Staging Parser Combinators for Efficient Data Processing

Parsing @ SLE, 14 September 2014

Manohar Jonnalagedda

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

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)
Last-Modified: Wed, 08 Jan 2012 23:11:55 GMT
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)
Last-Modified: Wed, 08 Jan 2012 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Content-Type: text/html; charset=UTF-8
Content-Length: 129
Connection: close

... payload ...
```

Status

Headers

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

Transform parse results on the fly

Make decision based on parse result

Example: HTTP Response

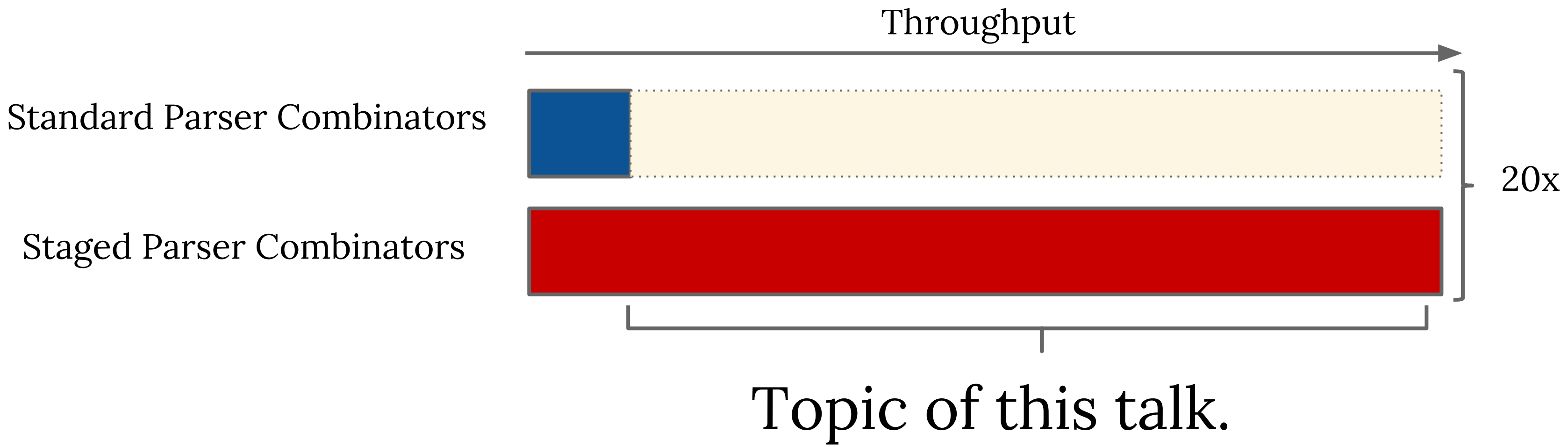
```
def status = ( ("HTTP/" ~ decimalNumber) ~> wholeNumber <~ (text ~ crlf)
) map (_.toInt)
def header = (headerName <~ ":") flatMap {
  key => (valueParser(key) <~ crlf) map {
    value => (key, value)
  }
}
def respWithPayload = response flatMap {
  r => body(r.contentLength)
}
```

Transform parse results on the fly

Make decision based on parse result

Make decision based on parse result

Parser combinators are slow



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)
}
```

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

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)
}
```

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

```
def ~[U](that: Parser[U]) =
new Parser[(T,U)] {
   def apply(i: Input) = ...
  }
```

Parser Combinators are slow

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

Staged Parser Combinators

Composition of Parsers

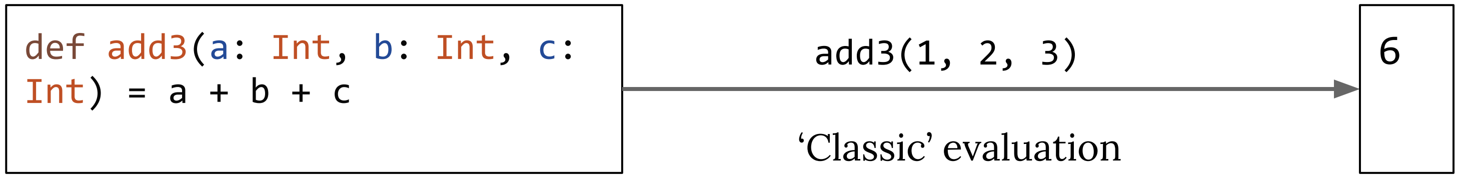
Staged Parser Combinators

~~Composition of Parsers~~

$\downarrow$

Composition of
Code Generators

Staging (LMS)

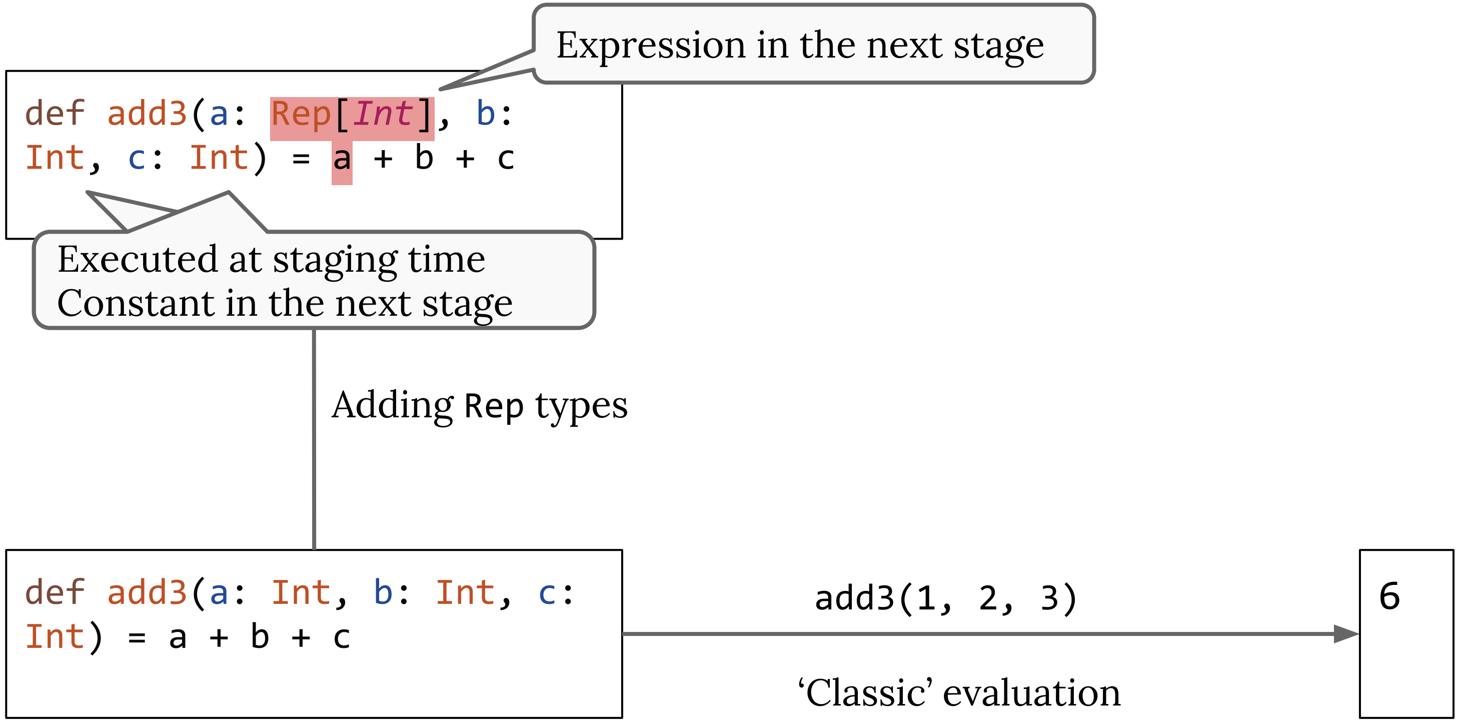


```
def add3(a: Int, b: Int, c:
Int) = a + b + c
```

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

```
6
```

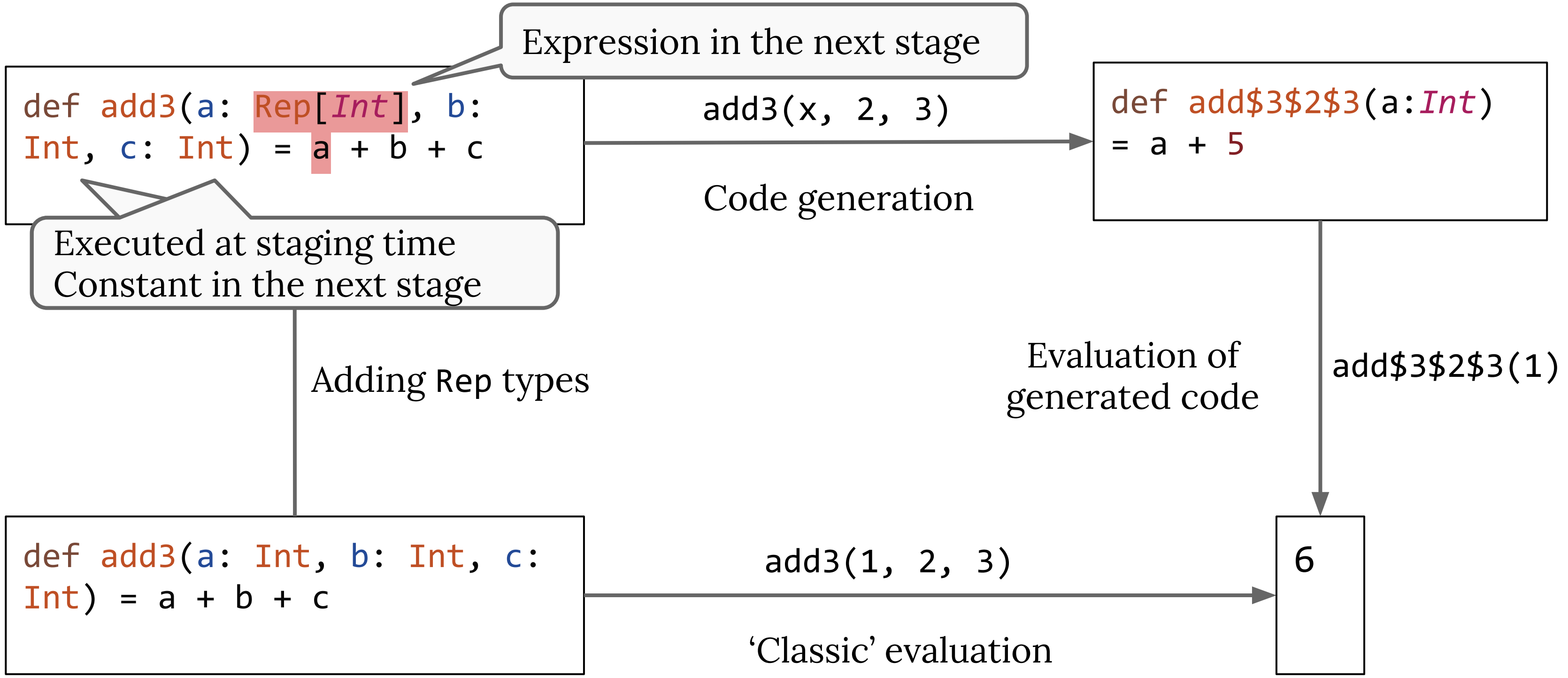
Staging (LMS)



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

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

Staging (LMS)

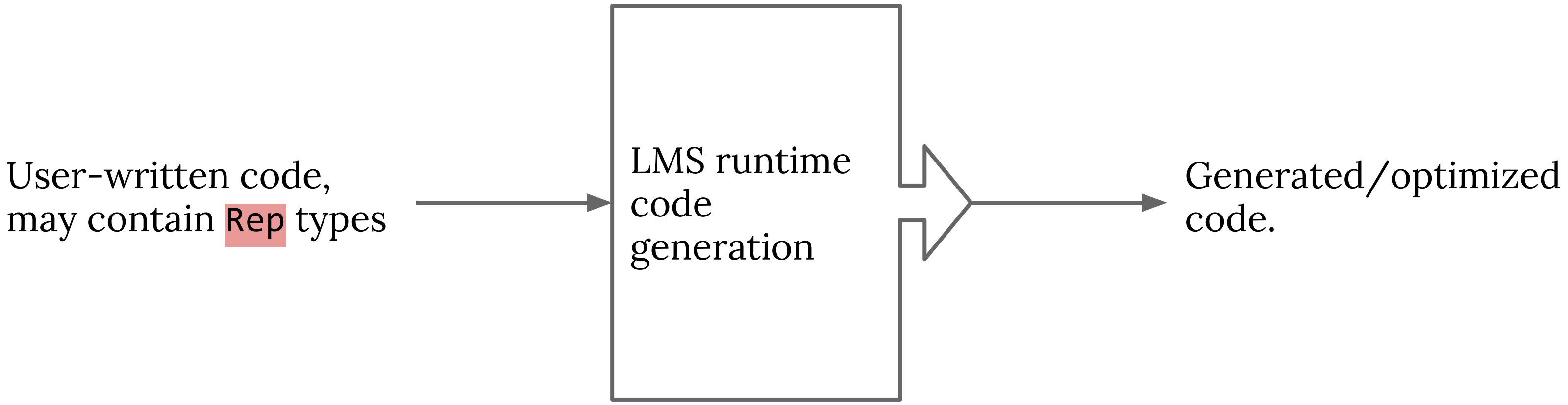


```
def add3(a: Rep[Int], b:
Int, c: Int) = a + b + c
```

```
def add$3$2$3(a:Int)
= a + 5
```

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

LMS


User-written code, may contain `Rep` types
LMS runtime code generation
Generated/optimized code.

Staging Parser Combinators

Composition of Code Generators

```
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])
```

→

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

dynamic input/output

static function: application == inlining for free

```
def ~[U](that: Parser
[U])
```

→

```
def ~[U](that: Parser
[U])
```

still a code generator

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

→

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

Staging Parser Combinators

Composition of Code Generators

dynamic input/output

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

→

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

static function: application == inlining for free

```
def ~[U](that: Parser
[U])
```

→

```
def ~[U](that: Parser
[U])
```

still a code generator

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

→

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

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

→

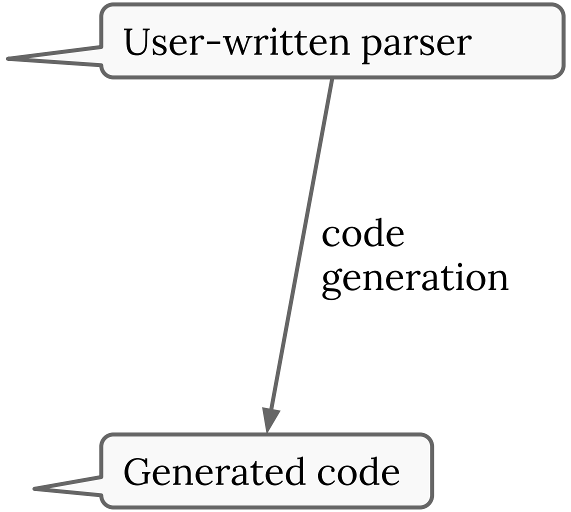
```
def flatMap[U](f: Rep[T] => Parser
[U])
: Parser[U]
```

still a code generator

A closer look

```
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
}
```



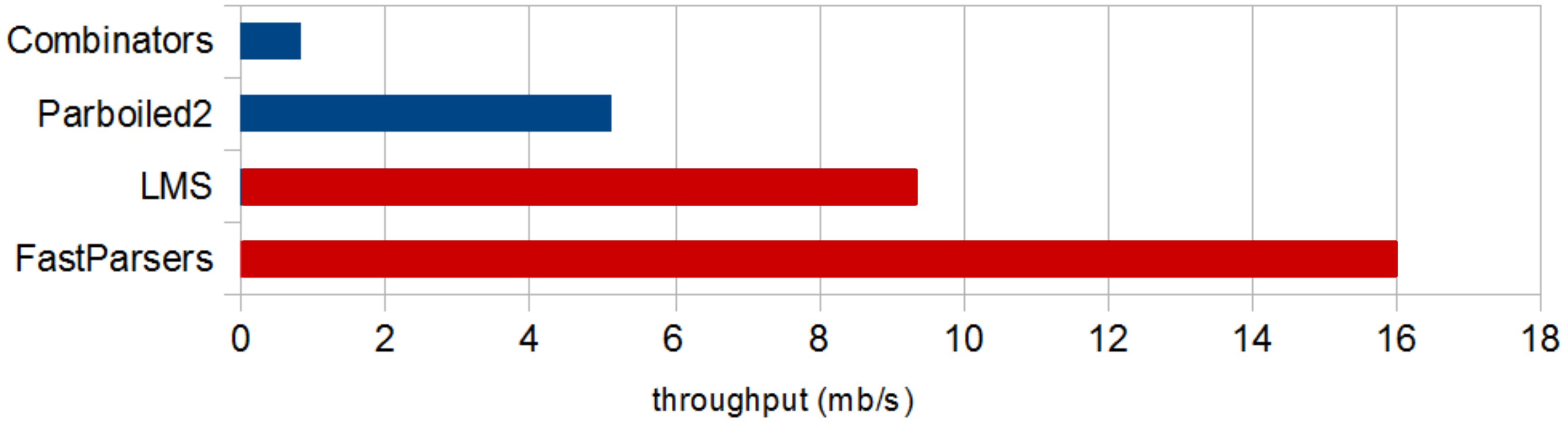
Gotchas

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

General solution

- generate staged functions (`Rep[Input => ParseResult]`)

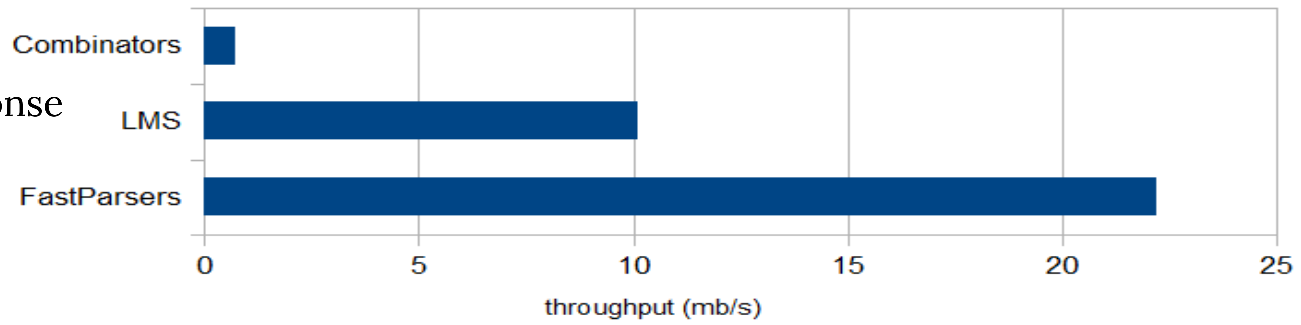
Performance: Parsing JSON



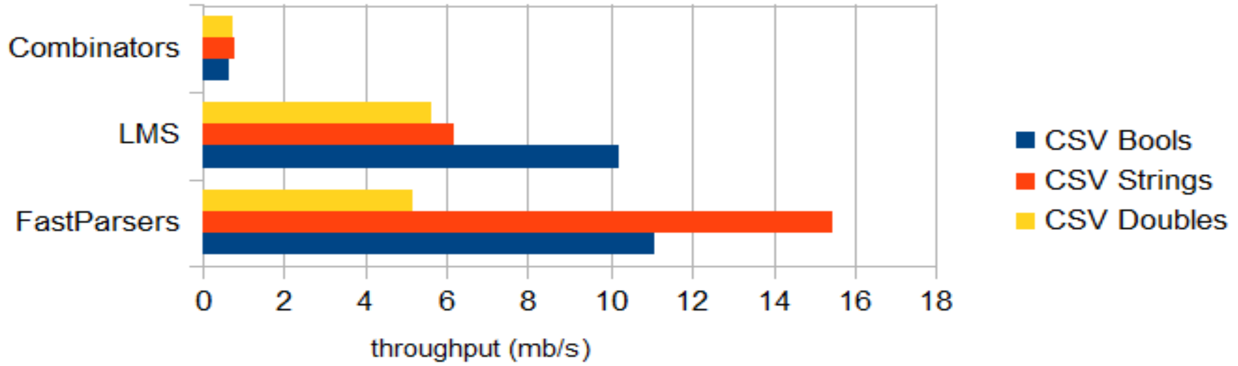
- 20 times faster than Scala's parser combinators
- 3 times faster than Parboiled2

Performance

HTTP Response



CSV



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

	Hand-written	Parser Generators	Staged Parser Combinators
Composable	×	✓	✓
Customizable	×	×	✓
Context-Sensitive	✓	~	✓
Fast	✓	✓	✅
Easy to write	×	✓	✓

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

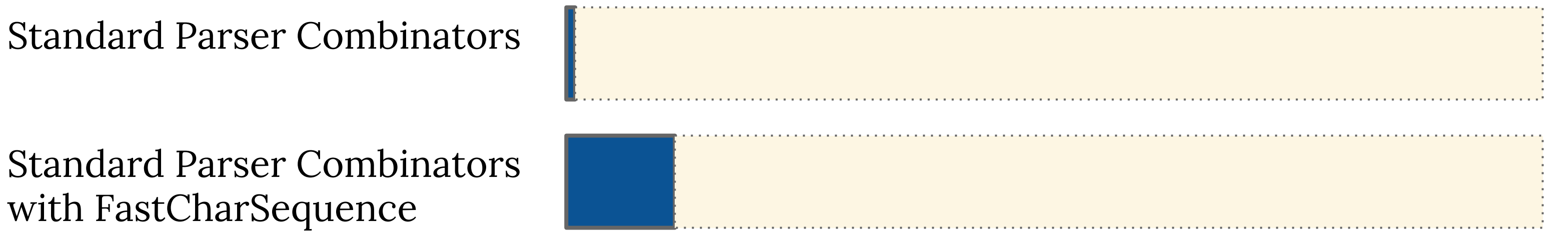
```
class InputWindow[Input](val in: Input, val start: Int, val end: Int){
    override def equals(x: Any) = x match {
        case s : InputWindow[Input] =>
            s.in == in &&
            s.start == start &&
            s.end == end
        case _ => super.equals(x)
    }
}
```

Key performance impactors

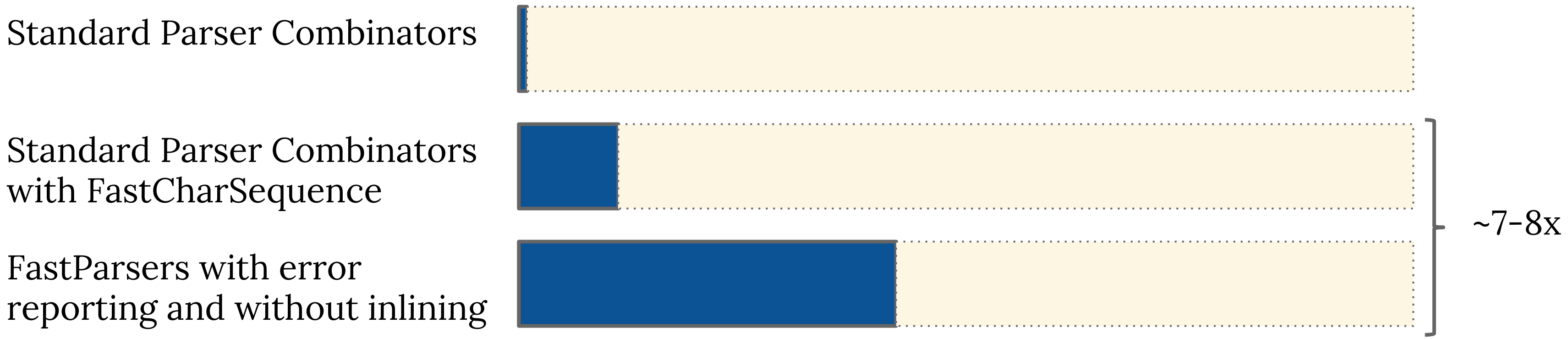
Beware!

- 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.

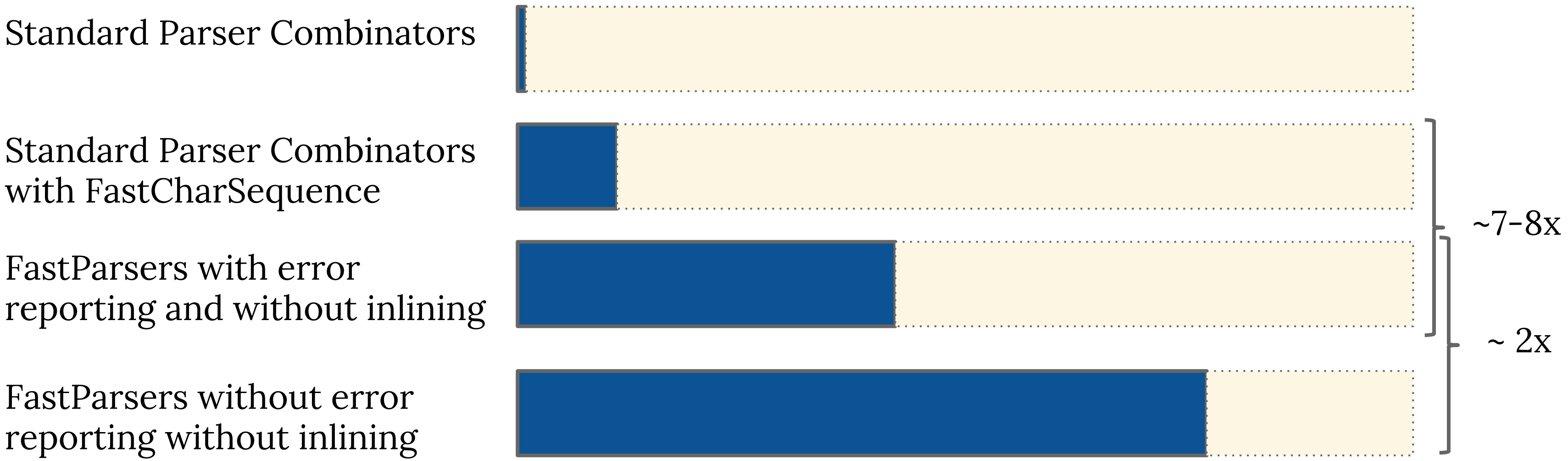
Key performance impactors



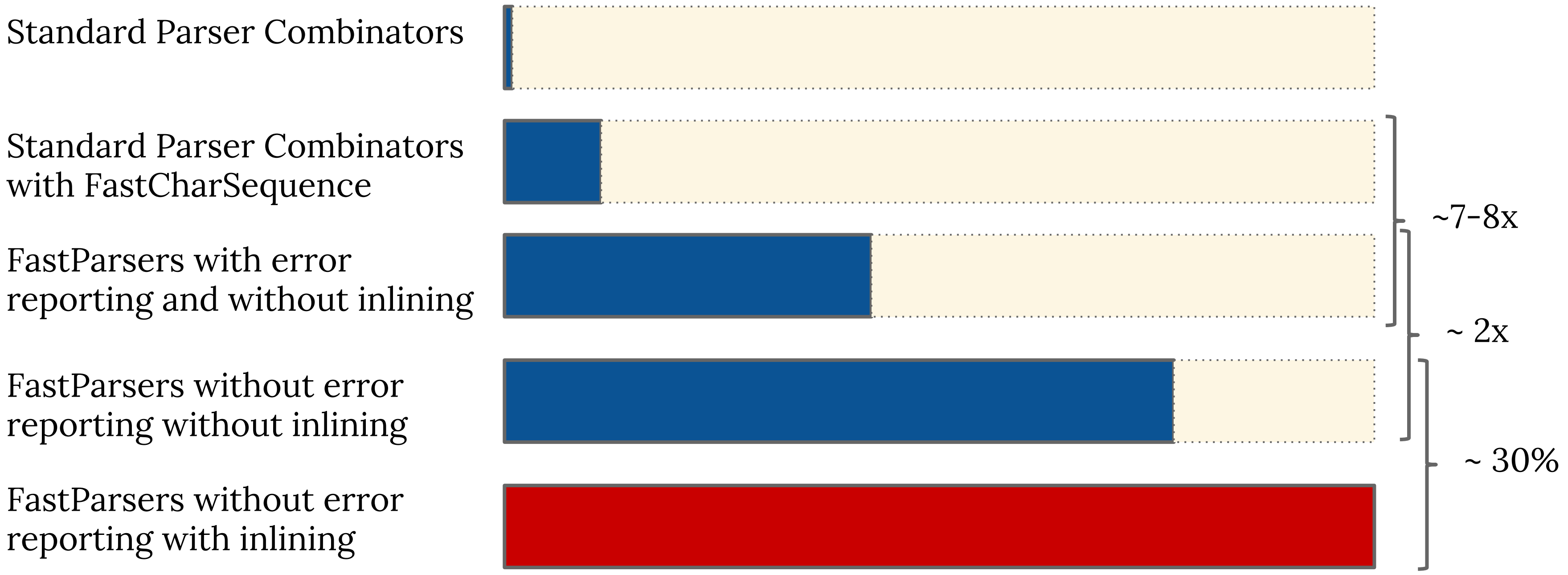
Key performance impactors



Key performance impactors



Key performance impactors


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