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
Independent Researcher, Moscow, Russia
e-mail: alexander@myltsev.com

Abstract

In today's computerized world, parsing is ubiquitous. Developers parse logs, queries to databases and websites, programming and natural languages. When Java ecosystem maturity, concise syntax, and runtime speed matters, developers choose *parboiled2* that generates grammars for parsing expression grammars (PEG). The following open source libraries have chosen *parboiled2* for parsing facilities:

- *akka-http* is the Streaming-first HTTP server/module of Lightbend Akka
- *Sangria* is a Scala GraphQL implementation
- *http4s* is a minimal, idiomatic Scala interface for HTTP
- *cornichon* is Scala DSL for testing HTTP JSON API
- *scala-uri* is a simple Scala library for building and parsing URIs

The library uses a wide range of Scala facilities to provide required functionality. We also discuss the extensions to PEGs. In particular, we show the implementation of an internal Scala DSL that features intuitive syntax and semantics. We demonstrate how *parboiled2* extensively uses Scala typing to verify DSL integrity. We also show the connections to inner structures of *parboiled2*, which can give the developer a better understanding of how to compose more effective grammars. Finally, we expose how a grammar is expanded with Scala Macros to an effective runtime code.

1 Introduction

Computer specialists have been parsing programming languages and protocols since the beginning of the computer era. They used Noam Chomsky's generative system of grammars, context-free grammars (CFGs), and regular expressions (REs) to encode syntax of programming languages and protocols. One of the purposes of generative grammars was to model natural languages and hence to inherit ambiguity in their design. The uncertainty of CFGs brings unnecessary complexity to parsing in machine languages that are explicit by design. There are several alternatives to CFGs to specify syntax formally.

Parser combinators (Wadler, 1995; Moors *et al.*, 2008) are popular due to their readability, modularity, and ease of maintenance, they cannot be fully used in production. The first reason is that naive implementations do not handle left-recursive grammars, unless they are implemented according to a solution given in (Frost *et al.*, 2007). Another reason lies in the expressive power that causes runtime inefficiency because of the composition overhead and the creation of intermediate data structures. A significant performance speedup is given in

(Béguet & Jonnalagedda, 2014) by removing overheads and deleting intermediate data representations. The authors used meta-programming techniques such as macros (Burmako, 2013) and staging (Rompf & Odersky, 2010).

Parsing Expression Grammars (PEGs) are an alternative solution to the parsing problem. The difference with CFGs is that PEGs eliminate the ambiguity by *prioritized choice* in the process of recognition-based syntax describing (Ford, 2004). Virtually, PEGs make a suitable replacement for REs (Mozzherin *et al.*, 2017). PEGs work as fast as REs-based parsers (even faster in some edge cases). The benefit is that PEGs allow natural parsing of sequences that are defined recursively (XML, JSON, programming languages, etc.). Finally, PEGs are much easier to read and maintain than REs, recognize left recursion (Medeiros *et al.*, 2014), support backtracking (Redziejowski, 2007), and semantic actions (Atkey, 2012).

In the paper, we describe the implementation of the *parboiled2* library. The *parboiled2* is an implementation of PEGs parsers generators in the Scala programming language (Odersky *et al.*, 2016). *parboiled2* is assembled as a regular Java Virtual Machine (JVM) library. Any JVM-oriented development environment, profiler, debugger, tracer, etc. can use the library.

The paper a) develops intuition about how to use PEGs with the *parboiled2* DSL, b) exposes inner structure of the library, c) explains tight connections of the library inner parts, d) describes how macro generates a fast runtime code, e) lists current limitations of the library.

Section 2 of the paper describes the core parts of a simple *parboiled2* grammar. Section 3 introduces a high-level domain specific language (DSL) to describe rules of recognition. Section 4 provides insight into the parsing process and describes its semantics in detail. Section 5 explains how *parboiled2* produces side effects with *Value Stack*. Section 6 explains how *parboiled2* uses a Scala type checker to verify every rule and their composition. Section 7 explains the process of step-by-step code generation of macro definitions (Burmako, 2013). Section 8 exposes the way how *parboiled2* catches and handles parsing errors.

2 Implementation of Inner Abstractions

A parser for a particular grammar should be derived from the `Parser` base class to inherit all the 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 (Fig. 1). The corresponding *parboiled2* parser is shown in Fig. 2. `CalculatorParser` is a Scala class. It contains a composition of rules that determine the parsing process. Note that `Expression` (Fig. 2, line 3) has an explicit type since it is recursively used in `Factor` (Fig. 2, line 9). All the rules bodies start with a `rule` method call. The call body contains a composition of built-in rules from the DSL and calls to other `CalculatorParser` rules in the scope.

The grammar in Fig. 1 is only recognized if the input string is an arithmetic expression. To be useful in practice, a parser performs semantic actions such as computing an

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

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

```
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. 2. *parboiled2* rules for mathematical formulas

expression or emitting AST nodes. With the underlined code parts on lines 4, 7, 10 (Fig. 2) `CalculatorParser` captures the input parts and produces AST nodes listed in Fig. 3.

`CalculatorParser` successfully parses the input string, and the result returned 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 the `Failure` of `ParseError` type. `ParseError` contains all the necessary information about errors to print a comprehensive string message that describes why the parsing failed. The following example shows `ParseError` generation by feeding an invalid string to a `CalculatorParser` constructor:

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

```
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. 3. AST for mathematical formulas grammar

```
or Digit (line 1, column 4):
1+2!3
```

Since the PEGs are recognition-based, a parser should define a rule `Expression`$\sim$`EOI` that would force the parser to move to the end of the input string. Otherwise, the parser successfully parses the arithmetic expression *"1+2"* until it encounters an 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 a composition of elementary rules that are listed in Table 1. The rules are naturally grouped into three categories: `basic`, `combinators`, and `semantic actions`. These categories are defined in the corresponding Scala traits `RuleDSLBasics`, `RuleDSLCombinators`, `RuleDSLActions`. `basic` and `combinators` rules are derived from the original definition of PEGs (Ford, 2004). `semantic actions` allow a parser to produce useful results (like the AST of the parsed expression).

parboiled2 directs a developer to program a statically correct grammar with two facilities.

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

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

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, whereas PEGs have only one parsing phase. PEGs do not require any look-ahead, and they perform quite well in most real-world scenarios. However, certain pathological languages implemented in PEGs and inputs exhibit exponential runtime (Ford, 2004).

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 a `rule` method, it creates an instance of a `ParserState` class that stores reference to the input and the `cursor` of the `Int` type. The `cursor` points to the next unmatched input character. In case of successful parsing by rule, the parser advances the cursor and potentially executes the next rule. Otherwise, when the rule fails, the cursor is reset to the last successful match. And the parser backtracks in search for another parsing alternative that might succeed.

Consider this simple *parboiled2* rule:

Table 1. parboiled2 rules

Rule category	parboiled2 rule	PEG operator	Description
Basic rules	ch[a]	' '	Literal character
	str[b]	" "	Literal string
	CharPredicate[c]	[ ]	Character class
	ANY[d]	.	Any character
		EOI	Matches end of line
	( )	(e)	Grouping
	anyOf()		
Combinator rules	$e_1 \sim e_2$	$e_1 \sim e_2$	Sequence
	$e_1 \mid e_2$	$e_1 \mid e_2$	Prioritized Choice (First Of)
	optional(e)	e?	Optional
	zeroOrMore(e)	e*	Zero-or-more
	oneOrMore(e)	e+	One-or-more
	&(e)	&e	And-predicate
	!(e)	!e	Not-predicate
Semantic actions	$\xrightarrow{\mathbf{f}_n}$		Action operator
	push(value)		Pushes the value to the ValueStack
	drop		Drops a value from the ValueStack
	capture(e)		Pushes captured string to the ValueStack

[a] Extended by `ignoreCase(c: Char)` that matches an input char case insensitively
[b] Extended by `ignoreCase(s: String)` that matches an input string case insensitively
[c] An efficient implementation composable of character sets. It comes with a number pre-defined character classes like `CharPredicate.Digit` or `CharPredicate.LowerHexLetter`
[d] 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

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

When the rule attempts to match against the input `"abd"`, the parser performs the following steps:

#1. Rule `foo` starts executing, which calls its first sub-rule `'a'`. The cursor sets to position 0.
#2. Rule `'a'` is executed against the 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 the input position 1, matches (succeeds), and the cursor advances to position 2.
#6. Rule `'c'` is executed against the 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 the input position 1, matches, and the cursor advances to position 2.
#10. Rule `'d'` is executed against the input position 2, matches, and the cursor advances to position 3.

1. `push` accepts one or more arguments and immediately pushes to the `ValueStack`
2. `capture` accepts a rule as a single argument. If the provided rule succeeds to match, then the captured part of the input is pushed to the `ValueStack`.
3. $\xrightarrow{\mathbf{f}_n}$ ("action expression") of arity n, pops values $v_1, v_2, \ldots, v_n$ (v_n is assigned to the first popped value, v_{n-1} – to the next popped, etc.) from the `ValueStack`, applies a given function $\mathbf{f}_n(v_1, v_2, \ldots, v_n)$, and pushes the result back to the `ValueStack`.
   Note the inversive order of sequential pops. We made this design decision to unify the usage of the `ValueStack`. We suggest to keep in mind this memo: the `ValueStack` grows from left to right, and arguments of the function $\mathbf{f}_n$ are assigned from left to right from the most recently pushed values that are at the right of the `ValueStack`.
4. `zero-or-more(e,` $\xrightarrow{\mathbf{f}_2}$`)` can be defined using binary "action expression" that becomes a "reduction expression". While `e` matches the input, the 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

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

5 Parsing Actions and Value Stack

The primary difference between *parboiled2* and Scala combinator parsers 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 function. The problem is that parsing produces plenty of intermediate and mostly redundant data structures that cause extra calls of memory allocations and garbage collections in JVM. (Haoyi, 2014) solved this problem by writing an effective runtime Scala code. (Jonnalagedda *et al.*, 2014) used compile-time staging to eliminate redundant data structures. *parboiled2* introduces a particular data structure called `ValueStack`: the library user should decide how to manipulate intermediate parsing structures.

The `ValueStack` is a mutable extension of `Iterable[Any]` that implements an untyped stack of values. `Parser` creates a fresh new instance of the `ValueStack` upon every start rule run. It is a private member of `ParserState` of `Parser`'s internals and it is not intended to be used directly. We extend seven inductive definitions of *parsing expressions* given in Section 3.1 of (Ford, 2004) with `semantic actions` to operate on the `ValueStack` in Fig. 4. In addition, we extend relation $\Rightarrow_G$ (Fig. 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 (Ford, 2004). S indicates the state of the `ValueStack` before a matching attempt. S' is the state after a matching attempt.

6 Typing of Rule DSL

parboiled2 uses the Scala type system to catch potential problems at the compile time in two ways as follows:

1. **Standard expression**: $(e,x,S) \Rightarrow (n,o,S)$. Any standard expression does not change the state of the `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 the `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 $\Rightarrow_G$

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

6.1 DeliveryScheme of Parsing Result

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

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

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

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

6.2 Rule Types

The parsing process changes the `ValueStack` as a side effect. Naive parsing can lead the `ValueStack` to an inconsistent state. For example, a rule might pop a value from an empty

```
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

stack, or cast a popped value to a wrong type. The 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 functions: it accepts the input of a particular type from the `ValueStack` values and produces an output of another type that pushes to the `ValueStack`. From this 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 the input and the output, respectively. For example, parser rules of the type `Rule[Int::String::HNil, String::HNil]` are only allowed to pop from the `ValueStack` value of the `Int` type, then of the `String` type (note the order: `Int` is first), and push a value only of the `String` type.

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

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

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

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

More sophisticated type-level computations stand behind *rule combinators*. A `Sequence` combinator should check whether type O_{LHS} of the left-hand-side (LHS) rule is compatible

```
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

with I_{RHS} of right-hand-side (RHS) rule, i.e., check if the LHS rule pushes the 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 a "larger" rule.

A special case of *rule combinators* is the so-called `reduction rule`. Consider this common scenario of reducing the input string to a single value on the `ValueStack`. The rule `Factor` from Fig. 2 is extended to handle multiplication operation:

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

`zeroOrMore` hosts two operations inside:

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 the `ValueStack`

The type of the 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 the empty `ValueStack` and intending to leave it empty or push some values to it, a custom rules composition of the `Rule` types mutually fulfill constraints:

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

It is worth mentioning that Scala erases all types information during compilation. It means that there is no overhead of any sophisticated types-casts at runtime.

Next, we will describe some rules in detail. To keep the length of the paper reasonable, we do not cover type signatures of all basic rules. We describe several simple rules to give some intuition on how to read the rest of the rules.

6.2.1 capture

`capture` (Fig. 7) pushes the matched string to the `ValueStack`. The `capture` takes any valid rule `r: Rule[I,O]` as an argument. The `capture` resulting input type is the same as the `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` (Fig. 7) does not depend on any inner rule. It pushes a value of an arbitrary type `T`. The complication arises from the cases of what type `T` might be:

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

This pattern match on the 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 an 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 the LHS and RHS rules on the `ValueStack` should be compatible on the type level. There are three possible cases:

- when both rules pop nothing from the `ValueStack`. 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]`∼`Rule[,B]=Rule[,A:B]`
- `Rule[A:B:C,D:E:F]`∼`Rule[F,G:H]=Rule[A:B:C,D:E:G:H]` type describes the case when the LHS rule pushes enough values to be popped by the RHS rule, no matter what the LHS rule actually pops, and the RHS pushes. The result type should be as it pops the LHS values, and the rightmost values of the LHS rules that are equivalent to the popped values of the RHS rule wiped out and replaced by the pushed values of the RHS rule
- and the final case is when the RHS rule pops more values than the LHS has pushed. In this case, the result rule demands the missing values to pop and leaves the pushed values as they are. The encoded type is `Rule[A,B:C]`∼`Rule[D:B:C,E:F]=Rule[D:A,E:F]`.

The type-level implementation of the algorithm is listed in Fig. 8.

```
@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

7 Code Generation

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

Consider this rule:

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

`rule` on the right of the equal sign is the Scala method defined as:

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

`"ab"` of the `String` type is neither the instance nor the subtype of type `Rule[I,O]`. The Scala compiler succeeds in finding the implicit: for this purpose, *parboiled2* defines implicit conversion from the `String` type to the `Rule0` type 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 the `rule` method guards the runtime execution from leaking of the `str` method: if the execution path somehow reaches the `str` at runtime, it throws an exception. The macro definition of the `rule` method evaporates all the definitions of the `str` method.

The `rule` method calls `ParserMacros.ruleImpl[I,O]`. `ruleImpl` is a special kind of the method: it takes the Scala AST as an 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. The `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 the case pattern is applied to the expression `SimpleParser.this.str("ab")`, the values of `a` and `s` on the right hand side are respectively `SimpleParser` and `"ab"`. The naive implementation should do three things:

1. compare the input slice to the string `"ab"`
2. if the input matches, it advances the `cursor` (Section 3) further to the length of `"ab"`
3. return the `Boolean` result of the match

`opTreePF` matches not only primitives, but complex rules operands as well. Consider the `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 the rules that could 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

The naive implementation generates a string slice on every match attempt. A possible optimization towards the efficient implementation would be a char-by-char comparison in the 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, e.g. `"abc"`, is unrolled to the nested list of if/else-s as follows:

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

Generally, `opTreePF` in the first turn checks if `s` is a literal string and then applies the `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 the code generation logic with assertions of what `s` is at compile-time.

parboiled2 applies a few more optimizations as follows:

- flatten a tree of `sequence` rules series
- same technique for `firstOf` rules series
- character sets (`CharPredicate`). They allow to determine if in the input character belongs to the set. *parboiled2* comes with plenty of predefined sets (like `CharPredicate.Digit` and `CharPredicate.Alpha`), and allows defining it from a function of the `Char -> Boolean` type

7.2 Code Generation Limitation

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

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

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

8 Catching Parsing Errors

An important part of the parsing process is error reporting: to identify why the parsing failed and at what position. The Scala compiler generates a code only once during the compilation. The exact same code should parse the input and inform 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 a successful result of the delivery scheme (Section 6.1)
- if it fails, next phases upon run determine the error index of the input and collect the rule traces. Each phase respects the rules that are marked as quiet. Finally, `parserError` is wrapped in the error result of the delivery scheme:

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

A rule should return the tracing information when the execution path reaches it. There is no code yet that preserves the tracing information (Section 7). A rule code generation is encapsulated in a reciprocal class. The class has two versions of code rendering: for the parsing phase and for the error collecting phase. Consider the `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 a code described in Section 7. The addendum is that `CharMatch` renders based on the `wrapped` flag. And the wrapped version should either update the max cursor if it matches, or register a mismatch. `TerminalOpTree` implements the mismatch registration and the error tracing information.

9 Further Work

Obstacles to wider optimizations originate in the narrow scope of the rule macro application, as mentioned in Section 7.2. Notably, it blocks cross-rule optimizations and indirectly increases the code base. For example, (Ford, 2004) theoretically showed that `oneOrMore`, `option`, and `and-predicate` operators are "syntactic sugar", i.e. the combination of other operators that can substitute them. Staging and compilation techniques (Rompf & Odersky, 2010) might evaporate intermediate data creation. But they require a 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 the facilities (like debugging and tracing). Such code generation potentially blow up the method code size (limited by JVM), complicate the code base support, and lessen *parboiled2* versions back compatibility.

Another problem arises from the fact that the `rule` macro depends on the context that it does not 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. This 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 the `ValueStack` arise from the inefficiency of the combinator approaches – they produce too many intermediate data structures. The first negative thing is *parboiled2* shifts side-effect result composition too much on the developers' shoulders. Hence, again we are constrained with the API and backward compatibility. Another drawback is the `ValueStack` type-based verification, which is good for the user when the type check passes. If a user makes a mistake somewhere in typing (i.e., missed an argument type in lambda for action operation), the Scala compiler fires tens of lines of machine-generated

typing errors that are really hard to interpret by a human. (Jonnalagedda *et al.*, 2014) showed how to eliminate intermediate data structures automatically.

The 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 is that a new version of Scala Macros should be sufficient to overcome all the obstacles (Liu & Burmako, 2017).

10 Acknowledgments

The core development is supported by Google Summer of Code 2013 grant. The development process leading and significant contributions was made by Mathias Doenitz.

The work on the paper is supported by the National Science Foundation (NSF DBI-1356347).

References

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

Béguet, Eric, & Jonnalagedda, Manohar. (2014). Accelerating parser combinators with macros. *Pages 7–17 of: Proceedings of the Fifth Annual Scala Workshop*. ACM.

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

Ford, Bryan. (2004). Parsing expression grammars: A recognition-based syntactic foundation. *Sigplan not.*, **39**(1), 111–122.

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

Haoyi, Li. (2014). *Fastparse*. https://github.com/lihaoyi/fastparse.

Jonnalagedda, Manohar, Coppey, Thierry, Stucki, Sandro, Rompf, Tiark, & Odersky, Martin. (2014). Staged parser combinators for efficient data processing. *Sigplan not.*, **49**(10), 637–653.

Liu, Fengyun, & Burmako, Eugene. (2017). *Two approaches to portable macros*. Tech. rept.

Medeiros, Sérgio, Mascarenhas, Fabio, & Ierusalimschy, Roberto. (2014). Left recursion in parsing expression grammars. *Sci. comput. program.*, **96**(P2), 177–190.

Moors, Adriaan, Piessens, Frank, & Odersky, Martin. (2008). *Parser combinators in scala*. Tech. rept.

Mozzherin, Dmitry Y., Myltsev, Alexander A., & Patterson, David J. (2017). "gnparser": a powerful parser for scientific names based on parsing expression grammar. *Bmc bioinformatics*, **18**(1), 279.

Odersky, Martin, Spoon, Lex, & Venners, Bill. (2016). *Programming in scala: Updated for scala 2.12*. 3rd edn. USA: Artima Incorporation.

Redziejowski, Roman R. (2007). Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Fundam. inf.*, **79**(3-4), 513–524.

Rompf, Tiark, & Odersky, Martin. (2010). Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Pages 127–136 of: Acm Sigplan Notices*, vol. 46. ACM.

Wadler, Philip. (1995). Monads for functional programming. *Pages 24–52 of: International School on Advanced Functional Programming*. Springer.