Accelerating Parser Combinators with Macros

Eric Béguet Manohar Jonnalagedda

EPFL, Switzerland
{first.last}@epfl.ch

ABSTRACT

Parser combinators provide an elegant way of writing parsers: parser implementations closely follow the structure of the underlying grammar, while accommodating interleaved host language code for data processing. However, the host language features used for composition introduce substantial overhead, which leads to poor performance.

In this paper, we present a technique to systematically eliminate this overhead. We use Scala macros to analyse the grammar specification at compile-time and remove composition, leaving behind an efficient top-down, recursive-descent parser.

We compare our macro-based approach to a staging-based approach using the LMS framework, and provide an experience report in which we discuss the advantages and drawbacks of both methods. Our library outperforms Scala's standard parser combinators on a set of benchmarks by an order of magnitude, and is 2x faster than code generated by LMS.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*Parsing, Optimization*

General Terms

Languages, Performance

Keywords

Parser combinators, macros, Scala, optimization

1. INTRODUCTION

Parser combinators [27, 14, 15] are an intuitive way to write parsers. In functional languages such as Scala, they are implemented as higher-order functions that map an input into a structured representation of this input. Parsers written in such a way closely mirror their formal grammar description. Moreover, as they are embedded in a host language, they are modular, composable, and readily executable.

The main reason why parser combinators are not widely adopted is that they suffer from extremely poor performance (see Section 4). This is because the abstractions that allow for expressivity have a high running time overhead. Despite its declarative appearance, a grammar description is interleaved with input handling, and so while input is processed, parts of the grammar description are rebuilt over and over again.

Let us note, however, that parser composition is mostly static. Before running a parser on an input, we have full knowledge about the structure of the parser itself. If we are able to dissociate composition from input processing at compile time, we can eliminate the overhead of the former away, leaving behind an efficient parser that will simply run on the input. In other words, we should be able to turn a parser combinator into a parser generator at compile-time. This gives us the best of both worlds: the composability, modularity and expressiveness of a host language coupled with the performance of a generator approach.

In the Scala ecosystem, there are two main approaches to compile-time optimizations and rewrites. The traditional way is to implement a compiler plugin. The main disadvantage of such an approach is that it exposes the full internals of the Scala compiler. A developer needs to know a lot about Scala's compiler trees, which are more general (less domain-specific) than the production rules of a grammar description. This also makes other domain-specific optimizations (for example grammar rewrites for left recursion) more cumbersome to implement.

An alternate approach is to use metaprogramming techniques, such as multi-stage programming (staging) [26, 19] or macros [5]. Such techniques allow us to operate with a much more high-level description of a parser program, rendering the implementation a lot easier and extensible.

In this paper, we present an implementation of parser combinators that uses Scala macros to eliminate the overhead of composition at compile-time. From a user's point of view, a parser can be written as easily as when using Scala's standard parser combinator library. In particular, we make the following contributions:

- We use Scala macros to separate static composition of parser combinators from the dynamic input processing at compile time. This is done in a two-phase transform. A parser is written inside a `FastParser` context, a macro, that closes the parser world. During the first phase,

Scala'14 July 28–29, 2014, Uppsala, Sweden

http://dx.doi.org/10.1145/2637647.2637653.

we analyse the parser structure and inline production rules so that the parser is simplified to contain a chain of elementary combinators (Section 3.1).
- During the inlining step, we also handle recursive parsers and parser calls across different `FastParser` contexts. In the second phase, we expand the definitions of elementary combinators, using quasiquotes [23] (Section 3.2).
- This transformation is not trivial for higher-order combinators like `flatMap`. We analyse the body of the function passed to the combinator and expand it as necessary (Section 3.4). We also handle production rules that take parameters (Section 3.5).
- We evaluate our macro-based library on a HTTP parser, a CSV parser and a JSON parser. We compare our performance against 3 implementations: the standard library, a staging based implementation in the LMS framework [11] and Parboiled2 [6], a parser combinator library that also uses macros (Section 4).
- We provide an extended experience report on improving parser combinator performance with Scala macros, and using staging and the LMS framework [19, 20]. We discuss advantages and drawbacks of both methods (Section 5).

Section 6 discusses related work, and we conclude in Section 7. Before delving into the macro-based implementation, we give some background on parser combinators and Scala macros.

2. BACKGROUND

Before describing our macro-based implementation, we quickly introduce parser combinators and macros.

2.1 Parser Combinators

Parser combinators are functions from an input to a parse result. A parse result is either a success or a failure. Figure 1 shows an implementation for parser combinators in Scala. We create a `Parsers` trait that acts as the context in which parser combinators are implemented. Note that we abstract over the element type (`Elem`), and the input type (`Input`). `Input` is itself a type alias for a `Reader`, which is an interface for accessing tokens in a sequence or a stream. A simple `Reader` can be defined over an array of characters, or a string. The basic element type is a `Char` in this case.

Some of the important combinators include `flatMap`, which binds a parse result to a parser. This is the monadic bind operator for parser combinators; it allows us to make decisions based on the parse result of a previous parser. The alternation combinator '|' parses the right-hand side parser only if the left side parser fails. The `map` combinator transforms the value of a parse result. Finally, the '~' combinator does sequencing, where we are interested in the results of the left and the right hand side. We also define a helper function for creating parsers more easily.

In the rest of this paper, in addition to the combinators present in Figure 1, we will use the following combinators:
- `lhs ~> rhs` succeeds if both `lhs` and `rhs` succeed, but we are only interested in the parse result of `rhs`
- `lhs <~ rhs` succeeds if both `lhs` and `rhs` succeed, but we are only interested in the parse result of `lhs`
- `rep(p)` repeatedly uses `p` to parse the input until `p` fails. The result is a list of the consecutive results of `p`.
- `repN(n,p)` uses `p` exactly `n` times to parse the input. The result is a list of the `n` consecutive results of `p`.

```
trait Parsers {
  type Elem
  type Input = Reader[Elem]
  abstract class ParseResult[T]

  case class Success(res: T, next: Input)
        extends ParseResult[T] {
    def isEmpty = false
  }
  case class Failure(next: Input) extends ParseResult[T] {
    def isEmpty = true
  }

  abstract class Parser[T]
        extends (Input => ParseResult[T]) {
    def | (that: Parser[T]) = Parser[T] { pos =>
      val tmp = this(pos)
      if(tmp.isEmpty) that(pos)
      else tmp
    }

    def flatMap[U](f: T => Parser[U]) = Parser[U] { pos =>
      val tmp = this(pos)
      if(tmp.isEmpty) Failure(pos)
      else f(tmp.res)(tmp.next)
    }

    def map[U](f: T => U) = Parser[U] { pos =>
      val tmp = this(pos)
      if(tmp.isEmpty) tmp
      else Success(f(tmp.res), tmp.next)
    }

    def ~[U](that: Parser[U]) : Parser[(T,U)] =
      for(
       r1 <- this;
       r2 <- that
      ) yield (r1, r2)
  }

  def Parser[T](f: Input => ParseResult[T]) =
        new Parser[T] {
          def apply(pos: Input) = f(pos)
        }
}

abstract class Reader[T] {
  def first: T
  def next: Reader[T]
  def atEnd: Boolean
}
```

Figure 1: An implementation of parser combinators

- `repsep(p,q)` repeatedly uses `p` interleaved with `q` to parse the input, until `p` fails. The result is a list of the results of `p`. For example, `repsep(term, ",")` parses a comma-separated list of terms, yielding a list of these terms.

We can now define an elementary parser for characters, which accepts a character based on a predicate.

```
trait CharParsers extends Parsers {
  def acceptIf(p: Char => Boolean) = Parser[Char] {
        in =>
    if (!in.atEnd && p(in.first))
                Success(in.first, in.rest)
    else Failure(in)
  }

  def accept(c:Char) = acceptIf(x => x == c)
}
```

We give a short example for parsing a sequence of string literals separated by commas, and enclosed in square brackets. An example input would be `["hello", "world"]` :

```
object StringParser extends Parsers with CharParsers
{
  def repToString(p: Parser[Char]): Parser[String]
    = rep(p) map { xs => xs.mkString }

  def stringLit: Parser[String] = (
  '"' ~> repToString(acceptIf(_ != '"'))
   <~ '"') map( x => '\"' + x + '\"')

  def stringList: Parser[List[String]] =
    '[' ~> repsep(double, ",") <~ ']'
}
```

We first create an instance of the `Parsers` trait and mix in `CharParsers` to be able to use combinators that parse characters, specifically. Parsing a string literal amounts to parsing the opening quote, repeatedly parsing non-quote characters, and finally the closing quote. To parse a sequence of literals, we make use of the `repsep` combinator.

Abstraction Penalties.

The functional implementation shown in Figure 1 leads to poor performance, because:

- The execution of a parser goes through many indirections. First and foremost, every parser is a function. Functions being objects in Scala, function application amounts to method calls. A composite parser, composed of many smaller parsers, when applied to an input, not only constructs a new parser at every application, but also chains many method calls, which incurs a huge cost due to method dispatch. The use of higher-order functions incurs this cost as well.
- We construct many intermediate parse results during the execution of a parser: for every combinator, we box the parse, plus the position, into a `ParseResult` object, before manipulating its fields. Inlining by itself will not rid us of these intermediate data structures: for one, recursive parsers are very common, and the control flow of a parser has many split and join points in the form of conditionals.

In summary, it is precisely the language abstraction mechanisms that enable us to *compose* combinators that are hindering our performance.

2.2 Scala Macros

Scala Macros [5] bring meta-programming capabilities to the Scala language. Just like in other languages like Lisp and Racket, a macro is a metaprogram that is executed at compile time. When a macro is invoked, it exposes internals of the Scala compiler API, thereby allowing us to manipulate and transform expression trees. There are many different flavors of Scala macros, which enable manipulation of types as well as terms. In this paper, we are concerned only with the most basic form, *def macros*. Here is an example of a def macro:

```
import MyParsers._

val jsonParser = FastParser {
 def value: Parser[Any] = obj | arr | stringLit |
       decimalNumber | "null" | "true" | "false"
 def obj: Parser[Any] = "{" ~> repsep(member, ",") <~ "}"
 def arr: Parser[Any] = "[" ~> repsep(value, ",") <~ "]"
 def member: Parser[Any] = stringLit ~ (":" ~> value)
}
```

Figure 2: A simple JSON parser

```
def mul(a: Int, b: Int) = macro mul_impl
...
def mul_impl(c: Context)(a: c.Tree, b: c.Tree) = b match {
  case q"2" => q"$a << 1"
  case _ => ...
}
```

The `mul` function defers its implementation to the `mul_impl` function, which is a macro, as the preceding keyword indicates. This function gets a context as a parameter, in addition to the two operands, which are now in AST form. We then pattern match the tree using quasiquotes (string literals of form `q""`), and provide an optimized implementation using quasiquotes. Quasiquotes simplify expression matching and rewriting by allowing us to use Scala syntax.

There are two variants of def macros [3]:

- *Blackbox macros* act like any ordinary Scala function. They are well typed, in that the macro has to respect the signature of the declaration. For a user, there is no difference between calling an ordinary function and a blackbox macro.
- *Whitebox macros* are more powerful. They differ from blackbox macros in that they are not bound to a specific type signature. They have a return type but they can refine it. This allows us to generate new types at compile time. For example, they can be used to implement Type Providers [5, §4.2].

Current implementation restrictions in Scala Macros force us to separate macro implementations and their usage into different compilation units. Hence, in the rest of the paper, we will refer to these two worlds as the *macro world* and the *external world*, respectively.

3. MACRO-BASED COMBINATORS

We now describe the implementation of our macro-based parser combinator library. Let us first look at an example use case of this library. Figure 2 shows an implementation of a JSON parser. We import functionality from a `MyParser` object, which gives us access to the combinators, as well as a `FastParser` scope. We then declare our parser inside a `FastParser` scope. This parser looks very similar to a standard parser combinator implementation [16, Chapter 31]. A JSON object is either:

- a primitive `value`, such as a decimal, string literal, a

```
//the interface
object MyParsers extends BaseParsers[Char, String]
  with TokenParsers with RepParsers ... {
  def FastParser(rules: => Unit): FinalFastParser =
    macro MyParsersImpl.FastParser
}

//BaseParsers contains basic combinators
trait BaseParsers[Elem, Input] {
 ...
 implicit class BaseParserHelpers[T](p1: Parser[T]) {
  @compileTimeOnly("can't be used outside FastParser")
  def ~[U](p2: Parser[U]): Parser[(T, U)] =
    throw new NotImplementedError
 }
}

//interface for token parsers
trait TokenParsers { ... }

//the implementation
class MyParsersImpl(val c: Context) extends BaseParsersImpl
  with TokenParsersImpl with RepParsersImpl
  with RulesTransformer with RulesInliner
  with FlatMapImpl with RuleCombiner
  with StringInput with IgnoreParseError {

  def FastParser(rules: c.Tree): FinalFastParser = ...
}

trait ParserImplBase {
  def expand(tree: c.Tree, rs: ResultsStruct ): c.Tree = ...
}
trait BaseParsersImpl extends ParserImplBase { ... }
trait TokenParsersImpl extends ParserImplBase { ... }
...
```

Figure 3: Interface and Implementation

boolean or the null value.
- or an array of values (the `arr` function).
- or an associative table of key-value pairs (the `obj` function).

We can then call the `value` production rule (hence referred to as rule) as follows:

```
val cnt = "{\"firstName\": \"John\"," +
      "\"age\": 25}"
jsonParser.value(cnt) match {
  case Success(result) =>
    println("success :" + result)
  case Failure(error) =>
    println("failure : " + error)
}
```

The separation between external and macro worlds mentioned in the previous section drives our architecture. We distinguish between what a user of the library sees, the *interface*, and the macro world where combinators are optimized, the *implementation*. Figure 3 shows this separation. The `MyParsers` object is the entry point to the interface. We mix in, among many traits, the `BaseParsers` and the `TokenParsers`, as we want to work with input strings, where single elements are `Char`s. This is reminiscent of the `StringParser` object in the previous section.

Each interface trait we mix in provides access to various combinators. The `BaseParsers` trait, for instance, defines the sequence combinator '~'. Note that this throws a `NotImplementedError`: it is just a dummy declaration. Such declarations are present solely to allow a user to compose parsers in a type-safe manner. The `@compileTimeOnly` annotation ensures that a user will get an error message at compile time if this combinator is not used in the `FastParser` context, and that the error will never be thrown.

This `FastParser` scope is where all the magic happens. It is a whitebox macro that takes a set of rules, transforms and optimizes them, and finally returns an object which is a subtype of `FinalFastParser`. This object contains optimized implementations for every rule defined in the `FastParser` scope. Using a whitebox macro allows us to refine the type of this object, so that its rules can be called from outside. Had we used blackbox macros instead, calling the `value` rule from `jsonParser` in the above example would have resulted in a compile-time error.

In the implementation layer, the `MyParsersImpl` class also mixes in functionality for transforming parsers and expanding combinators:

- traits `RulesTransformer` and `RulesInliner` contain implementations for preprocessing rules (Section 3.1).
- traits `BaseParsersImpl`, `TokenParsersImpl`, `RepParsersImpl` and `FlatMapImpl` contain rule expansion functionality (Section 3.2). They extend the `ParserImplBase` trait, which defines an `expand` function.
- the `RuleCombiner` trait combines rules defined in the `FastParser` macro into a final object (Section 3.3).
- the `StringInput` trait indicates that we will work with inputs in the form of strings. For discussion of other forms of input, we refer the reader to Section 3.6.
- the `IgnoreParseError` trait shows that we ignore any form of error reporting. Our implementation also contains a basic modular error reporting facility which can be turned on by mixing in a `DefaultParseError` trait.

In short, the macro-based library consists of an interface, which users declare their parsers against, and an implementation corresponding to the interface. The main restriction is that parsers must be declared inside a `FastParser` context. We now delve into how we optimize the rules inside the `FastParser` macro.

3.1 Rule Transformation

We have seen that the above interface contains dummy declarations of combinators. Intuitively, we want to implement these combinators in the macro world, using quasiquotes. A simple, local replacement of combinators by more efficient code is not sufficient, however. Consider `parser1` in Figure 4. If, for `rule2`, we simply expand the sequence combinator, we are still left with a call to `rule1`. While running the parser, this will still result in performance overhead due to a function call. It is preferable for the body of `rule1` to be expanded here.

In order to permit a more global optimization for combinators, before the macro expansion step (see Section 3.2), we first perform a preprocessing step. This step consists of walking through each rule in a depth-first manner. Whenever we encounter a rule on the path, we inline its right-hand side. After this phase, `rule2` is rewritten as

```
def rule2 = 'd' ~ ('a' ~ 'b')
```

Of course, we need to be careful with recursive parsers. We use the classic technique of tracking rules that we have seen so far. The decision to inline a rule is then based on whether we have seen it before (recursive call) or not. Mutually recursive parsers are handled in the same way, as we perform

```
val parser1 = FastParser {
   def rule1 = 'a' ~ 'b'
   def rule2 = 'd' ~ rule1
   def rule3 = 'y' ~ rule4
   def rule4 = rule3 | 'x'
}

val parser2 = FastParser {
    def rule1 = 'c' ~ parser1.rule2
}
```

Figure 4: Calling external rules

this preprocessing step for every rule. Therefore, `rule3` and `rule4` in figure 4, being mutually recursive, are left untouched.

External Calls.

A rule could call a production defined in another `FastParser` scope. In Figure 4, `rule1` in `parser2` calls `rule2` defined in `parser1` because we prefix `rule2` by `parser1`. This functionality is very useful if we want to reuse optimized parsers different in different libraries: `parser1` could be defined in a completely different package.

During, the preprocessing phase, we take such external calls into account. When we encounter code with the pattern `parser.rule(args)` we do the following:

- We check whether `parser` is a subtype of `FinalFastParser`. Recall that the `FastParser` macro will expand `parser1` into an instance of this class.
- We then check that `rule` is defined on this object, and whether it has been given the right arguments. Moreover we obtain its transformed AST which is contained in a `@saveAST` annotation (see section 3.3).
- When both conditions above hold, we can inline the body of `rule` as per the conditions described above. In the running example, `parser2.rule` will get transformed into

  ```
  def rule1 = 'c' ~ ('d' ~ ('a' ~ 'b'))
  ```

  When replacing a call to an external rule by its AST, one has to be careful to also prefix each rule call properly so that the correct rule is called: `parser1.rule1` is different from `parser2.rule1`. So we rewrite the right-hand side `'d' ~ rule1` as `'d' ~ parser1.rule1` first.

Note that `parser1` has to be expanded *before* `parser2`. This is because we need access to a *real* type in order to call the rules, otherwise `parser1` would only see an object of type `FinalFastParser` which does not contain any methods. `FastParser` being a whitebox macro, the real type is revealed only after macro expansion. Thus the Scala compiler would not even let us compile code where `parser1` called a rule defined in `parser2`. This is also the reason why mutually recursive calls between parsers in different `FastParser` scopes are disallowed.

3.2 Rule Rewriting

Now that the preprocessing and rule inlining step is done, we can expand the implementation of primitive combinators. The meat of the rewriting is done by the `expand` function given in Figure 3. This function takes as arguments the tree of the code to be expanded, and a `ResultsStruct`, which handles parse results (see below for more details). Recall that we want to minimize variable creation for each parse result, and that a parse result also contains a flag indicating success or failure

```
trait BaseParsersImpl extends ParserImplBase {

  override def expand(tree: c.Tree, rs: ResultsStruct):
  c.Tree = tree match {
    case q"$lhs ~[$_] $rhs" => q"""
      ${expand(lhs, rs)}
      if (success) {
        ${expand(rhs, rs)}
      }
    """

    case q"$lhs | [$_] $rhs" => ...
    ...
  }
    ...
}
```

Figure 5: Implementation of `expand` for the '~' combinator

```
var success = false
var result1 = ' '
var result2 = ' '
...
if (inputpos < inputsize && input(inputpos) == 'a'){
 result1 = 'a'
 inputpos += 1
 success = true
}
else {
 success = false
 error = "expected 'a' at " + inputpos
}
if (success){
 if (inputpos < inputsize && input(inputpos) == 'b'){
  result2 = 'b'
  inputpos += 1
  success = true
 }
 else {
  success = false
  error = "expected 'b' at " + inputpos
 }
}
```

Figure 6: Expanded code for `'a' ~ 'b'`

of a parse. We manually separate both concerns here. The `ResultsStruct` contains parse result information. For indicating success or failure we generate a global `success` variable.

During the expansion of combinator implementations, the `success` variable needs to be set to `true` or `false` based on the success of the current parser. Figure 5 shows the implementation of `expand` for the sequence combinator. Note that we use quasiquotes to match this combinator. We recursively expand the left-hand side `$lhs`, which will produce a result. If the result is a success, we match the recursively expanded right-hand side `$rhs`. If the result is an error, the parser stops, and propagates the error upstream. For a parser that parses the letter 'a' followed by the letter 'b' (`'a' ~ 'b'`), we get the expanded code as shown in Figure 6. We assume a `String` input type, thus we also generate variables needed to handle this type of input:

- `inputsize` is the length of the input.
- `input` is the input itself, here of type `String`.
- `inputpos` is the current position we are at in the input.

Naturally, it is possible for a user to define his own spe-

```
case q"$a map[$t] $f" =>
  r = new ResultsStruct
  q"""
  ${expand(a, results_tmp)}
  if (success)
   ${rs.assignNew(q"$f(${r.combine})", $t)}
  """
```

Figure 7: Implementation of `expand` for the `map` combinator, involving `ResultsStruct`

cific expansions, if he desires to add some specific optimized version of a combinator. All that is required is to extend the `ParserImplBase` trait and override the `expand` function.

Managing Parse Results.

As seen in Figure 6 we need to store each temporary result in a fresh variable. These results must be tracked during macro expansion, lest they are accessed further in the parsing process. With the sequence combinator, for instance, we generate two result variables for the left and right hand sides. But we don't need to construct the resulting tuple (recall that '~' a `Parser[(T,U)]`) until it is actually used. We track these dependencies during macro expansion in an instance of the `ResultsStruct` class. This class contains methods to create, combine, assign and track variables for parse results. As the sequence combinator does not explicitly require any variable generation (this is done in the recursive expansion of the left and right hand sides), we show an example use of `ResultsStruct` for the expansion of the `map` combinator in Figure 7. We need to create a new `ResultsStruct` before expanding the left hand side `a`. This `ResultsStruct` will collect results produced by `a`. If the parse is successful (the `success` variable will have been set to `true`), we first combine the results tracked so far, using the `combine` method. We then assign this combined result to a new result variable (`assignNew`). If `a` was a sequencing of other combinators, the resulting tuple creation would be generated at the point where the `combine` function is called.

3.3 Putting it all together

At this point, we have expanded each rule defined in the `FastParser` scope. In essence, from a parser description at the interface level, we have generated an efficient recursive-descent parser. The final step involves bundling these rules together in an object, so that the code can be called from the external world, as in the JSON parser example given at the beginning of Section 3. As hinted before, we use the whitebox capabilities of Scala macros to generate a subtype of `FinalFastParser`. In this object, we generate a method for each rule at the interface level. For each rule of the form `def rule(args: ...)`, we generate a new method with the same name, adding two extra parameters:

```
def rule(in: Input,args: ..., offset: Int):
  ParseResult[T]
```

The `in` parameter represents the input on which the parser will be applied, and `offset` is the position of the input from where the parsing should begin. The return type is `ParseResult[T]`, where `T` is the return type of the original rule.

We also need to anticipate any external calls to the rules we are generating. Recall that in section 3.1, we use the `@saveAST(tree)` annotation to get the original tree of the rule. This annotation is added during the packing up phase to each expanded rule. The `tree` parameter contains the code of the transformed rule (before expansion). This way when we identify an external call we can obtain its inlined AST. Figure 8 shows the final generated object for the parser defined in Figure 4.

```
new FinalFastParser {
  //original definitions, preserved for composition
  @compileTimeOnly("can't be used outside FastParser")
  def rule1: Parser[(Char, Char)] =
        throw new NotImplementedError
  @compileTimeOnly("can't be used outside FastParser")
  def rule2: Parser[(Char, (Char, Char))] =
        throw new NotImplementedError

  //expanded definitions
  def rule1(input: Array[Char],offset: Int):
    ParseResult[(Char, Char),String] @saveAST('a' ~ 'b') = ...

  def rule2(input: Array[Char],offset: Int):
    ParseResult[(Char, (Char, Char)),String]
        @saveAST('a' ~ rule1) = ...
}
```

Figure 8: The resulting `FinalFastParser` from expanding `parser1` in Figure 4

We now describe how to handle non-trivial combinators, such as `flatMap`, and rules that take extra parameters.

3.4 The `flatMap` Combinator

The `flatMap` combinator, as seen in Section 2, produces a new parser by taking a function from a result to a parser. It allows for context-sensitive parsers, such as a parser that reads an integer n, followed by n characters:

```
FastParsers {
  def rule = number flatMap { n => take(n) }
}
```

Applying `rule` on the input "5abcdefg" will return "abcde". Naturally, we would like to apply transformations to the function passed as argument to the `flatMap` combinator as well. The above example would trigger a `compileTimeOnly` error otherwise.

Recall the signature of `flatMap`:

```
def flatMap[U](f: T => Parser[U]): Parser[U]
```

In our implementation, we restrict `f` to be either an anonymous lambda function, or a partial function. In either case, the last statement of a lambda, or the last statements of each case in a partial function are of type `Parser[U]`. Using this knowledge, we can expand the use of a `flatMap` combinator. When we encounter a pattern of the form `a.flatMap{ params => body; ret }`, we do the following:

- We expand `a`, by recursively calling the `expand` function on it.
- We expand `ret`: if it is a simple lambda function, we expand it. If it is a partial function, we recursively expand the last expression of each case. We create a new function of the form `{ params => body; expand(ret) }`.
- Finally we apply the result of `a` to the resulting function.

3.5 Rules with Parameters

Rules defined in the external world can have parameters, of course. They are simply Scala methods, taking extra

```
FastParser {
  //rules with parameters
  def rule1(x: Int) = repN(x, 'b')
  def rule2 = 'a' ~ rule1(5)

  //mutually recursive rules with parameters
  def rule3(p: Parser[List[Char]], y: Int): Parser[Any]
   = 'a' ~ p ~ rule4(y)
  def rule4(x: Int): Parser[Any]
   = rule3(repN(x, 'c'), x + 1) | 'b'
}
```

Figure 9: Rules with parameters

arguments. Figure 9 shows examples of such rules. Note, however, that when rules with parameters are called, the parameters are *concrete*, or specified. In the example above, `rule2` calls `rule1` with a specific parameter (in this case `5`). Therefore, during the preprocessing phase we forward the concrete arguments to the combinators where they are used. As a result `rule2` is transformed to the following:

```
def rule2 = 'a' ~ repN(5, 'a')
```

Once again, recursive rules make things more complicated, as shown with `rule3` and `rule4`. First of all, we cannot pass primitive combinators like `repN(x, 'c')` as is. Recall that their interface type is `Parser[T]`, and expanded code cannot contain this type (a `@compileTimeError` would be raised).

The solution is to first convert rules taking `Parser` parameters into rules that take values of the equivalent expanded type. The signature of `rule3` now becomes

```
def rule3(in: InputType,
  p: (InputType, Int) => ParseResult[List[Char]],
  y: Int,
  offset: Int = 0): ParseResult[Any]
  = ...
```

For every location where `rule3` is called with a specific combinator, we create an expanded/optimized function for this combinator. For the current example, we need to create a function that contains the expanded equivalent of `repN`:

```
def anonymous$1$(in: InputType, x: Int, offs: Int)
  = expanded code for repN(x,'c')
```

Finally, we call the modified `rule3` function with the generated function:

```
rule3(in,
  (in: InputType, offs: Int) =>
  anonymous$1$(in, x, offs),
  x + 1,
  offs)
```

3.6 Changing the Input Type

It is important to be able to abstract over the type of input we parse over, as standard parser combinators do, with type members `Elem` and `Input`(Section 2). In our library, the support for additional input types is also deferred to the macro world, because we are interested in inlining access to a current, or next element.

For example if we want to enable parsing over arrays of integers, we have to create an input trait which extends the `ParserInput` trait. We then redefine its functions `inputElemType` and the `inputType` methods. Additionally we can also override methods defined in `ParserInput` to modulate the way the input is accessed. Figure 10 shows a stub implementation of an input abstraction over arrays of integers.

```
trait IntArrayInput extends ParseInput {
  import c.universe._

  def inputElemType = typeOf[Int]
  def inputType = typeOf[Array[Int]]

  def initInput(startpos: c.Tree, then: c.Tree) =
    q"""
      var inputpos = $startpos
      val inputsize = input.size
      $then
    """

  def currentInput = q"input(inputpos)"
  ...
}
```

Figure 10: Implementation of `IntArrayInput`

```
class InputWindow[Input](val in: Input,
                         val start: Int,val end: Int){
        override def equals(x: Any) = x match {
          case s: InputWindow[Input] =>
                s.in == in &&
                s.start == start &&
                s.end == end
          case _ => super.equals(x)
        }
}
```

Figure 11: An implementation of `InputWindow`

3.7 Handling Strings

A very common use case for parsers is to parse, and collect, a copy of part of the input. For example, the combinators `take(n: Int)`, `takeWhile(f: Char => Boolean)` or `stringLit` check that a part of the input satisfies a certain property, and return this part of the input. We don't however need the result until we do something with it. Creating a copy of the input is therefore a tremendous, and unnecessary, performance overhead. We can compute this result lazily with what we call an `InputWindow`. An `InputWindow` works exactly like an older version of the `subString` method in Java 1.6. It keeps a reference to the original input as well as the start and end position of the sliced section. An implementation is presented in figure 11.

4. EVALUATION

Our macro-based library is available online at [2]. We evaluated it against Scala's standard parser combinators, a parser combinator library based on LMS [11] Parboiled2, a macro-based PEG parser generator [6].

We used Scalameter [18] for benchmarking. This library handles JVM warmup, and runs tests until stability of performance times. We ran the tests on a laptop with an i7-3610QM core and 8 GB of RAM running on Windows 7 64 bits with Oracle's JVM version 1.7. We use the `-optimize` flag on the Scala compiler. We tested three different parsers:

- A HTTP header parser that parses 100 headers.
- A CSV parser, specialized for two different type of values: booleans, doubles and strings.
- A JSON parser, where we run tests on 4 small JSON files (from 8 to 80kb) and one larger file (898kb)

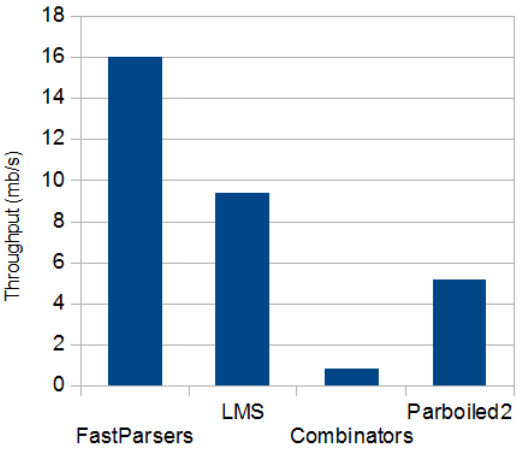

Figure 12: Performance of parsers for JSON

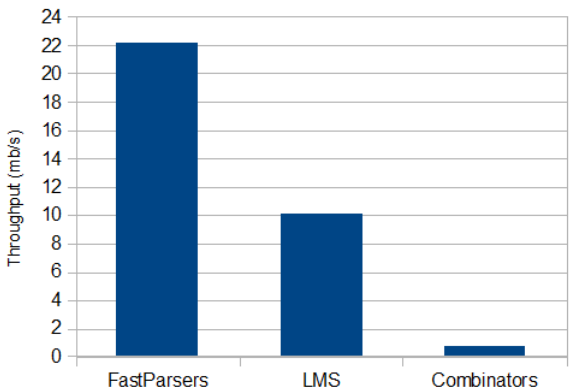

Figure 13: Performance of parsers for HTTP

All files are first loaded into an `Array[Char]` and passed to all the parsers.

The Scala's parser combinators library uses by default a `CharSequence` to read its input. This class makes extensive use of the `substring` method which is in $O(n)$ starting from JVMs version 7. Obviously, using this version of `CharSequence` would completly skew the results as `substring` is called a very large number of times during the parsing. This is why we use a modified version of `CharSequence` which mimics the old implementation of `String.substring` [1].

Our results are presented in Figures 12, 13 and 14. We measure the throughput of parsing in each case. As we can see, our macro-based implementation performs on average 19 times better than Scala's Parser combinators, more than 3 times better than Parboiled2 and even 2 times faster than the LMS version.

While Parboiled2 is also a macro-based solution, we believe our better performance is mainly due to the preprocessing step (Section 3.1). Optimized handling of string literals, as well as removal of error handling, help us as well. Moreover, in Parboiled2, results are stored in an explicit stack, implemented as a `VectorBuilder[Any]`. Accessing values on this stack involves casting, which induces an overhead

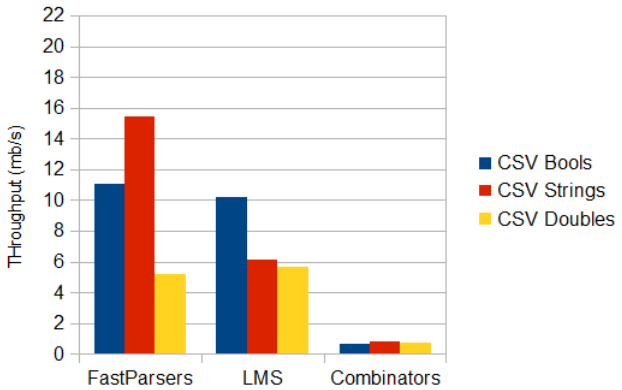

Figure 14: Performance of parsers for CSV

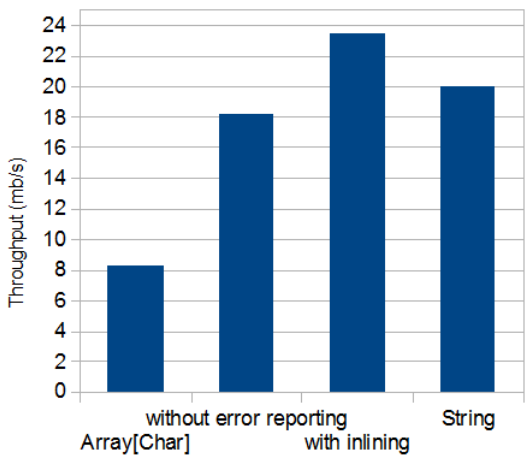

Figure 15: Performance breakdown with respect to various features

not present in our implementation.

LMS is also a code generation framework as well, so technically we should be able to get similar performance with LMS as with our macro-based library. This is indeed the case when parsing CSV files. Because we operate closer to the generated code with macros (see Section 5 for more details) we are able to fine-tune performance more easily.

Figure 15 gives a breakdown of how different optimizations affect the performances of our JSON parser. Disabling error reporting doubles the performance. This is normal as we track less information while parsing. On top of that, we gain an extra 30% with the preprocessing phase. Finally, using `Array[Char]` instead of `String` gives us a gain of 15%.

5. STAGING AND MACROS

As Burmako et al. point out [5, §4.1], the staging approach and the macros approach share many interesting similarities, beyond the fact that they are two approaches to metaprogramming, and that one is a runtime code generation approach while the other does compile-time code generation. We illustrate and compare some of the similarities and differences between both approaches in the context of parser combinators, with respect to DSL development, closed vs. open worlds and code generation guarantees.

5.1 Lightweight Modular Staging

Lightweight Modular Staging (LMS) is a type-directed staging framework for developing embedded DSLs in Scala. Running an LMS program generates an optimized program (runtime code generation), which in turn runs efficiently on dynamic input. Staged and unstaged computations are distinguished by a `Rep[T]` type constructor. An expression of type `T` will be executed at staging time, while an expression of type `Rep[T]` will be generated. As an example, consider the following functions:

```
def add1(a: Int, b: Int) = a + b
def add2(a: Rep[Int], b: Rep[Int]) = a + b
```

The `add1` function gets executed during code generation, producing a constant in the generated code, while `add2` represents a computation that will eventually yield a value of integer type, and is represented as an intermediate node `Add(a,b)`.

5.2 DSL Development

Both the macro and LMS implementations distinguish between interface and implementation. This is very natural: we are essentially creating a deep embedding for a parser combinator DSL, and this involves separating what a user sees from how the library is optimized. The core LMS library provides interfaces, IR nodes and code generation for many common programming constructs. These constructs include conditionals, boolean expressions, arithmetic expressions and array operations, and can be used out of the box.

As a result, staged parser combinators are implemented entirely at the interface level: for instance the sequence combinator's implementation mirrors the standard implementation (Figure 1). At code generation time, the **`for`** expression is converted into a conditional expression structure similar to the quasiquotes implementation.

The important benefit of being able to write implementations at the interface level is that we stay in the same language, or domain. Though quasiquotes enable rewriting trees using Scala syntax, they have the syntactic overhead of quotes and splicing which are avoidable at the library level. Moreover, a user could introduce his own optimized combinators directly at the interface level: this potentially increases his flexibility and productivity.

5.3 Closed vs. Open Worlds

The benefits above are largely due to the fact that LMS operates under the assumption of a fully closed world. Every construct used at the interface level in LMS either uses a corresponding implementation, or defines one, along with appropriate code generation. Such an approach allows for many advanced optimizations, such as inlining of higher-order function calls (for instance the `map` combinator), or even target heterogeneous hardware, through cross-language code generation [4]. This also means, however, that external code cannot be called easily from an LMS DSL. One could not, for instance, use a special `HashMap` unless we first define its interface, implementation, and code generation.

The macros approach allows to cross the open and closed world borders much more easily. As, in addition, quasiquotes target Scala trees, it is much easier to replace unstaged holes in a program with implementations from a Scala library. For example, the LMS parser combinator library uses the concept of staged records [11, §3.4]; there is currently no equivalent in the macro world. We can nonetheless use simple case classes, and forego this specific optimization, in the macro world. In essence, macros allow us to optimize locally, and defer to Scala trees when we do not know how to treat a certain computation. This allows not only for flexibility when developing with macros, but also enables re-use of external tools and libraries.

An interesting solution to bridge the gap between open and closed worlds is to use a front-end such as Yin-Yang [12], where shallow embeddings (libraries) are mapped to their deep embeddings (embedded performance-oriented DSLs) in a macro. Yin-Yang transforms code whenever it can find a deep embedding, and generates a default Scala tree when a deep embedding cannot be found.

5.4 Code Generation Guarantees

From a DSL developer perspective, the LMS framework's IR is extremely powerful. As mentioned before, DSL expressions are not immediately executed, but first converted into an IR. This IR guarantees preservation of evaluation order of expressions. A DSL developer using macros would have to handle such issues himself. For parser combinators such properties are easy to ensure, but it would not be so simple for other DSLs. This is because macros provide a lot more liberty to a developer, and as the adage says, with great power comes great responsibility. It is conceivable, however, to build abstractions on top of macros to help preserve these guarantees. In other words, macros could be considered as the code generation component on top of which a staging framework like LMS could be built.

6. RELATED WORK

Parsing.

Parser combinators and their implementations are popular in functional programming. They were initially proposed by Wadler to illustrate monads, and are of the more general sort, as they produce a list of possible parses [27]. They

have since been incorporated into programming languages as libraries, like Parsec [14] in Haskell and the libary in Scala [14]. These libraries focus on producing a single result. Koopman et al [13] use a continuation-based approach to eliminate intermediate list creation. Such combinator libraries have been extended to handle a bigger class of grammars: packrat parsers support left recursion [8, 28], and there are also combinators that do GLL parsing [21, 25].

On the other hand are parser generators like Yacc [10], Antlr [17] and Happy [9]. While such tools are good in terms of performance, they do not easily support context-sensitivity, which is required in protocol parsing.

Just like the staged parser combinator approach [11], the macro-based approach bridges the gap between both worlds in terms of features for a parser: ease of use, context-sensitivity, composability, specializability and performance.

Metaprogramming and Compiler Technology.

To match performance of lower-level implementations, high-level languages require compiler technology. The staged parser combinator approach makes use of multi-stage programming [26] to evaluate overhead away. Sperber et al. also use partial evaluation for optimising LR parsers which are implemented as a functional-style library [24].

Using macros to generate parsers is not a new idea. A lot of work has been done in the Scheme world: for instance Owens et al. generate lexers and parsers at compile time using DFA and LALR tables [22]. The main difference with our approach is that we use of reified ASTs to enable splitting a grammar across multiple parser blocks.

As stated before *Parboiled2* [6] also use macros to produce an efficient parser which speeds-up parsing considerably compared to Scala's standard parser combinators. Their macro expansion, however, is local to a single rule, while we can optimize a whole parser.

7. CONCLUSION AND FUTURE WORK

We have shown how to use Scala Macros for improving the performance of Parser Combinators. For a library user, the expressivity of using standard combinators is preserved. At compile time, parsers declared inside a macro scope are transformed so as to generate an efficient parser from the declaration. Performance evaluation shows that we outperform the standard implementation, and also beat a staging-based implementation. We also explored the differences between the macro and the staging approaches, from an implementation, and architecture point of view.

In terms of future work we see the following possibilities:

- Macros open up possibilities for better error handling for parser combinators. Since the structure of parsers can be explored at compile time, we can generate more meaningful error messages, and also perform better error recovery.
- Macros can also be used to inject profiling for parser combinators. Because we are interested in profiling after all, a domain-specific profiler would come in handy.
- We can naturally extend the approach to other classes of parsers, such LR, LALR, GLL [21] or Earley [7] parsers.
- In terms of DSL development, it would also be interesting to explore the sweet spot between open and closed world approaches more closely. Building better abstractions around macros could be a way of restricting certain types of unsafe behaviour but allowing more flexibility when using optimized libraries.

Acknowledgements

We would like to thank our colleagues at the programming methods lab (LAMP) at EPFL for many discussions and insightful comments regarding Scala Macros. We thank the anonymous reviewers whose remarks helped us improve the quality of the paper. This research was sponsored by ERC under the DOPPLER grant (587327).

8. REFERENCES

[1] FastCharSequence: Mimics java 1.6 substring in a charsequence. `https://issues.scala-lang.org/browse/SI-7710`.

[2] FastParsers: A macro-based parser combinators libraray. `https://github.com/begeric/FastParsers/tree/experiment`.

[3] Macros: Blackbox vs whitebox. `http://docs.scala-lang.org/overviews/macros/blackbox-whitebox.html`.

[4] K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[5] E. Burmako. Scala Macros: Let Our Powers Combine! In *4th Annual Workshop Scala 2013*, 2013.

[6] M. Doenitz and A. Myltsev. Parboiled2: A macro-based peg parser generator for scala 2.10+. `http://parboiled2.org/`.

[7] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970.

[8] B. Ford and M. F. Kaashoek. Packrat parsing: a practical linear-time algorithm with backtracking, 2002.

[9] A. Gill and S. Marlow. Happy: The parser generator for haskell, 2010.

[10] S. C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1979.

[11] M. Jonnalagedda, T. Coppey, S. Stucki, T. Rompf, and M. Odersky. On Staged Parser Combinators for Efficient Data Processing. Technical report, 2014.

[12] V. Jovanovic, V. Nikolaev, N. D. Pham, V. Ureche, S. Stucki, C. Koch, and M. Odersky. Yin-yang: Transparent deep embedding of dsls, 2013.

[13] P. Koopman and R. Plasmeijer. Efficient combinator parsers. In *In Implementation of Functional Languages, LNCS*, pages 122–138. Springer-Verlag, 1998.

[14] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, 2001.

[15] A. Moors, F. Piessens, and M. Odersky. Parser combinators in scala, 2008.

[16] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[17] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[18] A. Prokopec. Scalameter: Automate your performance testing today. `http://scalameter.github.io/`.

[19] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *GPCE*, pages 127–136, 2010.

[20] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 497–510, New York, NY, USA, 2013. ACM.

[21] E. Scott and A. Johnstone. Gll parse-tree generation. *Sci. Comput. Program.*, 78(10):1828–1844, Oct. 2013.

[22] O. S. Scott Owens, Matthew Flatt and B. McMullan. Lexer and parser generators in scheme.

[23] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for Scala. Technical report, 2013.

[24] M. Sperber and P. Thiemann. The essence of lr parsing. In *In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 146–155. ACM Press, 1995.

[25] D. Spiewak. Generalized parser combinators, 2010.

[26] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[27] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag.

[28] R. Warth, J. R. Douglass, T. Millstein, A. Warth, J. R. Douglass, and T. Millstein. T.: Packrat parsers can support left recursion. In *In: Proc. PEPM, ACM*, pages 103–110, 2008.