Staging Parser Combinators for Efficient Data Processing

Parsing @ SLE, 14 September 2014

Manohar Jonnalagedda
What are they good for?

- **Composable**
  - Each combinator builds a new parser from a previous one
- **Context-sensitive**
  - We can make decisions based on a specific parse result
- **Easy to Write**
  - DSL-style of writing
  - Tight integration with host language
Example: HTTP Response

HTTP/1.1 200 OK
Date: Mon, 23 May 2013 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Etag: "3f80f-1b6-3e1cb03b"
Content-Type: text/html; charset=UTF-8
Content-Length: 129
Connection: close

... payload ...
Example: HTTP Response

HTTP/1.1 200 OK
Date: Mon, 23 May 2013 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Etag: "3f80f-1b6-3e1cb03b"
Content-Type: text/html; charset=UTF-8
Content-Length: 129
Connection: close

... payload ...

Content
Example: HTTP Response

def status = ( "HTTP/" ~ decimalNumber ) ~> wholeNumber ~< (text ~ crlf)

) map (_.toInt)  

Transform parse results on the fly
Example: HTTP Response

def status = ("HTTP/" ~ decimalNumber) ~> \wholeNumber<~ (text ~ crlf)
        ) map (_.toInt)  

Transform parse results on the fly

def header = (headerName ~> ":") flatMap {
    key => (valueParser(key) ~> crlf) map {
        value => (key, value)
    }
}

Make decision based on parse result
Example: HTTP Response

def status = ("HTTP/" ~> decimalNumber) ~> wholeNumber <~ (text ~ crlf) map (_.toInt)

def header = (headerName <~ ":") flatMap {
    key => (valueParser(key) <~ crlf) map {
        value => (key, value)
    }
}

def respWithPayload = response flatMap {
    r => body(r.contentLength)
}
Parser combinators are slow

Topic of this talk.

Standard Parser Combinators

Staged Parser Combinators

Throughput

20x
Parser Combinators are slow

def status: Parser[Int] = ("HTTP/" ~ decimalNumber) ~> wholeNumber <~ (text ~ crlf)

) map (_.toInt)
def header = (headerName <~ ":") flatMap {
    key => (valueParser(key) <~ crlf) map {
        value => (key, value)
    }
}
def respWithPayload = response flatMap {
    r => body(r.contentLength)
}
Parser Combinators are slow

def status: Parser[Int] = ( "HTTP/" ~> decimalNumber ) ~> wholeNumber <<< ( text ~ crlf )

) map (_.toInt)
def header = (headerName <<< ":\n") flatMap {
  key => (valueParser(key) <<< crlf) map {
    value => (key, value)
  }
}
def respWithPayload = response flatMap {
  r => body(r.contentLength)
}
Parser Combinators are slow

- Prohibitive composition overhead
- **But:** composition is mostly static
  - Let us systematically remove it!
Staged Parser Combinators

Composition of Parsers
Staged Parser Combinators

Composition of Parsers

Composition of Code Generators
Staging (LMS)

def add3(a: Int, b: Int, c: Int) = a + b + c

add3(1, 2, 3)

6

‘Classic’ evaluation
Staging (LMS)

Adding Rep types

Expression in the next stage

Executed at staging time
Constant in the next stage

```
def add3(a: Rep[Int], b: Int, c: Int) = a + b + c
```

```
def add3(a: Int, b: Int, c: Int) = a + b + c
```

```
add3(1, 2, 3)
```

‘Classic’ evaluation

6
Staging (LMS)

def add3(a: Rep[Int], b: Int, c: Int) = a + b + c

Expression in the next stage

Adding Rep types

Executed at staging time
Constant in the next stage

Code generation

Evaluation of generated code

‘Classic’ evaluation
User-written code, may contain Rep types → LMS runtime code generation → Generated/optimized code.
class Parser[T] extends (Input => ParseResult[T])

dynamic input/output

class Parser[T] extends (Rep[Input] => Rep[ParseResult[T]])

static function: application == inlining for free
Staging Parser Combinators
Composition of Code Generators

class Parser[T] extends (Input => ParseResult[T])

def ~[U](that: Parser[U])
def map[U](f: T => U): Parser[U]

class Parser[T] extends (Rep[Input] => Rep[ParseResult[T]])

def ~[U](that: Parser[U])
def map[U](f: Rep[T] => Rep[U]): Parser[U]
Staging Parser Combinators
Composition of Code Generators

class Parser[T] extends (Input => ParseResult[T])

def ~[U](that: Parser[U])
def map[U](f: T => U): Parser[U]
def flatMap[U](f: T => Parser[U]): Parser[U]

class Parser[T] extends (Rep[Input] => Rep[ParseResult[T]])

def ~[U](that: Parser[U])
def map[U](f: Rep[T] => Rep[U]): Parser[U]
def flatMap[U](f: Rep[T] => Parser[U]): Parser[U]

static function: application == inlining for free

dynamic input/output

still a code generator

still a code generator
A closer look

```scala
def respWithPayload: Parser[..] =
  response flatMap {
    r => body(r.contentLength)
  }

// code for parsing response
val response = parseHeaders()
val n = response.contentLength
// parsing body
var i = 0
while (i < n) {
  readByte()
  i += 1
}
```

User-written parser → code generation → Generated code
Gotchas

- Recursion
  - explicit recursion combinator (fix-point like)
- Diamond control flow
  - code generation blowup

General solution
- generate staged functions (\texttt{Rep[Input \Rightarrow ParseResult]})
Performance: Parsing JSON

- 20 times faster than Scala’s parser combinators
- 3 times faster than Parboiled2

![Graph showing performance comparison between Combinators, Parboiled2, LMS, and FastParsers.](image-url)
Performance

HTTP Response

CSV

![Bar chart showing performance metrics for different methods (Combinators, LMS, FastParsers) and data types (CSV bools, CSV strings, CSV doubles).]
If you want to know more

- Parser Combinators for Dynamic Programming [OOPSLA ‘14]
  - based on ADP
  - code gen for GPU
- Using Scala Macros [Scala ‘14]
## Desirable Parser Properties

<table>
<thead>
<tr>
<th></th>
<th>Hand-written</th>
<th>Parser Generators</th>
<th>Staged Parser Combinators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composable</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Customizable</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Context-Sensitive</td>
<td>✓</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>Fast</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Easy to write</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
The people

- Eric Béguet
- Thierry Coppey
- Sandro Stucki
- Tiark Rompf
- Martin Odersky
Tack!
Fråga?
Staging all the way down

- Staged structs
  - boxing of temporary results eliminated
- Staged strings
  - substring not computed all the time
Optimizing String handling

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)
    }
}
Key performance impactors

Standard Parser Combinators

Beware!

- String.substring is in linear time (>= Java 1.6).
- Parsers on Strings are inefficient.
- Need to use a FastCharSequence which mimics original behaviour of substring.
Key performance impactors

Standard Parser Combinators

Standard Parser Combinators with FastCharSequence
Key performance impactors

- Standard Parser Combinators
- Standard Parser Combinators with FastCharSequence
- FastParsers with error reporting and without inlining

~7-8x
Key performance impactors

- Standard Parser Combinators
- Standard Parser Combinators with FastCharSequence (~7-8x)
- FastParsers with error reporting and without inlining (~2x)
- FastParsers without error reporting without inlining
**Key performance impactors**

- Standard Parser Combinators
- Standard Parser Combinators with FastCharSequence
- FastParsers with error reporting and without inlining (~7-8x)
- FastParsers without error reporting without inlining (~2x)
- FastParsers without error reporting with inlining (~30%)