Staged Parser Combinators for Efficient Data Processing

Manohar Jonnalagedda* Thierry Coppey‡ Sandro Stucki*
Tiark Rompf †* Martin Odersky*

*LAMP ‡DATA, EPFL {first.last}@epfl.ch
†Oracle Labs: {first.last}@oracle.com

Abstract

Parsers are ubiquitous in computing, and many applications depend on their performance for decoding data efficiently. Parser combinators are an intuitive tool for writing parsers: tight integration with the host language enables grammar specifications to be interleaved with processing of parse results. Unfortunately, parser combinators are typically slow due to the high overhead of the host language abstraction mechanisms that enable composition.

We present a technique for eliminating such overhead. We use staging, a form of runtime code generation, to dissociate input parsing from parser composition, and eliminate intermediate data structures and computations associated with parser composition at staging time. A key challenge is to maintain support for input dependent grammars, which have no clear stage distinction.

Our approach applies to top-down recursive-descent parsers as well as bottom-up nondeterministic parsers with key applications in dynamic programming on sequences, where we auto-generate code for parallel hardware. We achieve performance comparable to specialized, hand-written parsers.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code generation, Optimization, Parsing

Keywords Parser combinators; multi-stage programming; algebraic dynamic programming

1. Introduction

Parser combinators [18, 20, 36] are an intuitive tool for writing parsers. Implemented as a library in a host language, they use the language's abstraction capabilities to enable composition. As a result, a parser written with such a library can look like formal grammar descriptions, and is also readily executable: by construction, it is well-structured, and easily maintainable. Moreover, since combinators are just functions in the host language, it is easy to combine them into larger, more powerful combinators.

However, parser combinators suffer from extremely poor performance (see Section 5) inherent to their implementation. There is a heavy penalty to be paid for the expressivity that they allow. A grammar description is, despite its declarative appearance, operationally interleaved with input handling, such that parts of the grammar description are rebuilt over and over again while input is processed. Moreover, since parsing logic may depend on previous input, there is no clear phase distinction between language description and input processing that could be straightforwardly exploited for optimizations without giving up expressiveness and therefore some of the appeal of parser combinators.

For this reason, parser combinators are rarely used in applications demanding high throughput. This is unfortunate, because they are so useful and parsing is such an ubiquitous task in computing. Far from being used only as a phase of compiler construction, parsers are plentiful in the big data era: most of the data being processed is exchanged through structured data formats, which need to be manipulated efficiently. An example is to perform machine learning on messages gathered from social networks. Most APIs return these messages in a structured JSON format transferred over the HTTP protocol. These messages need to be parsed and decoded before performing learning on them.

The accepted standard for performance oriented data processing is to write protocol parsers by hand. Parser generators, which are common for compilers, are not frequently used. One reason is that many protocols require a level of context-sensitivity (e.g. read a value n, then read n bytes), which is not readily supported by common grammar formalisms. Many open-source projects, such as Joyent/Nginx and Apache have hand-optimized HTTP parsers, which span over 2000 lines of low-level C code [1, 34]. From a software engineering standpoint, big chunks of low-level code

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660241

are never a desirable situation. First, there may be hidden and hard to detect security issues like buffer overflows. Second, the low-level optimization effort needs to be repeated for new protocol versions or if a variation of a protocol is desired: for example, a social network mining application may have different parsing requirements than an HTTP load balancer, even though both process the same protocol.

Parser combinators could be a very attractive implementation alternative, if only they weren't so slow. Their main benefit is that the full host language can be used to compose parsers, so context sensitivity and variations of protocols are easily supported. To give an example, we show the core of a combinator-based HTTP parser next.

Parsing Communication Protocols with Combinators The language of HTTP request and response messages is straightforward to describe. Here is an 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 ...
```

In short, an HTTP response consists of a status message (with a status code), a sequence of headers separated by line breaks, a final line break, and the payload of the response. A header is a key-value pair. The length of the payload is specified by the `Content-Length` header. We can express this structure as follows using the parser combinator library that comes with the Scala distribution:

```
def status = (
    ("HTTP/" ~ decimalNumber) ~> wholeNumber <~ (text ~ crlf)
  ) map (_.toInt)
def headers = rep(header)
def header = (headerName <~ ":") flatMap {
  key => (valueParser(key) <~ crlf) map {
    value => (key, value)
  }
}
def valueParser(key: String) =
  if (key == "Content-Length") wholeNumber
  else text
def body(i: Int) = repN(anyChar, i) <~ crlf
def response = (status ~ headers <~ crlf) map {
 case st ~ hs => Response(st, hs)
}
def respWithPayload = response flatMap {
  r => body(r.contentLength)
}
```

In the code above:

- the `status` parser looks for the code in a status message. It uses the sequential operators `~`, `~>` and `<~`. The latter two combinators ignore parse results on their left and right sides, respectively. Here, we are only interested in the status code (`wholeNumber`). The `map` combinator is then used to convert the code to an integer value.
- the `headers` parser uses the `rep` combinator to parse a sequence of headers; `header` parses a key-value pair. It uses the `flatMap` combinator to bind the result of parsing a `headerName` to a `key`, which is then passed on to `valueParser`. Note that this parser is context-dependent: it decides whether to parse the subsequent value based on the key. The `map` combinator is used to create the pair itself.
- the `response` parser parses a status message followed by headers: here the `map` combinator is used to create a `Response` structure (case class) from the results.
- Finally, `respWithPayload` parses and extracts the body of the message, based on the value of the `contentLength` field of the `Response` structure.

It is easy to see the appeal of parser combinators from the above example. In less than 30 lines of code we have defined a parser for HTTP responses, and it reads almost like an English description. Of course a bit more work is needed to support the full protocol; the complete parser we developed contains around 250 lines. Here, we have also omitted the definition of simple parsers such as `decimalNumber` and `wholeNumber`: the advantage being that these are implemented in a standard parser combinator library. Moreover, we can easily extend our implementation to handle key-value pairs more precisely by adding cases to the `valueParser` function, for example. If these 30 lines of re-usable high-level code could be made to perform competitively with 2000 lines of low-level hand-written C, there would be much less incentive for the latter to exist.

Since parser combinators compose easily, a fast implementation would even have the potential to surpass the performance of hand-written parsers in the case of protocol stacks: layering independent hand-optimized parsers may require buffering which can be avoided if the parsers can be composed directly.

Nondeterministic Parsers and Dynamic Programming A key performance aspect of parser combinators is that the usual implementations perform top-down recursive descent with backtracking, which has exponential worst case performance. Communication protocols do not usually contain ambiguities, so backtracking will not frequently occur in this setting and recursive descent is a viable strategy. However, protocol parsers are not the only use case for parsers in general and parser combinators in particular. Other important data processing applications, for example in natural language processing and bioinformatics, require nondeterministic parsers and highly ambiguous grammars. These applications involve computing a parse result over a sequence with respect to some cost function: we are not only looking for a parse result, we seek the best possible parse result. For

these use-cases, recursive descent with backtracking is inefficient. An efficient implementation comes in the form of a memoization/dynamic programming algorithm. A general technique, Algebraic Dynamic Programming (ADP) [10], can be used to describe sequence structures as grammars in a parser combinator library. Contrary to recurrence relations, a grammar captures the structure of a sequence more intuitively. ADP uses a grammar for the structure, and an algebra to compute over that structure. The added benefit of this separation is modularity: we can define multiple algebras (cost functions) for the same sequence structure. Primary use cases for ADP-style parser combinators are dynamic programming problems on sequences, found in the realm of bioinformatics, such as sequence folding. For very large sequences, it is beneficial to turn the evaluation strategy into a bottom-up algorithm: this exposes uniform layouts which are amenable to parallelization.

1.1 Contributions

We present a library of parser combinators that performs competitively to hand-optimized parsers. Lifting the performance level of parser combinators to that of hand-written parsers removes a key incentive to write parsers by hand, and enables more developers to reap the productivity benefits of high-level programming. In particular, we make the following contributions:

- We dissociate static from dynamic computations for parser combinators using Lightweight Modular Staging (LMS) [25]. The key insight is to allow static (grammar level) computation to treat pieces of dynamic (input handling) computation as first class values, but not the other way around. We leverage the Scala type system to ensure that no parser combinators can appear in the generated code. As a result we create a program that, at the first stage, eliminates the combinator abstraction, and generates an efficient parser. The first stage can still use the full host language for parser composition (Section 3).
- The stage distinction is non-obvious, due in particular to context-sensitive and recursive parsers. The key trade-off is between inlining code as much as possible vs. when and where to emit functions in the generated code. We use a mix of present-stage and future-stage functions at key junctions to generate efficient recursive descent patterns (Section 3.2). A similar technique also protects us from code blowup (Section 3.3).
- Staging alone is not sufficient to remove intermediate data structures that cross control flow joins in the generated code. We perform additional rewrites at the intermediate representation (IR) level to remove these data structures as well. These rewrites involve splitting and merging data structures at conditional expressions so that their components can be mapped to local variables (Section 3.4).
- We generalize our technique to parsers for ambiguous grammars. As an extreme case, we consider general dynamic programming on sequences, which can be reduced to a parsing problem through the use of ADP (Section 4). From a grammar specification of a dynamic program and a cost function, we generate a CYK style parser. In contrast to the recursive descent parsers generated for unambiguous grammars, we impose a bottom-up order for their ambiguous counterparts. At staging time, we compose *iterations* over lists (of results) instead of the lists themselves, and replace recursion by memoization (Sections 4.1, 4.3, 4.2).
- Imposing a bottom-up order also opens up the possibility to parallelize processing, as independent intermediate results can be computed independently [30, 32]. Our use of staging coupled with generative programming enables us to generate code that runs on the GPU for dynamic programs (Section 4.4). The GPU code computes an optimal cost function efficiently. In Section 4.5, we discuss how to retrieve the trace of this optimal cost. This trace is independent of the cost function itself, and can be applied to other algebras later. Compared to previous approaches, we trade extra memory usage in the forward computation for a better running time complexity for the backtrace computation.
- We evaluate the performance of our top-down parsers by comparing to hand-written HTTP and JSON parsers from the Nginx and JQ projects. Our generated Scala code, running on the JVM, achieves HTTP throughput of 75% of Nginx's low-level C code, and 120% of JQ's JSON parser. Other Scala based tools such as Spray are at least an order of magnitude slower. We evaluate our bottom-up parsers on two different bioinformatics algorithms. We compete with hand-written C code for the Nussinov algorithm, and show good scalability for both CPU and GPU code generated from parser combinators. (Section 5).

Section 6 discussed related work and Section 7 concludes. Before going any further, we give some insight into the abstraction overhead related to parser combinators in Section 2.

2. Parser Combinators

The introduction gave us a taste for implementing parsers with combinators. In this section we give insights on what causes them to be inefficient. For this, we show how they are commonly implemented.

Parser combinators are functions from an input to a parse result. For now we only consider deterministic parsers, in which case a parse result is either a success or a failure.

Figure 1 shows an implementation for parser combinators in Scala. The implementation is contained inside a `Parsers` module. The abstract `Parser[T]` class is generic over the the type of results it can parse. It extends a function type: the notation `T => U` is sugar for the class `Function1[T, U]`, which has an abstract `apply` method corresponding to function application. Creating a new parser is equivalent to creating a

```
trait Parsers {

  type Input = Int

  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
  }

  def source: Array[Char] //abstract, defined later

  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)
  }
}
```

Figure 1. An implementation of parser combinators

new instance of `Function1`, and providing an implementation of the `apply` method. The `Parser` convenience function simplifies the creation of a parser. We fix the `Input` type member to be `Int`, because we study parsers over character arrays with integer indices. We create an algebraic data type (or a **`case`** class in Scala terms) to represent a parse result. A `ParseResult` is either a success, in which case it contains the result, or a failure. The `flatMap` combinator, generic in a type parameter `U`, allows to make decisions based on the result of a parser. Incidentally, this function corresponds to the monadic bind for parser combinators. 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. Note that we use **`for`** comprehensions, which are Scala's equivalent for the `do` notation in Haskell.

In this paper we use the following combinators, in addition to those present in Figure 1:

- `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 applies `p` to parse the input until `p` fails. The result is a list of successive applications of `p`.
- `repN(n,p)` applies `p` exactly `n` times to parse the input. The result is a list of the `n` consecutive results of `p`.
- `repsep(p,q)` repeatedly applies `p` interleaved with the separator `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 define two simple parsers which accept a character based on a predicate:

```
def acceptIf(p: Char => Boolean) = Parser[Char] { pos =>
  if (pos < source.length && p(source(i)))
    Success(source(i), i + 1)
  else Failure(i)
 }
}

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

We are interested in only one parse result here. In the more general case, we could also collect all possible parse results over an input, that is, a list of results rather than a single one. In the event of multiple parse results being present, we are often interested in an optimal parse result based on some cost function. We will see in section 4 how such problems relate to dynamic programming. For both use cases, we still want to preserve the expressiveness and intuition with which parser combinators can describe these problems. In particular, it is better to express a parse result and its cost function independently, even if we would like to compute them together.

2.1 The Overhead of Abstraction

The above, functional implementation 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.

One would expect a virtual machine like the JVM to inline most of these calls; the JVM's heuristics are however geared to optimize hot code paths. We may only be able to inline a parser that is called very often. With a pluggable VM, such as Truffle/Graal [39], we could hint the virtual machine to inline these code paths. The important insight here is that we want a more deterministic way of optimizing the method calls. This will also open up further optimization opportunities down the line.

- 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. With ambiguous grammars and dynamic programming, the problems are exacerbated further. Because parse results and cost functions are expressed independently, many large intermediate results may be created before the optimal result is eventually computed.

In summary, it is precisely the language abstraction mechanisms that enable us to *compose* combinators that are hindering our performance. In the following sections, we show how to get rid of the penalty.

3. Staging Parser Combinators

One way to remove composition abstraction in parser combinators follows a well known technique, multi-stage programming, or staging [35]. The idea is to separate computation into stages, where at each stage, parts of the program are partially evaluated, leaving behind a simpler, or faster, program. When a parser combinator is run on an input, its control structure is already fixed: it is statically known. The parsing itself depends on the input. We want to stage the static parts away, so that all that remains is the core logic of parsing itself. We perform the staging using the Lightweight Modular Staging (LMS) framework [25].

Lightweight Modular Staging LMS is a framework for developing embedded DSLs in Scala, that provides a library for domain-specific optimizations and code generation. A program written in LMS is compiled in two stages. During the first stage, the code is converted into an intermediate representation (IR), from which code is generated. We refer to this step as staging time. This generated code, representing an optimized version of the original DSL program is then run during the second stage. The IR is modular in the sense that it can be easily extended with domain-specific optimizations using classic programming features like pattern matching. Sitting atop the Scala Virtualized compiler [24] also enables domain-specific overriding of basic language constructs like conditional expressions, loops and object creation.

LMS differentiates between staged and unstaged computations using the `Rep` 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 IR node `Add(a, b)`. The intermediate representation gives the DSL developer a pragmatic way to specify domain specific optimizations. For example, we can pattern match on the parameters of the `Add` node to detect when one of the operands is the constant `0`, and rewrite the expression to be the other operand. This way, `add2(a, 0)` generates code for `a` only.

The core LMS library provides interfaces, IR nodes and code generation for many common programming constructs, such as conditionals, Boolean expressions, arithmetic expressions and array operations, among others. These can therefore be used out of the box.

3.1 Parser Combinators, Staged

Efficient staging of parser combinators is a question of correctly identifying which computations to stage, and which computations to evaluate away. Let us look at the signature for parser combinators once again:

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

Clearly, neither the input position nor the contents of the parse result are known statically; they have to be staged. However, we know how the parsers will be composed. This means that we can evaluate function composition at staging time. We represent a parser as an *unstaged* function on *staged* types:

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

A function `def f: Rep[T] => Rep[U]`, when applied to a parameter of type `Rep[T]`, is evaluated at staging time: at the call site, the body of the function is generated, instead of a call to the function. This amounts to inlining the function at the call site. On the other hand, functions of type `Rep[T => U]` represent staged functions; the generated code will contain a function declaration, as well as a function call, if the function is applied. The difference between both is clearer by desugaring the function type: unstaged functions are instances of `Function1[Rep[T], Rep[U]]`, while staged functions are instances of `Rep[Function1[T, U]]`.

We stage all composition functions in a similar way. Two rules drive our systematic staging of parsers:

- A primitive type `T` is converted into `Rep[T]`. By primitive type, we mean a type that is input dependent: inputs and parse results.
- A function type `T => U` is converted into `Rep[T] => Rep[U]`

This way, higher-order functions also get inlined at call sites. The interface for staged parser combinators is given in figure 2. The implementations for the combinators follow those of unstaged parser combinators in section 2. We show the code for the `flatMap` combinator as an example:

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

To get some insight as to how staging works for parser combinators, consider the following simple parser, which accepts the letter 'h' followed by the letter 'i':

```
val hi = (accept('h') ~ accept('i'))
hi(0)
```

Applying the function to an input will inline the parser. The generated code we get looks like the following:

```
val idx = 0
val len = input.length

val p1: ParseResult[Char] = // parsing 'h'
  if (idx < len) {
    val res = input(idx)
    if (res == 'h') Success(res, idx + 1)
    else Failure(idx)
  } else Failure(idx)

val p2: ParseResult[(Char, Char)] = // parsing 'i'
  if (p1 == Success) {
    val idx1 = p1.next
    if (idx1 < len) {
      val res = input(idx1)
      if (res == 'i') Success((p1.res, res), idx1 + 1)
      else Failure(idx1)
    } else Failure(idx1)
  } else p1
p2
```

We notice that there are no Parser objects in the generated code. The body of the `accept` function is inlined, and so is the body of the sequence function `~`.

3.2 Recursion

Even the most basic parsers have some form of recursion. Simply using unstaged functions will create infinite loops during code generation because recursive calls are unfolded and inlined during staging time. We are forced to stop the recursion by generating a staged function for recursive parsers. A recursive parser needs to be explicitly declared using the

```
abstract class Parser[T]
  extends (Rep[Input] => Rep[ParseResult[T]]){
  def map[U](f: Rep[T] => Rep[U]): Parser[U]
  def flatMap[U](f: Rep[T] => Parser[U]): Parser[U]
  def filter(f: Rep[T] => Rep[Boolean]): Parser[T]
  def ~[U](that: Parser[U]): Parser[(T, U)]
  def | (that: Parser[T]): Parser[T]
}
```

Figure 2. Interface for staged parsers

`rec` combinator which performs this lifting. The `rec` combinator works using the classical memoization scheme. The first time it sees a parser, it stores, in a static map, a staged version of the parser function. The next time the same parser is seen, we replace it with a function application, using the staged function stored before. In the generated code, we have a function application. The `rec` combinator is therefore similar to a fixpoint combinator:

```
def lift[T, U]: (f: Rep[T] => Rep[U]): Rep[T => U] = ...

val store = new Map[Parser, Sym]
def rec[T](p: Parser[T]): Parser[T] = {
  store.get(p) match {
    case Some(f) =>
      Parser[T] { i => f(i) }
    case None =>
      val funSym = fresh[Input => ParseResult[T]]
      store += (p -> funSym)

      val f = (i: Rep[Input]) => p(i)
      createDefinition(funSym, lift(f))

      store -= p
      Parser[T] { i => funSym(i) }
  }
}
```

The `Sym` type, and the `fresh` and `createDefinition` functions are part of the LMS internals. `Sym` represents a symbol in the IR of an LMS program. A new symbol for a specific type is created using `fresh`, and `createDefinition` links a `Rep` type to a symbol in the IR tree. Finally, the `lift` function converts an unstaged function into a staged one.

3.3 Reducing Code Generation Blow-up

Consider the following parser:

```
def aParser = (a ~ b | c) ~ d
```

where none of the parsers are recursive. Following our optimizations from above, there is a risk that code generation blows up: The `c` parser is tried when either `a` or `b` fails: the code for parser `c` will therefore be inlined in the fail branches for both `a` and `b`. The code for parser `d` is then further inlined in each of the success branches; it is inlined 3 times. Aggressive and naive inlining results in an exponential code

blow-up due to many different branches being created. Once again, simply staging a parser implementation is insufficient.

The explosion in branches happens because of combinators that offer multiple paths of computation. The alternation combinator | is the culprit here. We need to maintain a diamond control flow in the generated code in the presence of alternation. Once again, we use staged functions. We wrap the left alternative in a function. We also wrap the parser created by the alternation into a function. The implementation of | is a bit more complex this time:

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

3.4 Staged Records

So far, we have rid ourselves of the abstraction of combinators, and protected ourselves from recursion and code generation blowup. The shape of the generated code still needs to be improved. We still create many intermediate data structures. Recall the `hi` parser from above, and the code it generates.

Each time we use an elementary parser, we create an instance of a `ParseResult`. Further down, we use field lookups to inspect this object. There is some overhead due to boxing/unboxing.

But we can use staging again! The idea is to represent parse results as staged record/struct implementation. The idea is reminiscent of what we did with functions. We staged function declaration and application away by using unstaged functions. Similarly, we represent parse results as records, or structs, but stage the record creation, and field lookups, away.

For this, we re-use the generic struct interface present in LMS [26] (Figure 3). The struct interface provides IR nodes for struct creation and field lookup. During code generation a record is produced. To optimize struct creation we operate on the IR level. In a first step, we override conditionals that produce structs to split the conditional expression among every field. In a second step, we re-merge the conditional expressions, with the fields remaining split at the root of the conditional expression.

As a result of the split and re-merge, the `hi` parser above now generates the following code:

```
val idx = 0
val len = input.length
```

```
def Success[T](res: Rep[T], next:Rep[Int]) =
  Struct(
    classTag[ParseResult[T]],
    "res"   -> res,
    "empty" -> false,
    "next"  -> next
  )

def Failure[T](next: Rep[Int]) =
  Struct(
    classTag[ParseResult[T]],
    "res"   -> ZeroVal[T],
    "empty" -> true,
    "next"  -> next
  )

override def ifThenElse[T](cond: Rep[Boolean],
  left: Rep[T], right: Rep[T])
= (left, right) match {
  case (Struct(tagA, elems), Struct(tagB, elemsB)) =>
    assert(tagA == tagB)
    val elemsNew = elems.zip(elemsB) map { (left, right) =>
      if (cond) left else right
    }
    Struct(tagA, newElems)
}
```

Figure 3. Using structs for parse results, and splitting

```
var res: (Char, Char) = null
var next: Int = 0
var empty: Boolean = true

if (idx < len){
  val r1 = input(idx)
  // parsing 'h'
  if (r1 == 'h') {
    val idx2 = r1 + 1
    val r2 = input(idx2)
    // parsing 'i'
    if (r2 == 'i') {
      res = (r1, r2)
      next = idx + 1
      empty = true
    }
  }
}
ParseResult(res, next, empty)
```

Note that we only create the parse result after having parsed fully, and have inlined the parsing of 'i' into the success branch of 'h'.

Top-down, recursive-descent parsing works well for protocols, because the grammars are deterministic, with limited backtracking. In general, however, the search for a parse can be exponentially expensive, and that will dominate the inefficiencies due to boxing/unboxing and indirection. This problem occurs especially with ambiguous grammars. The

extreme case of ambiguity is when an input sequence has an exponential number of possible parses. In the next section, we see how to deal with non-deterministic, ambiguous grammars.

4. Staged Parser Combinators for Dynamic Programming on Sequences

We now consider parser combinators for ambiguous, non-deterministic grammars. These grammars can yield more than one parse result per input. Often, we are interested in one parse result, namely the optimal parse result with respect to some cost function. This is where techniques like memoization/dynamic programming come into play. In this section, we introduce the general method which is used to describe such grammars, using parser combinators. This method is known as Algebraic Dynamic Programming (ADP) [10, 14], and has practical applications in biology. We then focus on staging combinators for an optimal cost function. By processing the problem in a bottom-up fashion, we expose regular structure, which is good for parallelism. Staging permits, in addition to abstraction and intermediate data structure removal, to target GPUs as an additional code generation platform.

Matrix-chain Multiplication Let us start with a simple application problem: given a sequence of matrices m_i of appropriate dimensions, we want to determine the order of multiplication that minimizes the number of scalar multiplications. This is a classic case of dynamic programming. The problem satisfies the Bellman property: optimal solutions are constructed from optimal sub-solutions. Let $M[i,j]$ denote the optimal cost of multiplying matrices i to j, the solution is described by the following recurrence relation:

$$M[i,j] = 0 \qquad \text{if } i = j, \text{else}$$
$$M[i,j] = \min_{i \le k < j} \left\{ \begin{array}{l} M[i,k] + M[k+1,j] \\ +\text{rows}(m_i) \cdot \text{cols}(m_k) \cdot \text{cols}(m_j) \end{array} \right\}$$

We can memoize intermediate results in a matrix, and look them up when we need them.

Parser combinators for dynamic programming The recurrence relations above do not capture the *structure* of the problem in an intuitive manner. We can express the problem as a grammar instead:

```
val chain = tabulate((
  singleMatrix map single
| (chain ~ chain) map mult
) aggregate h)
```

A chain of matrices is made by either a single matrix, or two consecutive sub-chains. The `mult` and `single` functions act on a parse result, while `tabulate` memoizes the computation. The `aggregate` function, as its name indicates, combines results of a parse based on a given function.

To achieve this, we need a parser combinator implementation that produces multiple results. Figure 5 shows this implementation. Now, parsers are functions from a pair of integers to a list of parse results. The pair of integers represent the subsequence that we wish to parse. The alternation combinator is non-deterministic, so we concatenate results from the left parser and those from the right parser. The sequence combinator combines all results on the left side with those on the right (a cross-product). In essence, monadic operations on parsers are mapped to monadic operations on the underlying lists, as shown by Wadler [36].

The `mult`, `single` and `h` functions are customizable: we may be interested in the optimal cost of multiplication, but we may also be interested in simply visualizing the result. More interestingly, we may want to visualize the optimal result. Algebraic dynamic programming is a formalism allowing us to specify these possibilities. Formally, ADP decomposes into four components:

- A signature Σ that defines the input alphabet $\mathcal{A}$, a sort symbol $\mathcal{S}$ and a family of operators $\circ : s_1, ..., s_k \rightarrow \mathcal{S}$ where each s_i is either $\mathcal{S}$ or $\mathcal{A}$.
- A grammar $\mathcal{G}$ over Σ operating on a string $\mathcal{A}^*$ that generates sub-solutions candidates.
- An algebra, that instantiates a signature and attributes a score to extracted sub-solutions. The sort symbol and the signature functions have implementations.
- An aggregation function $h : \mathcal{S}^* \rightarrow \mathcal{S}^*$ retaining sub-solution with appropriate score (usually optimal ones).

An implementation of a signature and two algebras for the matrix-chain multiplication problem are given in Figure 4.

The `MatMultSig` trait defines the signature for the problem. We define two operations, `single` representing a single matrix and `mult` representing the multiplication of two matrices. The `h` function is for aggregation. Note that the arguments to `mult` are of type `S`. The `CostAlgebra` trait provides concrete types for `A` and `S`, as Scala tuples. It also contains concrete implementations for `single`, `mult` and `h`. These implementations correspond to the recurrence relation seen above. As its name suggests, the `CostAlgebra` looks to minimize the optimal cost.

The `PrettyPrint` algebra, on the other hand, defines a visualization for the problem. The aggregation function is the identity function: we want to see all possible tree constructions for chaining matrices.

The `MatMultGrammar` trait gives the structure for the problem, as described above. Note that we mix in parser functionality (the `Parsers` trait) with the `MatMultSig` signature. Finally, `MatMult` ties a grammar to a *concrete* algebra to a grammar using mix-in composition.

Single sequences The parser combinator library presented above works on single sequences, just like recursive-descent combinators. Along with the matrix multiplication problem above, other dynamic programming problems that benefit from this approach can be found in the bioinformatics realm, mostly in sequence folding. From a parsing point of view,

```
trait MatMultSig {
  type A, S
  def single(a: A): S
  def mult(l: S, r: S): S
  def h(xs: List[S]): List[S]
}
trait CostAlgebra extends MatMultSig {
  type A = (Int, Int)      // input matrix (rows,columns)
  type S = (Int, Int, Int) // product (rows,cost,columns)
  def single(a: A) = (a._1, 0, a._2)
  def mult(l: S, r: S) =
    ( l._1,
      l._2 + r._2 + l._1 * l._3 * r._3,
      r._3 )

  def h(xs: List[S]) = List(xs.minBy(_._2))
}
trait PrettyPrint extends MatMultSig {
  type A = (Int, Int); type S = String
  def single(a: A) = "[" + a._1 + "x" + a._2 + "]"
  def mult(l: S, r: S) = "(" + l + "*" + r + ")"
  def h(xs: List[S]) = xs
}
trait MatMultGrammar extends Parsers with MatMultSig {
  val chain = tabulate((
    singleMatrix map single
  | (chain ~ chain) map mult
  ) aggregate h)
}
object MatMult extends MatMultGrammar with CostAlgebra
```

Figure 4. The matrix chain multiplication problem in ADP

we are indeed trying to find the *best* structure for a given sequence.

Another prominent class of dynamic programming problems in bioinformatics is sequence alignment, with the most well-known algorithm being the Smith-Waterman algorithm: we want to find the best alignment of *two* sequences. While ADP can support dynamic programs on multiple sequences [30], state of the art implementations for sequence alignment face challenges that are orthogonal to its representation as a grammar. Not only do the size of the sequence require a stochastic approach, but there are also a number of hardware-specific optimizations that can be applied to accelerate Smith-Waterman even further [28, 29].

We therefore choose to focus on single sequence problems in this paper; not only are sequence sizes typically smaller (fit on the GPU), but the expressivity and performance gains are also higher for these problems.

4.1 Top-down to Bottom-up Evaluation

Running the `chain` parser above on an input sequence `in` with the argument `(0, in.length)` will compute the best cost for the sequence. The parser runs in a top-down manner, calling rules recursively on smaller subsequences. The advantage of

```
abstract class Parser[T] extends ((Int, Int) => List[T]) {
  def | (that: Parser[T]) = Parser[T] {
    (i, j) => this(i, j) ++ that(i, j)
  }

  def ~[U](that: Parser[U]) = Parser[(T, U)] {
    (i,j) => if (i < j) {
      for(k <- i until j;
          x <- this(i, j);
          y <- that(k, j)
      ) yield (x,y)
    } else List()
  }

  def map[U](f: T => U) = Parser[U] {
    (i, j) => this(i, j) map f
  }

  def aggregate(h: List[T] => List[T]) = Parser[T] {
    (i, j) => h(this(i, j))
  }

  def filter(p: (Int, Int) => Boolean) = Parser[T] {
    (i, j) => if (p(i, j)) this(i, j) else List()
  }
}

def el(in: Input) = Parser[T] {
  (i, j) => if (i + 1 == j) List(in(i)) else List()
}

def tabulate[T](p: Parser[T], mem: Array[Array[T]])
  = Parser[T] {
  (i, j) => if (mem(i)(j).isEmpty) {
    val tmp = p(i, j)
    mem(i)(j) = tmp
    tmp
  } else mem(i)(j)
}
```

Figure 5. An implementation of parser combinators for dynamic programming

this strategy is that unreachable subsequences are not parsed in sparse problems. For our application cases, however, it is common that all subsolutions need to be computed. It is therefore useful to use a bottom-up strategy instead:

```
def bottomUp(p: Parser[T]) {
  val n = in.length
  (0 until n).foreach { d =>
    (0 until n - d).foreach { i =>
      val j = i + d
      p(i, j)  // call parser between i and j
    }
  }
}
```

```
abstract class Foreach[T]
  extends (Rep[T] => Rep[Unit]) => Rep[Unit] {

  def map[U](g: Rep[T] => Rep[U]) = Foreach[U] {
    f: (Rep[U] => Rep[Unit]) => this {
      x: Rep[T] => f(g(x))
    }
  }

  def filter(p: Rep[T] => Rep[Boolean]) = Foreach[T] {
    f: (Rep[T] => Rep[Unit]) => this {
      x: Rep[T] => if(p(x)) f(x)
    }
  }

  def flatMap[U](g: Rep[T] => Foreach[U]) = Foreach[U] {
    f: (Rep[U] => Rep[Unit]) => this {
      x: Rep[T] => g(x)(f)
    }
  }

  def ++(that: Foreach[T]) = Foreach[T] {
    f: (Rep[T] => Rep[Unit]) = {
      this(f); that(f)
    }
  }
}
```

Figure 6. An implementation of Foreach

Here, we are taking advantage of the Bellman's optimality principle to process all subproblems of a certain size (`d`) before moving to the next size. This evaluation strategy computes all the results on the same anti-diagonal (also known as *wavefront*) and progresses along the diagonal of the tabulation matrix. All recursive calls to a parser become simple table lookups.

4.2 Staging and Memoization

With recursive-descent parsers, we introduced the `rec` combinator for handling recursion. The `tabulate` combinator plays an analogous role for memoization with dynamic programming. If a parser has already been seen, a table lookup is generated. Otherwise, the code for computing the current result is generated. Once again, evaluation order ensures that code generation will not loop forever.

4.3 Staging and Lists

As with combinators for recursive-descent parsing, we first introduce `Rep` types to stage away composition. Naive staging will give us a result type of `Rep[List[T]]`. However, as mentioned above, the goal is to return a single optimal solution, and we would like to avoid the additional boxing overhead of lists. The key to achieving this goal is to compose *iterations over lists*, represented by the type `Foreach[T]`, rather than the lists themselves. The signature of `Foreach` (see Figure 6) mirrors that of its namesake combinator: it takes a function `f`, of type `T => Unit`, and applies it to every element of the collection.

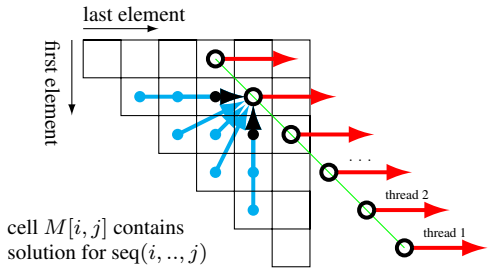

Figure 7. Threads progress jointly along matrix diagonal. Dependencies can be reduced to immediately preceding matrix cells.

In the implementation of parsers above, we replace `List[T]` by `Foreach[T]`, and change the signature of the `aggregate` function to return one optimal solution:

```
abstract class Parser[T]
  extends Rep[(Int, Int)] => Foreach[T] {
  ...
  def aggregate(h: Foreach[T] => Rep[T]) = Parser[T] {
    (i, j) => h(this(i, j))
  }
}
```

The use of `Rep` types will inline function calls, as previously. The end result of the staging step and the use of the `bottomUp` function is a tight nested loop over the resulting matrix that proceeds along the diagonal.

With mutually recursive parsers, production rules need to be ordered to ensure that a parser is computed only when all its dependencies are valid (dependency analysis). In addition, we can make the following observations:

- Referring to the intermediate solution matrix (Figure 7), all possible dependencies are contained in a sub-matrix. By induction, it suffices that the immediately preceding elements in the row and the column are valid for all other dependencies to be satisfied. Hence elements on a diagonal can be computed in parallel (*wavefront*).
- In the presence of multiple grammar rules, we evaluate all rules on a given subsequence at once; dependency analysis provides us with a correct evaluation order, assuming that the grammar is correct (satisfies the Bellman's property, and has no cyclic rules that would cause infinite loops in top-down evaluation).

4.4 GPU parallelization

Modern graphic cards[1] are powered by massively parallel processors running hundreds of cores, each able to schedule multiple threads. The threads are grouped by warps (32 threads) and blocks, and scheduled synchronously: a diver-

[1] We focus on Nvidia/CUDA features; other frameworks have similar concepts.

gence in execution path causes both alternatives to be scheduled sequentially, thereby stalling some threads. Two levels of memory are exposed to the programmer: global memory accessible by any thread, and a faster local memory accessible by threads in the same block. In many applications, the computational power of GPUs outperforms the memory bandwidth such that (global) memory accesses is often the bottleneck.

The key insight is that GPU programs need to be *regular*, both in their computation logic (all threads should be kept busy) and memory accesses (contiguous, or *coalesced*). In dynamic programming, the cost function is usually simple and regular with respect to the combination of sub-solutions.

Since elements computed in parallel are along one diagonal, the underlying matrix needs to be stored diagonal-by-diagonal. To respect the dependency order, threads progress along the rows of the matrix and synchronize with the neighboring thread (to validate column dependencies) before moving to the next diagonal. This synchronization is done by active waiting on threads between computations of diagonals [40].

We use the heterogeneous code generation capacities of LMS to dissociate these hardware-specific decisions from the parser description. The progress on diagonals is given by the `bottomUp` function above. We generate GPU specific code for the body (`p(i, j)`) of these loops.

4.5 Computing the Backtrace

In addition to the optimal cost of a dynamic program, one is typically interested in the corresponding parse result. In our matrix multiplication example above, we could construct such results simply by mixing in the `PrettyPrint` algebra and the `CostAlgebra`. This strategy does not extend easily to the GPU: the cost function is regular, but pretty printing involves string creation and storage, which adds unnecessary computation overhead on the GPU. It is better to treat `PrettyPrint` as a backtracing algebra, and reconstruct the best parse tree *after* the optimal cost has been found. In fact, it is possible to decouple the costing and backtracing algebras completely, by precomputing *backtraces*. The remainder of this section describes this process.

The cost function of a dynamic program computes its optimal cost, which corresponds to one or more optimal parse trees. A backtrace is a linear representation of an optimal parse tree. Figure 8 depicts the backtrace for an example instance of the matrix multiplication problem. Given matrix dimensions in the left third of the figure, there is one optimal multiplication order, given in pretty-print form. The optimal tree is overlaid on the cost matrix. The root of this tree naturally lies on cell $[0, 3]$, which contains the optimal cost for multiplying all four matrices. In the final third of the figure, we show the backtrace of the tree, that is, its in-order traversal.

The backtrace computation is done in two phases: during the forward cost computation phase, we store, along with

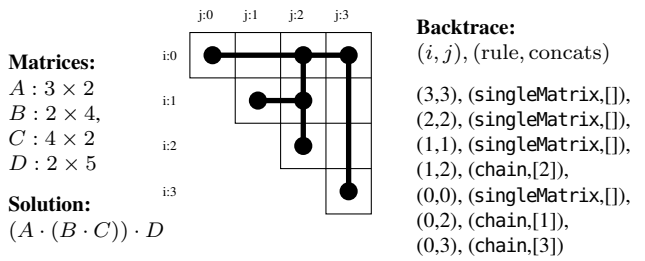

Figure 8. A backtrace for an instance of the matrix multiplication problem

the optimal cost, some backtrace-related information, which we call the *paper trail*. This trail records what decisions we have made to reach a given cell. Once the forward phase is complete, we start from the root of the tree, and reconstruct the optimal backtrace as the in-order traversal of the tree. We can then reuse this backtrace and apply it to various algebras.

Computing the paper trail The paper trail is computed along with the cost function. We need to store, for each cell, how we got there. The only two combinators that contribute to this decision are sequencing (~) and alternation (|). Alternation tells us which production rule was chosen, while sequencing tells us where a subsequence was split. The number of splits in a subsequence is determined by the number of concatenation combinators in a given rule, and corresponds to the number of children of the relevant node in the parse tree. For the matrix multiplication problem (Figure 4), there can be at most one split, resulting in a binary parse tree with leaves and internal nodes corresponding to applications of the `singleMatrix` and `chain` rules, respectively.

Therefore, at every cell, we store a pair `(ruleId, concats)`, where `ruleId` corresponds to a chosen alternative, and `concats` is a list of splits due to concatenation. We attribute a `ruleId` for every alternative in a `tabulate` combinator in a prior grammar analysis phase.

Backtrace Construction After the forward phase, both the cost matrix and the backtrace matrix have been filled out. The actual backtrace corresponding to the optimal tree is reconstructed by running a simple recursive in-order tree traversal starting from the root cell $[0, n]$. This yields a list representation of the backtrace (`List[(ruleId, concats)]`). The final third of Figure 8 depicts this representation.

Reuse of the Backtrace Now that the backtrace is constructed, it can be applied to any algebra. By following the list in order, we apply the score function of the given algebra to the rule specified by the `ruleId` parameter. Previous elements are guaranteed to be constructed beforehand.

A Note on Complexity Let t be the number of tabulations, c be the maximum number of concatenations and r be the number of rules in a grammar. Since these factors are constant for a given grammar, we will only take them into account when they appear as exponents in the following complexity bounds.

The space complexity for the optimal cost matrix $O(n^2)$. Storing the paper trail takes an extra $O(n^2)$ of memory. The running time complexity of the forward computation phase is not affected by the paper trail, and remains $O(n^{2+c})$.

The backtrace reconstruction phase is a tree traversal, where the depth of the tree is n (as the cost matrix is of size n^2). The length of the backtrace is bounded by the size of this tree, which is $O(n)$. This is also the running time complexity of the backtrace phase.

Finally, if we want to apply the backtrace to another algebra, the running time is once again bounded by the length of the trace.

Previous approaches combining ADP and backtracing allow to separate optimal cost computation and other algebras [14, 30]. The difference is that they do not explicitly compute a backtrace, but reuse the matrix computed in the forward phase. This saves on memory, but applying a new algebra has a higher complexity ($O(n^2)$).

5. Evaluation

5.1 Building Staged Libraries

We implemented parser combinator libraries for both recursive-descent parsers and dynamic programming parsers on top of the LMS framework. The core LMS library provides functionality that can be, and has been used for implementing many different DSLs [26]. Such functionality includes support for staged structs, arrays, lists, and strings, which are relevant to parser combinators. It also includes code generation support for Scala, C, and CUDA, when relevant. As mentioned in Section 3, LMS gives us inlining, dead code elimination and common subexpression elimination for free. Given the above, we regard the development of LMS itself as an upfront cost, which has been amortized over the development of DSLs on top of it. Arguably, this is analogous to a general purpose language being an upfront cost to the development of a library in this language.

The implementation of staged DSLs took a bit more effort than writing parser combinator libraries, however. This is expected: in DSL terminology, a library is a *shallow embedding*, while a staged DSL is a *deep embedding*. In particular, the additional development cost with respect to a library was spent for:

- ensuring that we reuse existing functionality as much as possible. The danger with code generation frameworks is that it is easy to generate code for a specific use case; some optimization patterns risk being repeated for many cases, which leads not only to bad software practice, but also to optimization patterns being lost to other DSLs.
- extending the struct functionality with optimizations for conditional expressions.
- extending the framework with the `Foreach` abstraction.
- adding CUDA code generation support for the parallel patterns exposed by dynamic programming problems.

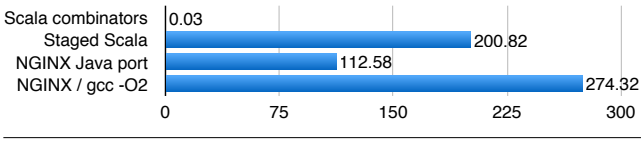

Figure 9. HTTP parser throughput in MB/s

- manually inspecting generated code to ensure that intermediate data structures are indeed eliminated. In addition to functionality and performance testing, we naturally had to make sure that the performance gains we obtain are due to abstraction overhead removal.

On the plus side, the effort spent on extensions to the LMS framework is well invested, as the extensions can be used by future projects as well.

5.2 Recursive-descent Parsers

We evaluated our staged recursive descent parsers, by implementing a HTTP parser and a JSON parser. The benchmarking environment for recursive descent parsers consisted of an Intel i7 3720-QM with 16GB of RAM, we use Scala 2.10 (-optimise) on Oracle JDK7 and GCC 4.8.2 with the most efficient optimization for given programs (-O2 or -O3). Our staged parser combinators generate Scala code for both the HTTP and JSON parser. We ran our Scala/Java benchmarks using Scalameter [23], a benchmarking and performance regression testing framework for the JVM. This framework handles JIT warm-ups as well as running a benchmark till performance stability.

HTTP Response Parser The first test is a comparison of the parsing throughput of different implementations of an HTTP response parser. To do that, we used a dataset of Twitter messages with 100 HTTP responses totaling 8.15 Mb of data that decompose in 54.2 Kb of HTTP headers and 8.10 Mb of JSON payload. The messages were obtained by querying the Twitter Search API. The HTTP parsing happens only on headers.

We compare our staged combinators with both Scala's standard parser combinators and the Nginx proxy client, a hand-optimized, fast, open-source implementation. We also ported this client to Java, for comparison. Figure 9 shows the results. As mentioned in the introduction, native Scala parser combinators are a non-option. The JIT engine of the JVM seems not able to optimize across functions. We perform better than the Java port. We think that this difference may be due to the JVM performing better speculation for conditional expressions generated by our code, than on the state-machine like structure present in the hand-optimized code. Similarly, the C version is better than staged parser combinators. We presume that the O2 compiler optimizes switch/case statements efficiently.

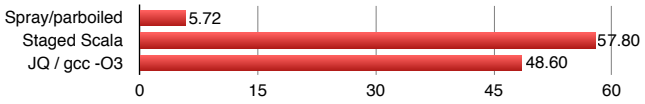

Figure 10. JSON parser throughput in MB/s

JSON Parser Our second evaluation (Figure 10) compares parsing of the JSON payload of the previous messages. The JSON grammar is given as:

```
val jsonParser = {
  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)

  value
}
```

This parser looks very similar to a standard parser combinator implementation [21, Chapter 31]. A JSON object is either:

- a primitive `value`, such as a decimal, string literal, a 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 compare with Spray-json, a JSON parser for the popular Spray web toolkit for Scala. Its JSON parser is written using a parser combinator library. We also compare with JQ, a popular, efficient, command line tool implemented in C to process queries on JSON. Once again, we see that a traditional parser combinator implementation performs poorly. On the other hand, staged combinators compete very well with the C implementation, and even surpass them.

5.3 Dynamic Programming

Dynamic Programming on CPU and GPU Our benchmarking environment for dynamic programming (CPU and GPU) consisted of a dual Intel Xeon X5550 with 96GB of RAM with an Nvidia Tesla C2050 (3Gb RAM) graphic card. We measured the running time of two different bioinformatics algorithms for RNA sequence folding. RNA folding consists of identifying matching basepairs that produce 2D features such as hairpins, bugles and loops in the secondary structure of RNA molecules [13, 15]. For both algorithms, we generate C code from staged parser combinators. For parallelization, we generate CUDA, additionally.

The Nussinov Algorithm The Nussinov algorithm maximizes the number of matching basepairs. Its running time complexity is $O(n^3)$. Its grammar is given by:

```
val s = tabulate(
    empty                          map nil
  | el ~ s                         map left
  | s ~ el                         map right
  | (el ~ s ~ el filter basePair) map pair
  | s ~ s                          map split
  ) aggregate h
```

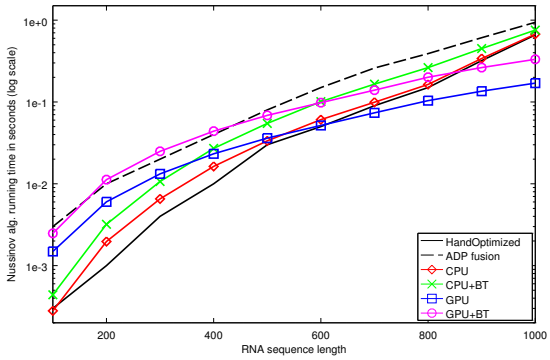

Figure 11. Nussinov algorithm running time

The Nussinov algorithm identifies 4 possible folding alternatives:

- an element on the left side is dropped.
- an element on the right side is dropped.
- two elements match according to the `basePair` filter.
- a fold is split into two smaller folds.

Our results are displayed in Figure 11. We show the running time for four variants of our generated code, along with a hand optimized C version, and an ADPfusion version:

- ADPfusion is an embedded Haskell DSL for ADP. It uses stream fusion [4] to eliminate intermediate list creation, and performs close to hand-optimized C on the CPU. We compare ADPfusion's optimal cost calculation with ours, omitting the computation of the backtrace.
- The CPU versions of our implementations run on a single thread. The CPU+BT version fills the backtrace matrix and runs the procedure to construct the backtrace.
- Similarly, our CUDA implementation is presented both with and without the backtracing.

The CPU version without backtracing has similar performance to hand-optimized C code. Indeed, manual inspection of the generated code reveals very close implementations. The ADPfusion version is about $2\times$ slower, which is in line with some remaining overhead that stream fusion is unable to completely eliminate [14, Section 7]. For sequences larger than 800 elements, the CPU version takes a slight extra performance hit: this can be attributed to data overflowing caches. We notice that parallelizing on the GPU is worthwhile only for significant sequence sizes. Also since the costing algorithm is fairly simple, we notice a clear overhead when backtracing is enabled.

The Zuker Algorithm The Zuker algorithm also predicts the optimal secondary structure of an RNA sequence. Instead of maximizing the number of basepairs, it minimizes free energy. Because the free energy is computed from hun-

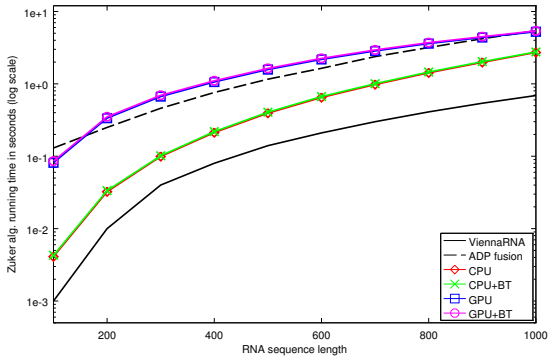

Figure 12. Zuker algorithm running time

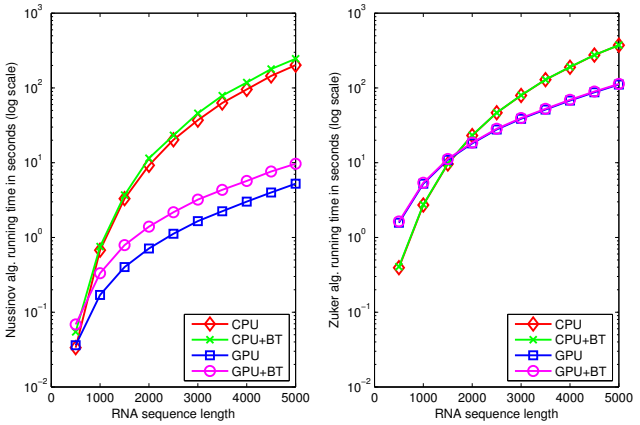

Figure 13. Nussinov and Zuker algorithms scalability

dreds of coefficients based on physical measurements, a lot of memory traffic is generated for each computation. The grammar for Zuker's algorithm consists of 4 tabulations with 15 productions and 13 scoring functions. The Zuker algorithm has a complexity of $O(n^4)$ but it is commonly accepted to bound some productions of very rare large structures to reduce its complexity to $O(n^3)$ with a large constant factor.

In Figure 12, we present results obtained for the Zuker algorithm. Once again, we show results for both CPU and GPU generated code, with and without backtracing. We compare our implementation to ViennaRNA [13], a highly optimized Zuker algorithm implementation written in C for CPU. ViennaRNA fares better than our implementation because it precomputes basepairs matching and stack-pairings for a sequence before launching the actual computation phase; our generated implementations do not benefit from such optimizations.

Compared to the simpler Nussinov algorithm, we can see that lookup tables introduce significant overhead to the computation. The overhead of computing the backtrace becomes negligible as a result. In Figure 12 the GPU is slower than CPU as the length of the sequences has not compensated for the transfer overhead yet.

Scalability To give an intuition of scalability for both the GPU and CPU versions, we benchmarked an extensive set of sequences ranging from 500 to 5000 elements in length (see Figure 13). The Nussinov algorithm scales better on GPU, offering speedups from $4\times$ to $40\times$ (respectively for 1000 and 5000 elements sequences). The Zuker lookup tables hamper GPU performance more significantly, as multiple random memory accesses are required for each scoring, thereby delivering at best $3.3\times$ speedup on the CPU for this algorithm.

6. Related Work

Tools for 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 [36]. They have since been incorporated into programming languages as libraries, like Parsec [18] in Haskell and the Scala parser combinator library [20]. These libraries focus on producing a single result. Koopman et al [17] use a continuation-based approach to eliminate intermediate list creation.

On the other hand are parser generators like Yacc [16], Antlr [22] and Happy [12]. While such tools are good in terms of performance, they do not easily support context-sensitivity, which is required in protocol parsing.

The staged parser combinator approach bridges the gap between both worlds in terms of features for a parser: ease of use, context-sensitivity, composability, specializability and performance.

Dynamic Programming/Memoization Parser combinators and memoization are common knowledge. Frost et al. introduce this technique [7, 8]. In libraries, techniques like packrat parsing [6] are supported. They are also known to support left recursion [37].

Our work on dynamic programming parsers is inspired by Algebraic Dynamic Programming [10]. The original implementation was a Haskell library, and later a external DSL known as Bellman's GAP was developed [30]. Bellman's GAP also includes analysis techniques for verifying some soundness properties, and optimizing memory consumption and running time through yield-size analysis [11]. ADPfusion is a more recent Haskell DSL that uses compiler optimizations to remove intermediate data structures (see below).

Bellman's GAP and ADPfusion both have support for computing backtraces. Bellman's GAP has an explicit product algebra construct, which the compiler uses to generate a backtrace for one of the algebras which is specified to be backtracing. ADPfusion supports a combine operator `<**` which combines two algebras, and a `backtrace` operator which can be used to apply an algebra on a matrix result-

ing from the optimal cost (forward) phase. Both these approaches reuse the matrix computed in the forward phase, and may recompute some results during the backtrace. Our approach on the other first creates an algebra-agnostic backtrace. We trade memory consumption (the paper trail) for an improved running time complexity. The cost matrix can be ignored during the backtracing phase: if the matrix is computed on the GPU, we only need to transfer the backtrace back to the CPU, and not the full resulting matrix.

Other language approaches are StagedDP [33] (programs are expressed using classic recurrences) and Dyna [5] (a logic-programming style language prevalent in the field of NLP/stochastic grammar parsing).

Metaprogramming and Compiler Technology To match performance of lower-level implementations, high-level languages require compiler technology. ADPfusion [14] uses Stream Fusion [4] to optimize away intermediate lists and generates tight efficient loops. Stream fusion depends on compiler optimizations performed further down for optimal performance. Our modular staging approach gives us more control on which optimizations we can activate.

Staged parser combinators are a specific instance of partial evaluation [9]. We make use of the well known technique of multi-stage programming [35]. Sperber et al. also use partial evaluation for optimizing LR parsers which are implemented as a functional-style library [31].

Cartey et al[2] propose an intermediate DSL for recurrence relations, which they analyze and generate GPU programs from. Their approach is more general as they try to infer a parallel schedule from recurrence relations; we leverage our domain-specific knowledge to force diagonal progress.

Parallelization Bellman's GAP supports a parallelization scheme for the GPU that is similar to ours, based on bottom-up evaluation and computation following the diagonal [32]. We additionally retrieve the backtrace on the GPU and transfer it to the CPU. The Nussinov and matrix multiplication problems have also been studied as pure GPU implementations [3, 38].

Much work on parallelizing dynamic programming has focused on the Smith-Waterman sequence alignment problem [19]. CudAlign [27, 28] matches sequences much larger than the GPU memory size, using a hybrid divide-and-conquer and dynamic programming approach. We have focused in this paper on folding problems rather than alignment problems.

7. Conclusion and Future Work

In this paper, we have demonstrated an approach for improving the performance of parser combinators, which removes the overhead of the composition. This is achieved by combining staging with abstractions that eliminate the creation of intermediate data structures. We not only retain user benefits (usability, modularity, composition), we also introduce domain- and architecture-specific optimizations for further benefits. Our approach is applicable to different classes of parsers: top-down recursive descent and bottom-up dynamic programming.

An interesting avenue for future work is the optimization of more complex parsers that perform interleaving. For example, we can interleave the parsing of a chunked HTTP response with parsing of the underlying payload. The naive way of parsing the payload is to first buffer all of it, and then run an underlying parser. If the size of this payload is big, we will hit memory bottlenecks. Instead, it is possible to parse the payload upstream. The idea is to compose not only over parsers, but also over `Input`. The input, instead of just being a position in a sequence, will contain additional information about the sequence, such as the current character/token, and the rest of the input. The input becomes a representation of a stream reader over a sequence: `type Input = Reader[T]`.

Such a representation implies that we can compose and abstract over stream readers. For example, this allows to differentiate between lexers and token readers, where for the latter, the rest of the input is computed by first running a white space parser over the sequence. In a similar fashion, chunked parsing can be implemented by feeding a chunked sequence to a reader which understands how to parse chunk sizes. From the view point of a parser combinators user, he can still write the payload parser and the chunking parser independently. In terms of optimizations, we can view stream readers as staged records: functions that compute the position, current character, and rest of the input can be staged away and inlined, using our approach.

In the big picture, we can see that parser combinators, beyond helping with parsing, are also a nice way of expressing streaming problems. Our techniques for optimizing composition can have interesting applications in other streaming-related applications as well.

Acknowledgments

We thank members of the LAMP team at EPFL for many fruitful discussions and insightful suggestions. Special thanks to Christian Höner zu Siederdissen for helping us understanding and setting up ADPfusion. This research was sponsored by ERC under the DOPPLER grant (587327). Part of this work was performed while the first author was an intern at Oracle Labs.

References

[1] The Apache HTTP server project. `http://httpd.apache.org/`.

[2] L. Cartey, R. Lyngsø, and O. de Moor. Synthesising graphics card programs from DSLs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 121–132, New York, NY, USA, 2012. ACM.

[3] D.-J. Chang, C. Kimmer, and M. Ouyang. Accelerating the Nussinov RNA folding algorithm with CUDA/GPU. In *Proceedings of the 10th IEEE International Symposium on Signal Processing and Information Technology*, ISSPIT '10, pages 120–125, Washington, DC, USA, 2010. IEEE Computer Society.

[4] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.

[5] J. Eisner, E. Goldlust, and N. A. Smith. Dyna: A declarative language for implementing dynamic programs. In *Proceedings of the ACL 2004 on Interactive Poster and Demonstration Sessions*, ACLdemo '04, Stroudsburg, PA, USA, 2004. Association for Computational Linguistics.

[6] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36–47, New York, NY, USA, 2002. ACM.

[7] R. Frost. Monadic memoization towards correctness-preserving reduction of search. In *Proceedings of the 16th Canadian Society for Computational Studies of Intelligence Conference on Advances in Artificial Intelligence*, AI '03, pages 66–80, Berlin, Heidelberg, 2003. Springer.

[8] R. A. Frost and B. Szydlowski. Memoizing purely functional top-down backtracking language processors. *Science of Computer Programming*, 27(3):263–288, November 1996.

[9] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[10] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, June 2004.

[11] R. Giegerich and G. Sauthoff. Yield grammar analysis in the Bellman's GAP compiler. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, LDTA '11, pages 7:1–7:8, New York, NY, USA, 2011. ACM.

[12] A. Gill and S. Marlow. Happy: The parser generator for Haskell. `http://www.haskell.org/happy/`, 2010.

[13] I. L. Hofacker. Vienna RNA secondary structure server. *Nucleic Acids Research*, 31(13):3429–3431, 2003.

[14] C. Höner zu Siederdissen. Sneaking around concatmap: efficient combinators for dynamic programming. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 215–226, New York, NY, USA, 2012. ACM.

[15] S. Janssen, C. Schudoma, G. Steger, and R. Giegerich. Lost in folding space? Comparing four variants of the thermodynamic model for RNA secondary structure prediction. *BMC Bioinformatics*, 12(429), 2011.

[16] S. C. Johnson. *YACC: Yet Another Compiler-compiler*, volume 32 of *Computing Science Technical Report*. Bell Laboratories, Murray Hill, NJ, 1975.

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

[18] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University, 2001.

[19] Y. Liu, A. Wirawan, and B. Schmidt. CUDASW++ 3.0: Accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14:117, 2013.

[20] A. Moors, F. Piessens, and M. Odersky. Parser combinators in Scala. CW Reports CW491, Department of Computer Science, K.U.Leuven, February 2008.

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

[22] T. J. Parr and R. W. Quong. ANTLR: A predicated-$LL(k)$ parser generator. *Softw., Pract. Exper.*, 25(7):789–810, 1995.

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

[24] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-Virtualized: Linguistic reuse for deep embeddings. *Higher Order and Symbolic Computation*, August-September:1–43, 2013.

[25] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, October 10–13 2010. ACM.

[26] 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.

[27] E. F. d. O. Sandes and A. C. M. A. de Melo. CUDAlign: Using GPU to accelerate the comparison of megabase genomic sequences. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 137–146, New York, NY, USA, 2010. ACM.

[28] E. F. d. O. Sandes and A. C. M. A. de Melo. Smith-Waterman alignment of huge sequences with GPU in linear space. In *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '11, pages 1199–1211, Washington, DC, USA, May 16–20 2011. IEEE Computer Society.

[29] E. F. d. O. Sandes and A. C. M. A. de Melo. Retrieving Smith-Waterman alignments with optimizations for megabase biological sequences using GPU. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1009–1021, 2013.

[30] G. Sauthoff. *Bellman's GAP: a 2nd generation language and system for algebraic dynamic programming*. PhD thesis, Bielefeld University, 2011.

[31] M. Sperber and P. Thiemann. The essence of LR parsing. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '95, pages 146–155, New York, NY, USA, 1995. ACM.

[32] P. Steffen, R. Giegerich, and M. Giraud. Gpu parallelization of algebraic dynamic programming. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part II*, PPAM '09, pages 290–299, Berlin, Heidelberg, 2010. Springer.

[33] K. Swadi, W. Taha, O. Kiselyov, and E. Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 160–169, New York, NY, USA, 2006. ACM.

[34] I. Sysoev. The nginx HTTP server. `http://nginx.org/`.

[35] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[36] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Tutorial Text*, volume 925 of *LNCS*, pages 24–52, Berlin, Heidelberg, May 24–30 1995. Springer.

[37] A. Warth, J. R. Douglass, and T. Millstein. Packrat parsers can support left recursion. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '08, pages 103–110, New York, NY, USA, 2008. ACM.

[38] C.-C. Wu, J.-Y. Ke, H. Lin, and W. chun Feng. Optimizing dynamic programming on graphics processing units via adaptive thread-level parallelism. In *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*, ICPADS '11, pages 96–103, Washington, DC, USA, December 7–9 2011. IEEE Computer Society.

[39] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 187–204, New York, NY, USA, 2013. ACM.

[40] S. Xiao and W. chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '10, pages 1–12, Washington, DC, USA, April 19–23 2010. IEEE Computer Society.