API interface and backward compatibility. The second bad thing is *ValueStack* type-based verification is good for a user when type check passes. If a user makes a mistake somewhere in typing (i.e. missed an argument in lambda for action operation), she would get tens of lines of machine-generated typing errors that are really hard to interpret. M. Jonnalagedda et al. [14] showed how to eliminate intermediate data structures automatically.

All described limitations restrict intuitive feature implementation: creating custom rules that need inner API access. For example, it is hard to implement a rule that tracks position coordinates of parsed AST nodes.

The good news that a new version of Scala Macros should be enough to overcome all the obstacles [15].

Bibliography

P. Wadler, "Monads for functional programming," in *International School on Advanced Functional Programming*, pp. 24–52, Springer, 1995.

A. Moors, F. Piessens, and M. Odersky, "Parser combinators in scala," 2008.

R. A. Frost, R. Hafiz, and P. C. Callaghan, "Modular and efficient top-down parsing for ambiguous left-recursive grammars," in *Proceedings of the 10th International Conference on Parsing Technologies*, IWPT '07, (Stroudsburg, PA, USA), pp. 109–120, Association for Computational Linguistics, 2007.

E. Béguet and M. Jonnalagedda, "Accelerating parser combinators with macros," in *Proceedings of the Fifth Annual Scala Workshop*, pp. 7–17, ACM, 2014.

E. Burmako, "Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming," in *Proceedings of the 4th Workshop on Scala*, SCALA '13, (New York, NY, USA), pp. 3:1–3:10, ACM, 2013.

T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Acm Sigplan Notices*, vol. 46, pp. 127–136, ACM, 2010.

B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," *SIGPLAN Not.*, vol. 39, pp. 111–122, Jan. 2004.

D. Y. Mozzherin, A. A. Myltsev, and D. J. Patterson, ""gnparser": a powerful parser for scientific names based on parsing expression grammar," *BMC Bioinformatics*, vol. 18, p. 279, May 2017.

S. Medeiros, F. Mascarenhas, and R. Ierusalimschy, "Left recursion in parsing expression grammars," *Sci. Comput. Program.*, vol. 96, pp. 177–190, Dec. 2014.

R. R. Redziejowski, "Parsing expression grammar as a primitive recursive-descent parser with backtracking," *Fundam. Inf.*, vol. 79, pp. 513–524, Aug. 2007.

R. Atkey, "The semantics of parsing with semantic actions," in *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, (Washington, DC, USA), pp. 75–84, IEEE Computer Society, 2012.

M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: Updated for Scala 2.12*. USA: Artima Incorporation, 3rd ed., 2016.

L. Haoyi, "Fastparse." https://github.com/lihaoyi/fastparse, 2014.

M. Jonnalagedda, T. Coppey, S. Stucki, T. Rompf, and M. Odersky, "Staged parser combinators for efficient data processing," *SIGPLAN Not.*, vol. 49, pp. 637–653, Oct. 2014.

F. Liu and E. Burmako, "Two approaches to portable macros," tech. rep., 2017.