Specialising Parsers for Queries

THÈSE N^{O} 7249 (2016)

PRÉSENTÉE LE 11 NOVEMBRE 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Manohar JONNALAGEDDA

acceptée sur proposition du jury:

Prof. V. Kuncak, président du jury
Prof. M. Odersky, directeur de thèse
Prof. E. Visser, rapporteur
Dr A. Warth, rapporteur
Prof. C. Koch, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

Is it a bird? No!
Is it a plane? No!
It’s a PhD thesis!

Acknowledgements

During my adventure as a PhD student, I've been extremely fortunate to have worked in a great environment with inspiring people, from whom I have learned so much. I have also been lucky to have had great friends thanks to whom life became just that bit easier. For this I would like to give thanks.

To Martin, my advisor, for his passion for programming languages, his ability to federate an amazing team, and the independence and patience he gave for my research.

To Alex, Christoph, Eelco, my committee members, who took real time out to read this document, and provided great feedback and support, and helped improve the quality of this dissertation.

To Tiark, who got me interested in parsers back in my master studies days, and later in parsing and staging. Who also encouraged me to write up on results as I went, leading up to my academic blogging, and ensuing publications.

To Sandro, my dear friend and colleague, whose love for theory, thoroughness and clarity of thought helped me understand categorically complex papers, and inspired me to explain my research ever more clearly and accurately. To the countless hours spent with him discussing everything and anything, usually over beers, and having great fun doing so.

To Vojine, whose never-say-die attitude to life taught me to put that little bit extra effort whenever I felt the chips were down and reverse tough situations as a result. To the amazing unplanned adventures had with him during various trips.

To Heather, whose advice and suggestions helped me avoid many pitfalls and focus on the important stuff much better.

To Nico, my office mate, with whom much fun was had designing "Ce gars là", and so many more Scala and LAMP related jokes and memes.

To Seb "John-Bob" Doeraene, whose extraordinary discipline when developing software eventually left some traces in my work. With whom much fun was had discussing and creating musical interpretations of lab life.

To Denys, whose love for hacking, in-depth knowledge of quasiquotes and great insights on software development influenced my own design decisions and approach to presentations. To his happy go lucky approach to life which made us experiment with many a thing in life. Not caring too much about the consequences helped us achieve very positive ones.

To Don Macros whose deep knowledge of meta-programming and macros was instrumental in the final stages of this dissertation. Whose unique approach to software development and life influenced me in approaching life differently myself. Yes at all.

To Nada, whose appetite for reading made me discover some wonderful books, each of them unusual reads in their own right.
To Hubert, with whom many profound discussions about life, research and dance were had.
To all the other lampions, past and present, Dima, Felix, Olaf, Julien, Vladimir, Guillaume, Guillaume, Olivier, Alex, for the great atmosphere at work, and the even better atmosphere during non-work times.
To Georg, whose last-minute graph generation capabilities and general good vibes were essential to complete this document.
To Ingo "ShadeRide" Maier, Herr Doktor Haller, Eva Darulova, Michel Schinz, Sammy G, ever inspiring in their positivity, their approach to research, and their patience to read through and listen to various drafts and dry runs of mine.
To Thierry, Eric, Mathieu, Alexis, Jorge, who all graciously accepted to help me with various parts of my research, and whose ability to code I remain envious of.
To my other EDIC mates Alexandre "The Duke" Duc, Mr. White, Stefan, Goran, Niranjan, Etienne, with whom countless hours were spent at coffee breaks, football grounds, climbing gyms, tennis courts and of course, bars: productively procrastinating by having fun all the time.
To so many other friends, who have been a constant reminder of a world of opportunities in all walks of life.
To Tim I, Miss Tikaly, Duke Luke and Little Warrior, members of the Skankin' Society Sound System, singing and dancing to reggae vibes all night long, many nights long.
To Vinz Del, with whom it has been a lot of fun pursuing ambitious yet measurable goals every year.
To Indira and Suryanarayana, my grandparents, whose unconditional love and support is so relieving to have had over the years, a continent of distance notwithstanding.
To Sash, my brother and roommate, whose mere presence has been a joyful experience and calming influence. To his happy vibes which help realise that everything can be done while having fun. To his passport, which has occasionally accompanied me on trips.
And finally to Rama and Siva, my parents, who worked very hard to provide me with this platform. To their unwavering guidance and support thanks to which I never really had to worry too much. To my dad's adventurous spirit, which shows that crazy and amazing things can be done anytime, if only you want to. To my mom's exceptional cooking, which has so often kept my belly full, and most importantly, satisfied.

Lausanne, October 31, 2016 M. J.

Abstract

Many software systems consist of data processing components that analyse large datasets to gather information and learn from these. Often, only part of the data is relevant for analysis. Data processing systems contain an initial preprocessing step that filters out the unwanted information.

While efficient data analysis techniques and methodologies are accessible to non-expert programmers, data preprocessing seems to be forgotten, or worse, ignored. This despite real performance gains being possible by efficiently preprocessing data.

Implementations of the data preprocessing step traditionally have to trade modularity for performance: to achieve the former, one separates the parsing of raw data and filtering it, and leads to slow programs because of the creation of intermediate objects during execution. The efficient version is a low-level implementation that interleaves parsing and querying.

In this dissertation we demonstrate a principled and practical technique to convert the modular, maintainable program into its interleaved efficient counterpart.

Key to achieving this objective is the removal, or *deforestation*, of intermediate objects in a program execution. We first show that by encoding data types using Böhm-Berarducci encodings (often referred to as Church encodings), and combining these with partial evaluation for function composition we achieve deforestation. This allows us to implement optimisations themselves as libraries, with minimal dependence on an underlying optimising compiler.

Next we illustrate the applicability of this approach to parsing and preprocessing queries. The approach is general enough to cover top-down and bottom-up parsing techniques, and deforestation of pipelines of operations on lists and streams.

We finally present a set of transformation rules that for a parser on a nested data format and a query on the structure, produces a parser specialised for the query. As a result we preserve the modularity of writing parsers and queries separately while also minimising resource usage. These transformation rules combine deforested implementations of both libraries to yield an efficient, interleaved result.

Key words: Parsers combinators, partial evaluation, Church encodings, staging, deforestation, fusion, querying.

Résumé

Bien des logiciels comportent des composantes dédiées aux tâches de traitement et d'analyse de données, afin d'en tirer de l'information. Souvent, seulement une partie des données globales est pertinente pour l'étape d'analyse. Il est donc commun de pré-traiter ou pré-filter les données brutes afin d'en rejeter la partie inutile.

Il est de nos jours devenu facile d'accéder à des techniques avancées d'analyse d'information, même pour un programmeur non-expert. Ce n'est hélas pas le cas pour les tâches de pré-traitement, des réels gains en performance nonobstant.

Traditionnellement, l'implantation de l'étape de pré-traitement doit choisir entre modularité du programme et performance. La modularité est obtenue en séparant l'analyse, ou le *parsing* des données brutes, et le filtrage. Cette séparation résulte en une éxecution lente, due à la création d'objets intermédiares durant l'éxecution. La version efficace du programme est quant à elle écrite dans un style bas-niveau, entremêlant la requête et le parsing.

Dans cette thèse nous proposons une technique pratique et bien fondée qui convertit le programme maintenable et modulaire en son équivalent efficace et bas-niveau.

Pour atteindre cet objectif il est capital d'éliminer, ou de *déforester*, les objets intermédiaires. Nous montrons tout d'abord qu'il est possible d'obtenir la déforestation pour les structures de données de base en les représentant par leur *encodages de Böhm-Berarducci (communément dites de Church)*, et en les combinant avec l'évaluation partielle. Ceci nous permet de concevoir les optimisations elles-mêmes comme des bibliothèques, sans dépendre des capacités d'optimisation d'un compilateur sous-jacent.

Nous montrons ensuite que cette approche s'applique systématiquement au parsing et aux requêtes de pré-traitement. L'approche s'avère suffisemment générale pour couvrir des stratégies diverses de parsing, et la déforestation de séquences d'opérations sur des listes.

Nous présentons enfin un ensemble de règles de transformation qui, étant donné un parser pour un format de données imbriquées, et une requête sur cette structure, produit un nouveau parser qui est, quand à lui, spécialisé pour la requête en question. Nous préservons ainsi la modularité voulue, et minimisons aussi l'utilisation de ressources. Ces règles de transformation combinent les implantations déforestées des deux bibliothèques et produisent un résultat entremêlé et efficace.

Mots clefs : Combinateurs pour le parsing, évaluation partielle, programmation multi étapes, fusion, requêtes, déforestation, encodages de Church.

Contents

List of Figures

1 Introduction

We live in an era of information. We make a lot of decisions, be they political or personal, global or local, by analysing data gathered from around us. As the quantity of information increases, so does the need to process it efficiently. Hence the recent surge in the development of data processing systems.

Often we distinguish two main phases in such a system, data acquisition and data analysis. Consider, for example, a transportation network company that wishes to analyse driving patterns of its vehicles in a particular region of the world. Reasons for doing so may vary from optimising routes based on specific geographic circumstances to training autonomous vehicles for these circumstances. The analysis is based on data collected by the company, pertaining to, among other factors, geographic coordinates of a vehicle, its speed, its trajectory and direction. These logs obey a certain format specification, and due to their sheer size, are dumped in this raw format into files on servers owned by the company.

The core analysis phase will likely be a statistical (machine learning) algorithm. Once data relevant to the region in question has been filtered from the full dataset, it will be shipped to a cluster of machines so that the algorithm can benefit from parallelism to run faster. Nowadays, writing programs that seamlessly operate on large datasets in a distributed and parallel setting has become a lot easier for the general programmer, through the development of frameworks such as Spark [94]. These libraries abstract away the details of distributing and parallelizing a program: a developer only needs to focus on the core logic of his application, yet can obtain excellent performance as well as modular and declarative programs. Manipulating large sets of data is therefore on the cusp of being available to the masses.

An area where such abstractions are lacking, however, is the acquisition of relevant data from a full, raw, dataset. This is sometimes referred to as data preprocessing. Traditionally, this would be implemented in two steps:

- We first read all the raw data, and construct a representation for it that is easy to manipulate from a program. In other words, we *parse* data into a structure as per the specifications of the data format.

- We then write a program that, from this collection representing the full dataset, selects the subset that we are interested in. In other words, we *query* the dataset for a specific subset.

There are multiple advantages to implementing these steps independently. Most of all, there are good abstractions for writing parsers and queries, as both domains have been studied for a long time. We could use an external parser generator tool, or a parser combinator library for the first step. Likewise we can use an external query language such as SQL, or a querying library such as LINQ [54], for the second step. In all cases, not only is it easy to write these programs, it is also easy to read, maintain, or even change them. If the company wishes to apply analysis to another region in the world for instance, the change to the query is simple to make. Similarly, if the format of the vehicle data changes, it is also easy to change the parser.

Unfortunately this modularity comes with a heavy performance cost. By parsing the full dataset, we create a collection that is potentially far larger than the eventual data we are truly interested in. These intermediate structures take up a lot of memory resources. In memory-managed runtimes such as the JVM, this can directly lead to significant performance hit as well, because the virtual machine will spend a lot of time in tasks such as garbage collection. Moreover, during the querying step, we iterate over these intermediate structures again, causing more performance hits. Clearly it is better to discard unwanted data as early during the parsing step as possible.

The issue of intermediate objects manifests itself in subtler ways too. External tools for parsing and querying exhibit good performance in general. For our purposes though it is required that both steps have a common interface to communicate in. Hence the need for parsing and querying libraries in a common programming language. Alas, such libraries tend to run significantly slower than their external counterparts: the latter tend to be optimised for their specific tasks. Libraries, meanwhile, make use of many language features in order to appear intuitive and become easy to use. These features contribute to indirections and intermediate object creation during the execution of a program, which ultimately results in a performance hit.

Two possible solutions to the parsing and querying problem come to mind: manual interleaving and stream-based parsers.

Specialising by hand

The naive solution is to write a carefully hand-crafted low-level program that interleaves a given parser and a query. The advantage of this approach is that we can focus on removing inefficiencies in a very detailed manner, and obtain fast execution times. However, this solution does not scale: writing a program that interleaves parsing and querying is substantially more difficult than writing two separate modules. Also, changing either the query or the parser becomes significantly harder. More philosophically, it is rather sad that we must sacrifice program elegance for performance.

Event-based stream parsing

A hybrid approach is to use an event-based parser, also known as a push parser. An event-based parser allows us to specify actions to perform every time a sub-structure in the raw data is parsed: as soon as we detect subsets of data we do not need, we can immediately discard it. This approach is not without its issues though.

Crucially, writing queries here is not as declarative as with the independent step approach. A programmer has to manually intertwine his query as possibly many events in an event-based parser. He therefore needs to be familiar with the structure of the original data format and cannot operate in a parser-agnostic manner anymore.

Moreover, event-based parsers tend to be hand-specialised and optimised for specific data formats. The best known class of such parsers are SAX and STAX [57], which target the XML format. At the time of writing, XML has gone out of fashion, in favor of newer formats such as JSON or protocol buffers. Many domains specify their own unique data formats as well. Implementing hand-specialised parsers for these is once again tedious: there is a need for a more declarative way to write efficient parsers.

This dissertation

There is a need for a solution that reconciles declarative parsing and querying libraries with performance. Both parsers and queries must be written independently of each other as well. Beyond the domain of data preprocessing, such a solution would have implications on other domains such as network protocols and data marshalling/unmarshalling. Essentially, good programming abstractions *and* performance are needed any time raw data must traverse a pipeline in which only a subset of it is relevant.

In this dissertation we describe a solution that has it all: given a parser for a data format, and a query operating on the parsed data, we automatically specialise the query for the parser, that is we interleave both so that unwanted data is discarded as early as possible: the resulting program looks similar to a low-level hand-crafted one, and exhibits performance equivalent to the latter. Previous solutions to this problem focus on a specific data format, or use analysis and optimisation techniques that are complex. Our solution is unique in that it applied to any data format. Moreover, we considerably simplify the analysis and optimisation phases by viewing the problem in a programming language context. As such, we contribute to the state of the art.

Libraries for parsers (and queries) rely on creating complex parsers (or queries) by *composing* simpler ones. The crucial insight is to realise that this composition is *static*, even if the data being processed is *dynamic*. By evaluating the static composition parts of a parser (or query) before the dynamic parts, we eliminate intermediate objects that result in performance overhead, and obtain programs that run faster.

It is not only key to correctly identify which parts are static, but also to have an abstraction removal mechanism that is simple, so that it can be readily ported to any programming language. Dependence on an underlying compiler should be minimal. Essentially, we want *optimisations as libraries*.

Contributions and overview

To specialise parsers for queries, we must therefore combine different features: a conversion mechanism from an independent parser and a query to an interleaved parser, optimised, high-level libraries for parsers and queries. These libraries should be implemented so as to minimally rely on a specific compiler for efficient execution. To this effect we make the following contributions.

Optimisations as libraries

We present a methodology for developing libraries which by construction present good abstraction removal properties in Chapter 3.

The idea is to encode programming constructs and data types as functions, as proposed by the lambda calculus. These are known as Böhm-Berarducci encodings [9], and often referred to as Church encodings. By doing so, pipelines of operations on these reduce to chains of function compositions. Note that the functions themselves are static: they merely help in organising the flow of a program. The real computation takes place on underlying primitive datatypes.

In such a program, overhead is caused by intermediate structures created at the boundaries of functions, and by the indirection of having to call and apply these functions. The only abstraction removal mechanism now needed is the elimination of this static composition. This can be done by *partially evaluating* the composition away, or by inlining function bodies. The residual program then contains one function that is the optimised equivalent of its pre-transformed counterpart. Indeed, inlining is the main feature that a compiler needs to have in order to support our solution. In other words, any programming language that has a good inliner can design libraries as we describe them, and benefit from very good performance.

The above idea is by itself not new. Combining Church encodings with partial evaluation is implicitly understood as an elegant way to achieve performance [10]. Our key contribution is to distill the essential properties of Church-encoding based libraries, and of partial evaluators for these to the point where they are portable, and do not depend on a particular inliner or partial evaluation framework. Some aspects are not completely straightforward: these include the handling of join-points in a program flow, as well as recursion. We illustrate and solve these issues by presenting implementations in the Lightweight Modular Staging framework [69]. Similar results can be achieved using other frameworks, such as Scala Macros. This methodology provides the basis for implementing efficient parser and query libraries.

Short-cut fusion as a library

Queries can be viewed as a pipeline of operations over list-like data structures. Evaluated naively, they create intermediate structures that once again hinder performance. Following the optimisations as libraries methodology, we present a library for Church-encoded lists (or folds) in Chapter 4. Partially evaluating function composition converts a pipeline of operations into a tight loop over some initial data. This technique is commonly known as short-cut fusion. By representing lists in their Church-encoded form we get an optimisation as powerful as `foldr/build` fusion [29]: it does well on functions that operate on a single input list, i.e. that can be easily expressed as a fold, such as `map`, `flatMap` and `filter`. We extend the technique to the `partition` and `groupBy` functions.

List pipelines are also constructed using functions that work on multiple lists. Such pipelines are better optimised when using a stream (or unfold) based representation. In Chapter 5 we present a library for streams based on the above methodology. This library allows fusion of zip-like operations. It is slightly less powerful than the generic stream fusion algorithm [15] in the handling of the generic `flatMap` function. In addition to the optimisation as a library approach for streams, the main contribution here is to show that a variant of the latter function, known as name-capturing `flatMap` [14, 19] can also be optimised in this framework.

Staged parser combinators

Parser combinators can also be implemented using the optimisation as libraries approach (Chapter 6). A key advantage of combinators over parser generators is that the latter are embedded libraries: it becomes easier to specify actions to perform on parsed data. Moreover, combinator libraries also facilitate specification of context-sensitive parsing, where the manner in which data should be parsed depends on previous input.

This approach not only applies to top-down recursive-descent parsing, but also to bottom-up, CYK style parsing (Chapter 7). It is particularly useful when handling ambiguous grammars where we seek an optimal result: such problems correspond to dynamic programming instances. In addition to removing intermediate structures, we take advantage of the inherent parallelism present in these, and generate code that runs on the GPU.

In both cases, we show that these libraries exhibit good performance.

Specialising parsers for queries

The above give us efficient implementations for parsers as well as pipeline-like query operations. We combine both to achieve our final goal in Chapter 8.

We describe a transformation from a tree representing a parser for a data format to a specialised parser for a given query. We treat queries that are a combination of the selection, projection and aggregation operators on a nested relational algebra. The transformation includes the conversion of skipped parsers to recognisers.

A key insight to make the transformation possible is to realise that repetition parsers, i.e. parsers that repeatedly look for the same structure, can also be treated as Church-encoded lists. By composing over repetition parsers, we therefore systematically push query operations directly to the point where a single element is parsed: it can be processed or discarded very early in the pipeline.

We show that our transform is semantics preserving: if a non-transformed parser succeeds on an input, a transformed parser succeeds with the same result. The correctness argument relies on correct implementations of recognisers for base parsers. We also discuss optimisation properties of the transformation.

This transformation further improves on the already partially evaluated parsing and querying libraries. Indeed, discarding data as early as possible considerably reduces strain on memory and garbage-collection tasks a virtual machine would have to otherwise perform.

Implementation

The libraries presented in this dissertation are implemented in Scala. To properly showcase partial evaluation capabilities we use the Lightweight Modular Staging framework (Section 2.3.1). We assume that the reader has some basic notions of the language, and provide context and details as and when necessary.

2 Background

In this chapter we review Church encodings and partial evaluation frameworks. We introduce two Scala frameworks for partial evaluation, namely Lightweight Modular Staging and Scala Macros. The main reason to use these is to decouple our optimisations from anything an underlying compiler may do. As a result, we are able to demonstrate the portability of our technique.

Böhm-Berarducci (Church) encodings

The lambda calculus is a formal system used to reason about computation/programming languages. It consists of two operations, function abstraction and function application. Using just these two, we can encode any programming language construct, such as booleans, numerals, conditional expressions, products, sums and lists. These encodings are known as *Church encodings*.

Adding types and polymorphism to the lambda calculus gives us System F. Equivalent encodings exist in this calculus as well: the Böhm-Berarducci encodings [9]. For ease of presentation (and as the former term is more prevalently used), we will refer to them as Church encodings. In this section we describe encodings for products, sums, and lists, following the presentation by Pierce [65]. We elide booleans and numerals, as we use their primitive counterparts instead.

Products

In the untyped lambda calculus, pairs are encoded using the following functions:

```
pair = λl. λr. λb. b (f s)
fst  = λp. p (λl. λr. l)
snd  = λp. p (λl. λr. r)
```

That is, a pair is a function that takes two "elements" `l`, `r`, a "constructor" `b` and applies the constructor to the elements. To retrieve the first/second element, we pass the function that returns the left/right element respectively. In System F, the type of a Church pair becomes:

```
Pair A B = ∀X. (A -> B -> X) -> X
```

where `A, B` represent the types of the underlying elements. The type `X` represents an eventual representation of the data type, which is only decided when a function of type `A -> B -> X` is passed. This function is also referred to as a continuation. Therefore in the rest of this dissertation we will also use the term *CPS-encoding* to refer to Church encodings. This encoding generalises to products of any arity.

We redefine `pair, fst, snd` to operate with `Pair A B`. Their definitions follow their untyped counterparts above. In addition, products are functors, which means we can define a `map` function over them:

```
pair : ∀A. ∀B. A -> B -> Pair A B
pair = λX. λA. λB.
         λa: A. λb: B. λk: A -> B -> X. k (a b)

fst  : ∀A. ∀B. Pair A B -> A
fst  = λA. λB.
         λp: Pair A B. p (λ a: A. λ b: B. a)

snd  : ∀A. ∀B. Pair A B -> B
snd  = λA. λB.
         λp: Pair A B. p (λ a: A. λ b: B. b)

map  : ∀A. ∀B. ∀C. ∀D. (A -> C) -> (B -> D) -> Pair A B -> Pair C D
map  = λX. λA. λB. λC. λD.
         λf: A -> C. λg: B -> D.
           λp: Pair A B.
             λc: C. λd: D. λk: C -> D -> X. p (λ a: A. λ b: B. k (f(a) g(b)))
```

Figure 2.1 – Encoding pairs in the lambda calculus

Note how the `map` function does not extract the first and second elements of the pair `p`, but instead forwards the mapping by applying a continuation to `p`.

A Scala implementation. For completeness sake, and especially for reference further on, we provide a Scala implementation of Church pairs in Figure 2.2. A Church pair is represented by the abstract `CPSPair` class, parametric in `A, B`. Scala does not have a universal quantifier, unlike System F. We must therefore encode it. Here we achieve this by making the `apply` function polymorphic in `X`. In the rest of the dissertation, for simplicity sake, we will often abuse notation and use the `forall` keyword when referring to Church encodings: Church pairs therefore will be represented using the pseudo type alias:

```
type CPSPair[A, B] = forall X. ((A, B) => X) => X
```

```
abstract class CPSPair[A, B] { self =>
  def apply[X](k: (A, B) => X): X

  def fst = self.apply((a, b) => a)
  def snd = self.apply((a, b) => b)

  def map[C, D](f: A => C, g: B => D) = new CPSPair[C, D] {
    def apply[X](k: (C, D) => X) = self.apply((a, b) => k(f(a), g(b))
  }

  def toPair: (A, B) = self.apply((a, b) => (a, b))
}

// companion object
object CPSPair {
  def mkPair[A, B](a: A, b: B) = new CPSPair[A, B] {
    def apply[X](k: (A, B) => X): X = k(a, b)
  }
}
```

Figure 2.2 – A Scala implementation of Church pairs

The self type (`self`) in Figure 2.2 is a reference to the outer instance: we make use of this reference to construct new pairs, for instance in the `map` function. We can also construct a classic pair with the `toPair` function.

Sums

Sums express the choice between different variants. The simplest sums are a) the `Maybe` of `Option` data type, which expresses the choice between some value or nothing, and b) the `Either` data type, which expresses the choice between a left and a right value. The `Either` data type are encoded as follows:

```
left  = λl. λr. λa. l a
right = λl. λr. λb. r a
```

The `left` function takes two functions `l, r` and applies `l` to the argument `a`. Analogously, the `right` function applies `r` to the final argument. The functions `l, r` represent the choice between a left and right side. In System F, we get the following encoding:

```
Either A B = ∀X. (A -> X) -> (B -> X) -> X
```

As with products above, we define the sum constructors and the `map` function, shown in Figure 2.3. As expected, the `map` function propagates `f` and `g` correctly to the appropriate variants.

```
left : ∀A. ∀B. A -> Either A B
left = λX. λA. λB.
         λa: A.
           λl: A -> X. λr: B -> X. l(a)

right : ∀A. ∀B. A -> Either A B
right = λX. λA. λB.
         λb: B.
           λl: A -> X. λr: B -> X. r(b)

map  : ∀A. ∀B. ∀C. ∀D. (A -> C) -> (B -> D) -> Either A B -> Either C D
map  = λX. λA. λB. λC. λD.
         λf: A -> C. λg: B -> D.
           λe: Either A B.
             λl: C -> X. λr: D -> X.
               e (λ a: A. l(f(a))) (λ b: B. r(g(b)))
```

Figure 2.3 – Encoding sums in the lambda calculus

```scala
abstract class CPSEither[A, B] { self =>
  def apply[X](l: A => X, r: B => X): X

  def map[C, D](f: A => C, g: B => D) = new CPSEither[C, D] {
    def apply[X](l: C => X, r: D => X) =
      self.apply(a => l(f(a)), b => r(g(b)))
  }
  def toEither: Either[A, B] = self.apply(a => Left(a), b => Right(b))
}
//Companion object
object CPSEither {
  def mkLeft[A, B](a: A) = new CPSEither[A, B] {
    def apply[X](l: A => X, r: B => X) = l(a)
  }
  def mkRight[A, B](b: B) = new CPSEither[A, B] {
    def apply[X](l: A => X, r: B => X) = r(b)
  }
}
```

Figure 2.4 – A Scala implementation of Church `Either`

A Scala implementation. Figure 2.4 gives a Scala implementation of `CPSEither`. We use a translation that is similar to the one for `CPSPair`, and is fairly straightforward.

Lists

Finally, lists are another basic data type in functional programming, and are used to represent an arbitrary sequence of elements. A list is a recursive structure, which is either the empty list

```
nil : ∀A. CPSList A
nil = λX. λA.
        λn: X. λc: (A -> X) -> X. n

cons : ∀A. A -> CPSList A -> CPSList A
cons = λX. λA.
         λhd: A. λtl: CPSList A.
           λn: X. λc: (A -> X) -> X. c (hd (tl [X] n c))

map : ∀A. ∀B. (A -> B) -> CPSList A -> CPSList B
map = λX. λA. λB.
        λf: A -> B. λls: CPSList A.
          λn: X. λc: (B -> X) -> X. ls [X] (n (λa: A, λacc: X. c (f (elem) acc)))
```

Figure 2.5 – Encoding lists in the lambda calculus, based on `foldRight`

```scala
abstract class CPSList[A] { self =>
  def apply[X](z: X, combine: (A, X) => X): X

  def map[B](f: A => B) = new CPSList[C, D] {
    def apply[X](z: X, combine: (B, X) => X) =
      self.apply(z, (elem, acc) => combine(f(elem), acc))
  }
}
//Companion object
object CPSList {
  def nil[A] = new CPSList[A] {
    def apply[X](z: X, combine: (A, X) => X): X = z
  }
  def cons[A](hd: A, tl: CPSList[A]) = new CPSList[A] {
    def apply[X](z: X, combine: (A, X) => X): X =
      combine(hd, tl.apply(z, combine))
  }
  def foldRight[A](ls: List[A]): CPSList[A] = new CPSList[A] {
    def apply[X](z: X, combine: (A, X) => X): X = ls match {
      case Nil => z
      case a :: as => combine(a, foldRight(as))
    }
  }
}
```

Figure 2.6 – A Scala implementation of Church lists

(`Nil`) or an element prepended to a list. In System F, the Church encoding for a list is given by the following type:

```
CPSList A = ∀X. X -> ((A -> X) -> X) -> X
```

We define the basic constructs `nil`, `cons` and `map` in Figure 2.5: The `nil` function returns a base element `n` provided by the continuation whereas the `cons` function applies the accumulator `c` to the element `hd`. The `map` function applies the function `f` to every element in the original list.

A Scala implementation. Figure 2.6 gives a Scala implementation of `CPSList`. We also define a function that converts a classic `List` to a `CPSList`: this function is commonly known as `foldRight`. Indeed, a Church list corresponds to the `foldRight` function. We present a much more detailed library for `CPSList` in Chapter 4.

Of Church encodings and initial algebras

Products, sums and lists are but a few data structures that have Church encodings. Church encodings exist for any algebraic data type. This is because these data types correspond to least fixed points (or initial algebras) on functors implicitly defined by them [55]. The `apply` methods defined above correspond to folds, that is, they are *unique* morphisms from the initial algebra to any other algebra.

Partial evaluation and multi-stage programming

Partial evaluation [24] is a technique used primarily to perform program optimisation. Suppose we want to run a program `p` on inputs `in1` and `in2`:

```
val res = p(in1, in2)
```

If we know `in1` statically (but `in2` remains dynamic), we can evaluate `p` in two steps:

- We first evaluate `p` by replacing all occurrences of `in1` in its body by the static value. This yields a *residual* program `p_in1` that is *specialised* for `in1`.
- We then evaluate `p_in1` on the dynamic input `in2`.

The result of either is the same (i.e. `res`). The difference is that since `p_in1` is specialised, it can potentially run faster than the original program `p`. We also say that `p_in1` is obtained by *partially evaluating* `p` over `in1`.

Function composition and application

Consider the following functions:

```
def f(i: Int) = i * 2
def g(i: Int, j: Int) = i + 3 + j
def composed(i: Int, j: Int) = g(f(i), j)
```

In the above, we assume `i, j` to be dynamic inputs. Yet the operations on these are statically available, i.e. the bodies of the functions are statically known. Therefore we can partially evaluate `g` and `f` in `composed`, yielding a function where the bodies of `g` and `f` are inlined:

```
def composed_g_f(i: Int, j: Int) = i * 2 + 3 + j
```

Effectively, partial evaluation of function composition and application corresponds to *inlining* the respective function bodies.

Multi-stage programming

A partial evaluator detects which parts of a program are static, and which dynamic, by performing *binding-time analysis*. This phase is often aided by the programmer, who annotates parts of the program as static, or dynamic.

A closely related concept is multi-stage programming (MSP) or staging [85], a form of generative programming. In a multi-staged program, one explicitly specifies which parts of the program are to be evaluated at the current stage, and which parts should be evaluated at a later stage. Running a staged program generates a new program, where current stage computations have been evaluated away. MSP can therefore be used to achieve controlled partial evaluation.

Multi-stage programming in practice

In practice, staging can be implemented in many different ways. We look at two possibilities for Scala in this section, Lightweight Modular Staging (LMS) and Scala Macros.

Lightweight Modular Staging

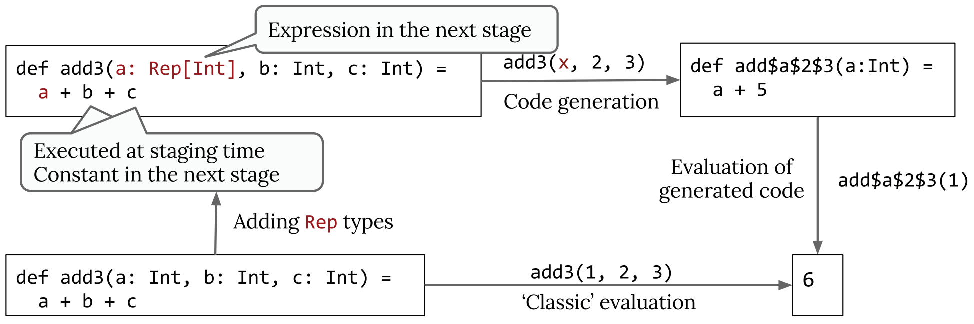


Figure 2.7 – Staging in LMS

Lightweight Modular Stating (LMS) is a staging/runtime code generation framework written in Scala. The evaluation of expressions is controlled through the use of a special abstract type `Rep[T]`:

- an expression of type `T` evaluates to a constant of type `T` in the generated code,
- an expression of type `Rep[T]` generates code for an expression of type `T`.

Essentially, binding-time analysis is guided by the type system itself. Figure 2.7 illustrates this principle. Starting from a program as in the bottom-left corner, a programmer adds `Rep` types, as in the top-left corner. The LMS framework will run this program, yielding later-stage code (top-right corner). Only when this code is executed do we get the final result of the program. Note that when we compose expressions of type `Rep[T]`, we *compose code generators*.

Staged functions. We saw above (Section 2.2.1) that partially evaluating function composition and application amounts to inlining. A static function in LMS is given by the type `Rep[T] => Rep[U]`. Therefore, in order to achieve inlining for the `composed` method above, we add `Rep` types as follows:

```
def f(i: Rep[Int]) = i * 2
def g(i: Rep[Int], j: Rep[Int]) = i + 3 + j
def composed(i: Rep[Int], j: Rep[Int]) = g(f(i), j)
```

Sometimes it is desirable to stage a function, for example with recursive functions. Such a function has the type `Rep[T => U]`.

The LMS infrastructure. Note in the example above that the multiplication and addition operators are defined on values of type `Rep[Int]`. Similarly, it is possible to use boolean operators on values of type `Rep[Boolean]`. LMS takes a closed world view: we progressively increase the expressiveness of staged values by combining separate modules which define operations on them. By convention, these modules are traits suffixed by `-Ops`. In order to program with staged conditional expressions, boolean and arithmetic expressions, we would mix these modules in:

```
trait MyProgramOps extends Base with BooleanOps with ArithOps with IfThenElse
```

The `Base` trait is the root of the hierarchy, and defines the `Rep` type constructor. The core LMS library contains modules for many other basic data types and structures, such as arrays and lists. These can be used out of the box.

A value of type `Rep[T]` is an *abstract* representation. To this abstract representation we must map a concrete representation. In LMS, to each instance of `Rep[T]` corresponds a concrete expression datatype `Exp[T]`. For a given staged program, the collection of these `Exp` datatypes forms its *LMS intermediate representation* (IR), and a particular instance is a node in this IR. To define a domain-specific IR node we create a definition `Def[T]`, which is a subtype of `Exp[T]`. By convention these nodes are defined in a trait suffixed by `-Exp`:

```
trait MyProgramOpsExp extends MyProgramOps with BaseExp
  with BooleanOpsExp with ArithOpsExp with IfThenElseExp
```

The intermediate representation is useful for performing domain-specific optimisations and rewrites. For instance, a conditional expression where the condition is known to be constant may be replaced by one of its branches. Once such optimisations are performed, the IR is used for code generation.

It is useful to think of `Rep` and `Exp` as two distinct worlds, *interface* and *implementation*: the interface is visible to a DSL user, while the implementation is where domain-specific nodes are created and optimised. For instance, if we wanted to add the modulo operation for integers, the interface would contain the following abstract method:

```
def mod(dividend: Rep[Int], divisor: Rep[Int]): Rep[Int]
```

In the `Exp` world we would define an IR node represeting a modulo operation, and implement `mod` such that it creates this node:

```
case class Mod(dividend: Exp[Int], divisor: Exp[Int]) extends Def[Int]
def mod(dividend: Exp[Int], divisor: Exp[Int]): Exp[Int] = Mod(dividend, divisor)
```

In addition to the programming interfaces for basic constructs mentioned above, the core LMS library also defines intermediate nodes for these. These building blocks can be used to build more complex code generators [71].

Language virtualisation For maximal developer productivity, staged programs should look as close as possible to their non-staged counterparts. Adding the extra `Rep` wrapper may be an acceptable trade-off, but other basic constructs in the language should remain unchanged.

In LMS, it is possible to write conditional expressions where the condition is a `Rep[Boolean]` (instead of `Boolean`). These facilities extend to other basic Scala constructs, such as mutable variable creation and assignment, pattern matching, while loops. This is possible by *virtualising* these constructs, which is accomplished by the Scala Virtualized compiler [67]. Here, every basic construct is treated as a method call. Domain-specific behaviour is implemented for these constructs by simply overriding implementations for corresponding methods. At the time of writing, there is also an alternate macro-based virtualisation implementation.

Scala Macros

Scala Macros are a compile time meta-programming tool in Scala. They allow to analyse and transform code blocks at compile time. Scala Macros come in many flavours, each of them allowing to accomplish particular kinds of tasks [8]. In this dissertation we only present **`def`** macros: they are sufficient for partial evaluation purposes. The implementations we describe follow those of Scala 2.12.x, and may change for future Scala versions.

A **`def`** macro looks very much like a classic Scala function. It is defined as follows:

```
def myMacro(x: => T): U = macro myMacro_impl
```

We declare myMacro to be a macro that takes trees of type T and returns trees of type U. The distinguishing feature from a normal Scala function is the usage of the macro keyword. A macro is expanded at compile time, as opposed to a function, which is applied at runtime. The implementation of the macro is delegated to the myMacro_impl function (Figure 2.8). This function's parameters are *representations* of expressions of type T.

A **def** macro receives well-typed trees: ill-typed Scala programs cannot be operated on in such a macro. Such trees also have the type c.Tree, instead of being wrapped as c.Expr[T]: since we know these trees are well-typed, we can recover the type via the Scala reflection libraries. Using the reflection libraries, we can also manipulate these expressions, in a manner analogous to the manipulation of Exp nodes in LMS. Quasiquotes act as a more programmer friendly frontend in this respect [76]. The myMacro_impl function must return expressions of type U.

A **def** macro can inspect trees defined inside its scope, but not beyond: as such it also takes a closed-world view.

```
import scala.reflect.macros.Context
import scala.language.experimental.macros
trait MyMacro {

  val c: Context
  import c.universe._

  def myMacro_impl[T, U](x: c.Expr[T]) : c.Expr[U] = {
    // implementation
  }
}
```

Figure 2.8 – The implementation of a macro in Scala

Note the resemblance between c.Expr[T] and Rep[T] from LMS. Programming with either can indeed be thought of as composing code generators (macros allow for more through the reflection library): the same principles present when programming with Rep types apply directly to programming with expressions in macros. A user of a staged library would however benefit from a macro implementation: his program is wrapped in as myMacro function, and does not need to contain any extra boilerplate notation that Rep or c.Expr bring. Lifting user code into Rep form remains the responsibility of the library developer. We illustrate this lifting process for parser combinators in Section 8.4.

In this dissertation, we seek to lay out the principles behind staged libraries, and therefore will explicit the use of code generators. Since Rep or c.Expr can be used in an equivalent way for our purposes, we will use the former unless explicitly mentioned otherwise.

3 Optimisations as libraries

In this chapter we develop a methodology for developing efficient high-level libraries from first principles. These guidelines will be instrumental in implementing subsequent libraries for lists and parsers.

The key insight explored by this dissertation is to realise that Church encodings and staging can be combined for great benefit, i.e. to achieve deforestation of arbitrary data types *by construction*. Deforestation [86], or fusion, is a technique whereby intermediate objects that typically appear in a functional program are eliminated. The resulting program exhibits better performance as a result. In a nutshell:

1. By Church encoding data structures, we can reduce fusion of arbitrary data types to fusion of functions.
2. Staging allows us to partially evaluate function composition, thus effectively achieving fusion of functions.

In this chapter we illustrate this insight for products and sums. This is essential in order to optimise more complex structures. Deforestation of lists using these principles is covered in detail in Chapter 4. As mentioned previously, we use `Rep` types from here on to distinguish dynamic and static values (see Section 2.3.1 for more details).

Staged products

Recall the CPS encoding of pairs seen in Section 2.1.1, represented here as a (pseudo) type alias:

```
type CPSPair[A, B] = forall X. ((A, B) => X) => X
```

We systematically add `Rep` wrappers to argument and return types, and get the following type alias:

```
type CPSPair[A, B] = forall X. ((Rep[A], Rep[B]) => Rep[X]) => Rep[X]
```

A staged Church pair is an *unstaged* function that takes a continuation (itself an unstaged function) and returns an eventual value of type `X`. This systematic introduction of `Rep` types can be performed for the full API presented in Figure 2.2, to yield a staged implementation, given in Figure 3.1.

```
trait CPSPairOps extends Base with IfThenElse with PairOps {

  abstract class CPSPair[A, B] { self =>
    def apply[X](k: (Rep[A], Rep[B]) => Rep[X]): Rep[X]

    def fst: Rep[A] = self.apply((a, b) => a)
    def snd: Rep[B] = self.apply((a, b) => b)

    def map[C, D](f: Rep[A] => Rep[C], g: Rep[B] => Rep[D]) = new CPSPair[C, D] {
      def apply[X](k: (Rep[C], Rep[D]) => Rep[X]) =
        self.apply((a, b) => k(f(a), g(b))
    }

    def toPair: Rep[(A, B)] = self.apply((a, b) => make_tuple2(a, b))
  }

  // companion object
  object CPSPair {
    def mkPair[A, B](a: Rep[A], b: Rep[B]) = new CPSPair[A, B] {
      def apply[X](k: (Rep[A], Rep[B]) => Rep[X]): Rep[X] = k(a, b)
    }
    ...
  }
}
```

Figure 3.1 – An implementation of staged Church pairs in LMS

The enclosing trait `CPSPairOps` contains definitions for staged `CPSPair`. It mixes in modules for staged pairs, which are used by the `toPair` function, as well as conditional expressions, which we describe in more detail below (Section 3.2.1).

With this implementation of Church pairs we can now chain pipelines of operations where no intermediate objects are created, as shown in Figure 3.2. In the first half, `x` and `y` are symbolic expressions of type `Rep[Int]`. In the second half we see the partially evaluated code generated by LMS. Note that no pair structure is created anywhere.

Staged sums

Staging Church sums follows the same procedure as for products. By adding `Rep` types, we get the following:

```
// these operations on staged pairs
mkPair(x, y).map(a => a * 3, b => b + 7).apply((a, b) => a + b)

//yield the following partially evaluated code
===>
val x: Int = ...; val y: Int = ...
val x_2: Int = x * 3
val y_2: Int = y + 7
x_2 + y_2
```

Figure 3.2 – An example of partial evaluation for `CPSPair`

```
trait CPSEitherOps extends Base with IfThenElse with EitherOps {

  abstract class CPSEither[A, B] { self =>

    def apply[X](l: Rep[A] => Rep[X], r: Rep[B] => Rep[X]): Rep[X]

    def map[C, D](f: Rep[A] => Rep[C], g: Rep[B] => Rep[D]) = new CPSEither[C, D] {
      def apply[X](l: Rep[C] => Rep[X], r: Rep[D] => Rep[X]) =
        self.apply(a => l(f(a)), b => r(g(b)))
    }

    def toEither: Rep[Either[A, B]] = self.apply(a => Left(a), b => Right(b))
  }

  //Companion object
  object CPSEither {
    def mkLeft[A, B](a: Rep[A]) = new CPSEither[A, B] {
      def apply[X](l: Rep[A] => Rep[X], r: Rep[B] => Rep[X]) = l(a)
    }

    def mkRight[A, B](b: Rep[B]) = new CPSEither[A, B] {
      def apply[X](l: Rep[A] => Rep[X], r: Rep[B] => Rep[X]) = r(b)
    }
  }
}
```

Figure 3.3 – An implementation staged Church `Either` in LMS

```
type CPSEither[A, B, X] = (Rep[A] => Rep[X], Rep[B] => Rep[X]) => Rep[X]
```

Figure 3.3 provides the full staged implementation.Once again, partial evaluation removes intermediate structures. So the following expression:

```
mkLeft(x).map(a => a * 3, b => b + 7).apply((a => a + 5), (b => b))
```

generates the following code:

```
val x: Int = ...
val x_2: Int = x * 3
x_2 + 5
```

Note how code for the right hand side is never generated as it is not used.

Conditional expressions

Consider the following code that creates Church pair based on a dynamic condition:

```
val cond: Rep[Boolean] = ...
val p = if (cond) mkPair(x1, y1) else mkPair(x2, y2)
p.map(a => a * 3, b + 5).apply((a, b) => a + b)
```

We must first specialise conditional expressions for `CPSPair`, since by default, conditional expressions expect values of type `Boolean` (see Section 2.3.1). This is done by implementing the virtualised `__ifThenElse` method. A naive implementation would create a new instance of `CPSPair`, where the continuation is pushed to both branches of the condition:

```
trait CPSPairs ... {
  ...
  def __ifThenElse[A, B](
      cond: Rep[Boolean],
      thenp: => CPSPair[A, B],
      elsep: => CPSPair[A, B]) = new CPSPair[A, B] {

    def apply[X](k: (Rep[A], Rep[B]) => Rep[X]): Rep[X] = {
      if (cond) thenp.apply(k)
      else      elsep.apply(k)
    }
  }
}
```

Figure 3.4 – A naive virtualisation of conditional expressions for Church pairs in LMS

Unfortunately this implementation results in duplicating code. Following the execution trace of the partial evaluator shows why:

```
p.map(a => a * 3, b + 5).apply((a, b) => a + b)
-> if (cond) mkPair(x1, y1).map(a => a * 3, b + 5).apply((a, b) => a + b)
   else      mkPair(x2, y2).map(a => a * 3, b + 5).apply((a, b) => a + b)
```

The generated code reflects this duplication (Figure 3.5). Indeed, conditional expressions act as split points in a program. Naively choosing to inline continuations in each branch will lead to exponential code generation as the number of nested conditional expressions increases.

While in some cases, it might pay off to duplicate code (if we seek to run both branches in parallel for instance), a safe solution is to introduce an explicit join point at the end of a

```
val cond: Boolean = ...
if (cond) {
  val x_3: Int = x1 * 3
  val y_3: Int = y1 + 7
  x_3 + y_3
} else {
  val x_3: Int = x2 * 3
  val y_3: Int = y2 + 7
  x_3 + y_3
}
```

Figure 3.5 – Code resulting from the use of naive conditional expressions for Church pairs

conditional expression. Since we want to avoid allocating intermediate objects, a solution is to introduce intermediate mutable variables that are assigned to in each branch. Figure 3.6 shows an implementation for `CPSPair`.

```
trait CPSPairOps extends ... with Variables {
  ...
  def __ifThenElse[A, B](
      cond: Rep[Boolean],
      thenp: => CPSPair[A, B],
      elsep: => CPSPair[A, B]) = new CPSPair[A, B] {

    def apply[X](k: (Rep[A], Rep[B]) => Rep[X]): Rep[X] = {
      var aTmp: Rep[A] = zeroVal[A]
      var bTmp: Rep[B] = zeroVal[B]

      val assignK = (a: Rep[A], b: Rep[B]) => { aTmp = a; bTmp = b }
      if (cond) thenp.apply(assignK) else elsep.apply(assignK)

      k(aTmp, bTmp)
    }
  }
}
```

Figure 3.6 – Virtualising conditional expressions for Church pairs using local state

Note that we now mix in the `Variables` component, which allow us to program with staged mutable variables. The `zeroVal` method provides default values for various types. In each conditional branch, we pass continuations that simply assign the new values to previously declared temporary variables. The final continuation is applied to these temporary variables. For the above example with `CPSPair`, the generated code looks as one would expect (Figure 3.7). For `CPSEither`, we use an extra boolean variable `isLeft` to help us decide which variant in the sum we are dealing with. For sums with more variants, a more complex flag (an integer for instance) will have to be used instead.

```
val cond: Boolean = ...
var xTmp = 0; var yTmp = 0

if (cond) { xTmp = x1; yTmp = y1 }
else      { xTmp = x2; yTmp = y2 }

val x_3: Int = xTmp * 3
val y_3: Int = yTmp + 7
x_3 + y_3
```

Figure 3.7 – Code generated for conditional expressions after introduction of a join point

This implementation is safe: it makes use of state in a controlled manner in a very localised scope. It corresponds to wrapping the conditional expression in a state monad [81].

Nested Church encodings

The attentive reader will have noticed that the above solution is not quite general yet. If `A` or `B` are CPS-encoded structures themselves, our current library does not allow us to create such values, as the `mkPair` constructor expects values of type `Rep[A]` and `Rep[B]`. The issue also carries to `zeroVal`.

A naive solution would be to maintain specialised Church encodings for nested structures (for example `CPSPairPair[A, B, C]`). Naturally this does not scale, as maintaining every possible combination of nesting will result in exponentially large libraries.

A more elegant solution is possible. Note that we always precisely know the points where and how Church-encoded structures are created: by calling the constructors in the library interface, by using higher-order library functions, and in branches of conditional expressions. From an intermediate representation point of view, these are all constant, static nodes. Therefore it is sufficient to create constant wrappers for `CPSPair[A, B]` in order to lift them into the `Exp` world, and simply unwrap them as and when needed.

Recall from above (Section 2.3.1) that IR nodes (`Exp[T]`) in LMS are either constants `Const[T]` or definitions `Def[T]`. For our case it is sufficient to wrap a value `p` of type `CPSPair[A, B]` into a constant by using the `unit` lifter. For reasons tied to the way Scala resolves implicits, however, it is more practical to create a deliberate wrapper definition, which we do as follows:

```
case class PairWrapper[A, B](p: CPSPair[A, B]) extends Def[CPSPair[A, B]]
```

The rest of the implementation for `PairWrapper` is straightforward. Every single method in the API of `PairWrapper` has a corresponding method that simply forwards the implementation down, as shown in Figure 3.8. We refer the interested reader to [71] for more details on the LMS IR.

```
trait CPSPairExp extends CPSPair with ... {
  /** The wrapper acting as Rep[CPSPair[A]] */
  case class PairWrapper[A, B](p: CPSPair[A, B]) extends Def[CPSPair[A, B]]

  def mkPair[A, B](a: Rep[A], b: Rep[B]): Exp[CPSPair[A, B]]
    = PairWrapper(PairStruct(a, b))

  def paircps_map[A, B, C, D](
      pair: Rep[CPSPair[A, B]],
      f: Rep[A] => Rep[C],
      g: Rep[B] => Rep[D]): Rep[CPSPair[C, D]] = pair match {
    case Def(PairWrapper(pair)) => PairWrapper(pair.map(f, g))
  }

  def paircps_apply[A, B, X](
      pair: Rep[CPSPair[A, B]],
      k: (Rep[A], Rep[B]) => Rep[X]): Rep[X] = pair match {
    case Def(PairWrapper(pair)) => pair(k)
  }

  def pair_conditional[A, B](
    cond: Rep[Boolean],
    thenp: => Rep[CPSPair[A, B]],
    elsep: => Rep[CPSPair[A, B]]
  ): Rep[CPSPair[A, B]] = (thenp, elsep) match {
    case (Const(t), Const(e)) => PairWrapper(conditional(cond, t, e))
  }

  def paircps_toPair[A, B](pair: Rep[CPSPair[A, B]]): Rep[(A, B)] = pair match {
    case Def(PairWrapper(pair)) => pair.toPair
  }
}
```

Figure 3.8 – PairWrapper: LMS IR wrapper around `CPSPair`

With this extra wrapping layer, it is now possible to create values which are nested, yet deforested, pairs. The following staged program:

```
def aProgram(i: Rep[Int]): Rep[Int] = {
  val p = mkPair(mkPair(i + 1, i + 2), i + 3)
  p.apply { (p1, c) => p2.apply { (a, b) => a + b + c }}
}
```

generates, as expected, no intermediate pairs:

```
def generated(i: Int): Int = {
  val b = i + 1; val c = i + 2
  i + b + c
}
```

Customised handling of mutable variables. The final step is to customise mutable variables as well. Recall that we require them as join points in conditional expressions. In Figure 3.6 we specifically assigned values to both elements (`zeroVal[A]` and `zeroVal[B]`). Since we now have a wrapper for `CPSPair`, we can create default values for Church pairs using `zeroVal[PairCPS [A, B]]`: its implementation is to simply create a wrapper where the underlying values are themselves default values. So conditional expressions can now be implemented as follows:

```
def __ifThenElse[A, B](
    cond: Rep[Boolean],
    thenp: => CPSPair[A, B],
    elsep: => CPSPair[A, B]) = new CPSPair[A, B] {

  def apply[X](k: (Rep[A], Rep[B]) => Rep[X]): Rep[X] = {
    var tmpPair = zeroVal[PairCPS[A, B]]

    val assignK = (a: Rep[A], b: Rep[B]) => { tmpPair = mkPair(a, b) }
    if (cond) thenp.apply(assignK) else elsep.apply(assignK)
    k(tmpPair._1, tmpPair._2)
  }
}
```

Figure 3.9 – Virtualising conditional expressions for Church pairs after customising mutable variables

Variable creation, assignment and reading are virtualised constructs, and therefore can be overridden for our specific needs. With Scala Virtualised and LMS the `var_new`, `var_assign` and `readVar` methods can be customised. With Church pairs, this involves creating a specific wrapper `PairVar` that flags a mutable pair when a new variable is created. Just as is the case with `zeroVal` we forward actions on mutable operations to the underlying values. We elide the details of the implementation in this section, as it relates more to how the internals of LMS work than the principles themselves. We nonetheless provide the signature and general sketch for these in Figure 3.10. The interested reader may directly refer to the source code referred to in this section.

Discussion

In this section we have built up basic deforestation for products and sums from first principles (CPS-encodings), and judicious use of partial evaluation. We have also seen that both code blowup and mutable variable creation can be elegantly handled by this framework as well, provided we add minimal access to a meta intermediate representation. As a result the technique generalises to sums and products of any arity and nesting.

It may seem tedious, nonetheless, to implement such encodings for every data type we need in the program. Many compilers already optimise for some of these cases: Scala with value

```
trait CPSPairExp ... {
  case class PairVar[A, B](a: Var[A], b: Var[B])
    extends Def[Variable[PairCPS[A, B]]]
  def mkPairVar[A, B](a: Var[A], b: Var[B]) = PairVar(a, b)

  override def var_new[T](init: Exp[T]): Var[T] = init match {
    case Def(PairWrapper(PairStruct(a, b))) => ...
    case _ => super.var_new(init)
  }

  override def var_assign[T](lhs: Var[T], rhs: Exp[T]): Exp[Unit] = (lhs, rhs) match {
    case (Variable(Def(Reflect(PairVar(v1, v2), _, _))), Def(PairWrapper(p))) => ...
    case _ => super.var_assign(lhs, rhs)
  }

  override def readVar[T](v: Var[T])(implicit pos: SourceContext): Exp[T] = v match {
    case Variable(Def(Reflect(PairVar(v1 @ Variable(a), v2 @ Variable(b)), _, _))) => ...
    case _ => super.readVar(v)
  }
}
```

Figure 3.10 – Customising mutable variable creation for `CPSPair`

classes [58], Haskell with `newtype` and single-constructor datatypes [53]. Even C has support for structs which are stack allocated, and remove the need to carefully handle conditional expressions.

In fact, many optimisations performed by the Glasgow Haskell Compiler mirror, and surpass, those presented here [64, 63]. The main difference is that Haskell targets a larger variety of programs than suggested by the library approach here: we advocate APIs for algebraic data types that are fold-based [26]. These are already capable of expressing pipelines of operations on data, and optimising them involves fewer analysis algorithms than in the general functional programming language case.

If the language these libraries are developed in does not have good inlining/optmising capabilities, a developer can nonetheless choose to build some extra infrastructure, in the form of a partial evaluator for function composition. Arguably this is more complicated, and seems to contradict the idea of optimisations as libraries. Especially due to structures such as `PairWrapper` and `PairVar`, which require manipulating IR notes in a meta-language. We argue however that the knowledge required is minimal. A library developer needs to know how to create constant nodes in the meta-language of his choice, and how to extract underlying values from these nodes. In many frameworks, this remains easy. With Scala Macros for instance (see Section 2.3.2), we could create a specific subclass of the generic `Tree` type and match on it as and when needed.

Note also that the use of language virtualisation is solely for aesthetic purposes: it is desirable for a program written using a staged library to have a look and feel similar to that of a traditional program. Yet it is not mandatory. In the absence of virtualisation capabilities, a developer can offer API functions that are the equivalent to `__ifThenElse`, `var_new`, `var_assign` and `readVar`.

Related work

This chapter shows that Church encodings and staging can be systematically and practically combined to achieve deforestation as a library. As such, it brings together theoretical ideas and compiler implementations.

In his seminal paper, Wadler initially proposed deforesting programs that are in treeless form [86]. Church encodings are in treeless form: partial evaluating function composition corresponds to a usage of this treeless algorithm. Through staging we provide a library-level implementation of this algorithm. Since we support non-linear tree forms via the usage of conditional expressions we also support deforestation of blazed treeless forms.

The Church encoding of datatypes we present is very close in spirit to the Finally Tagless approach by Carette et al. [10]: any datatype we wish to not create during partial evaluation time is represented as a function that eventually yields a value. This is mainly used for embedding domain-specific languages in a host language without the use of complex constructs such as GADTs. We propose using this encoding for developing libraries. In addition, we also restrict dependence on a partial evaluator to inlining of function composition.

Beyond encodings at a library-level, continuations are a classic way to represent a programs structure in a compiler to perform further optimisations [44]: our treatment of conditional expressions is a reuse of Phi nodes from SSA, which has been shown to have a correspondence with CPS [5]. LMS also has a generic way to handle nested conditional expressions in a manner similar to Phi nodes [70]. In both cases the optimisation is performed at the intermediate representation level. Our use of the IR is merely for tying the knot with respect to nested data types. The implementation itself is lifted to the library level.

LMS enables embedding of domain-specific languages in Scala, and is based on a multi-staged approach. It draws inspiration from MetaOCaml [32]. Feldspar is a Haskell-based framework for embedding DSLs [6]. Many DSLs in this framework also tend to use a functional (tagless) representation of domain-specific data types to achieve deforestation.

Staged Shortcut Fusion Part I

Chapter 3 introduced us to a methodology to achieve deforestation for simple data types: Church-encode them, and partially evaluate function composition. We expand this technique to lists in this chapter. Lists are a compelling data structure: they represent a collection of elements that can be iterated over. The fusion techniques covered in this chapter therefore apply to any linear, traversable data structure.

This part of the thesis has partially been previously published [43], and handles shortcut fusion similar to `foldr/build` [29]. We extend it here to apply to Stream Fusion [15]. The code for both is available online [41].

4 Fold-Based fusion as a library

Introduction

Suppose we are given a list of people, along with a list of movies each of these people like. If we want to find out how many people like each movie, here is a Scala snippet to do the job:

```
def movieCount(people2Movies: List[(String, List[String])]): Map[String, Int] = {
  val flattened = for {
    (person, movies) <- people2Movies
    movie            <- movies
  } yield (person, movie)

  val grouped = flattened groupBy (_._2)
  grouped map { case (movie, ps) => (movie, ps.size) }
}
```

Figure 4.1 – The `movieCount` function in an idiomatic Scala style

The function creates intermediate data structures: `flattened` and `grouped` are explicitly declared, while some additional structures are implicitly created by the `for` comprehension. These data structures are helpful in organising the program and making it more readable. On the other hand, their allocation and construction incurs a significant memory and processing overhead. Yet it is possible to implement the `movieCount` function without creating any intermediate structures (Figure 4.2). The `movieCount2` function is arguably harder to read, but more efficient.

Fusion is a program transformation that converts functions written in a `movieCount` style to efficient equivalents in the `movieCount2` style. Its goal is to avoid the creation of costly intermediate data structures. Fusion has been extensively studied, both theoretically [34] and in practice [29, 80, 15].

Practical implementations of fusion tend to rely on an optimising compiler for a pure, func-

```
def movieCount2(people2Movies: List[(String, List[String])]): Map[String, Int] = {
  var tmpList = people2Movies
  val tmpRes: Map[String, Int] = Map.empty

  while (!tmpList.isEmpty) {
    val hd = tmpList.head
    var movies = hd._2

    while (!movies.isEmpty) {
      val movie = movies.head
      if (tmpRes.contains(movie)) {
        tmpRes(movie) += 1
      } else tmpRes.update(movie, 1)
      movies = movies.tail
    }
    tmpList = tmpList.tail
  }
  tmpRes
}
```

Figure 4.2 – The `movieCount` function, without intermediate structures

tional language. In non-pure languages, it is more difficult to implement fusion as part of the compiler, due to the possible presence of side-effects, open recursion in datatypes, virtual method dispatch, etc. There are however many pure, functional subdomains in such languages that could greatly benefit from fusion. Examples for such subdomains include collection libraries and query-like languages. Essentially, programs that process data through "pipelines" of operations are amenable to fusion.

Using the principles shown in Chapter 3, we present in this chapter libraries for performing fusion for pipelines of operations over iterable collections, or lists. This decouples the optimisation from an underlying compiler, making it portable, and readily applicable to different contexts. In particular:

- We present an API for staged, CPS-encoded lists (Section 4.2) based on the `foldLeft` function. Staging is used as a means to systematically separate function composition from data processing. The library closely follows that of classic lists, such that programmers require minimal changes to the way they program with classic lists. In reality, they compose operations over *code generators of folds*. This composition is partially evaluated away at staging time, yielding code that contains no intermediate data structures. Our fusion technique remains as powerful as `foldr/build` fusion [29]: it does well on functions that operate on a single input list.

- Some functions are not trickier: while consuming a single list, they produce multiple outputs. Partitioning and grouping fall under this category. We present variants of these functions that are easier to fuse (Section 4.3).

- These variants introduce extra boxes around data in order to continue operating under a single pipeline. We present a technique to systematically eliminate them. Once again, the key is to CPS-encode the data representations of the boxes, and stage these representations (Section 4.4).

By embracing generative programming as a paradigm [68] and combining it with functional APIs, we get an implementation that has a library look-and-feel.

Staging foldLeft

Many operations on lists can be implemented in terms of the generic fold function [29]. For lists, there are two variants of this function, `foldLeft` and `foldRight`. They are equivalent in that one can be implemented using the other (see below). We choose `foldLeft`: we will see later in this section why this representation benefits us more. Here is an implementation of `foldLeft`:

```scala
def foldLeft[A, S](ls: List[A])(z: S, comb: (S, A) => S): S = ls match {
  case Nil     => z
  case x :: xs => foldLeft(xs)(comb(z, x), comb)
}
```

Figure 4.3 – The `foldLeft` function on lists

It takes a zero (or initial) element of type `S`, and returns this element if the input list is empty. If the list contains some elements, they are recursively combined with the element `z` using the binary operator `comb`. The elements are combined to the left, hence the name of the function.

The `foldRight` function, on the other hand, combines elements to the right:

```scala
def foldRight[A, S](ls: List[A])(z: S, comb: (A, S) => S): S = ls match {
  case Nil     => z
  case x :: xs => comb(x, foldRight(xs)(z, comb))
}
```

Figure 4.4 – The `foldRight` function on lists

As mentioned above, an alternative implementation of `foldRight` can be achieved using `foldLeft`:

The idea is to make a first pass over the list using `foldLeft` to construct a function which will eventually combine the elements to the right.

```
def foldRight[A, S](xs: List[A])(z: S, comb: (A, S) => S): S = {
  val accumulated = foldLeft[A, S => S](xs)(s => s, { (acc, elem) =>
    s => acc(comb(elem, s))
  })
  accumulated(z)
}
```

Figure 4.5 – The `foldRight` function on lists implemented using `foldLeft`

An example of an operation on list that can be implemented with `foldLeft` is `map`:

```
def map[A, B](ls: List[A], f: A => B): List[B] = foldLeft[A, List[B]](ls)(
  Nil,
  (acc, elem) => acc :+ (f(elem))
)
```

Figure 4.6 – The `map` function on lists implemented using `foldLeft`

Starting with an empty list, the combination function simply appends (`:+`) to the accumulator the results of applying `f` to the elements of the input list. The append function on lists is inefficient in terms of complexity. One could however imagine using a more efficient collection (such as a list buffer) to efficiently append to the right, and convert it to a list at the end of `foldLeft`. We defer the presentation of the full API to Section 4.2.2.

Church-encoded lists. Recall from Section 2.1.3 that a Church-encoded list corresponds to the `foldRight` function. We can construct an alternate Church-encoding for lists using `foldLeft` instead (often referred to as snoc-lists). Consider the type signature of `foldLeft`:

```
List[A] => (S, (S, A) => S) => S
```

The signature tells us that, given a list over any type `A`, `foldLeft` returns a function that will fold the elements of that list into a structure of some type `S`. The type of this function is a Church encoding for lists, or equivalently the list functor [55]:

```
type CPSList[A, S] = (S, (S, A) => S) => S
```

Here, `S` denotes the eventual result type of operations over the list. For instance in the above `map` example, `S = List[A]`. In essence, `foldLeft` maps plain lists to Church lists. In the background section on Church lists(Section 2.1.3), we used `CPSList` to represent those that are resulting from `foldRight`. For the rest of this dissertation, we override the name so that it applies for `foldLeft` instead.

CPSList, staged

We now follow the principles elaborated in Chapter 3 to stage `CPSList`. The dynamic types here are the input type `A` and the eventual return type `S`:

```
type CPSList[A, S] = (Rep[S], (Rep[S], Rep[A]) => Rep[S]) => Rep[S]
```

Note that the name is deliberately overloaded. For the rest of the paper, unless explicitly mentioned, `CPSList` refers to the staged version. As promised, we use unstaged functions.

```
trait CPSListOps extends ListOps with IfThenElse ... with Variables with While {
  type Comb[A, S] = (Rep[S], Rep[A]) => Rep[S]

  abstract class CPSList[A] { self =>
    def apply[S](z: Rep[S], comb: Comb[A, S]): Rep[S]
    //operations on CPSList go here
  }
}

//companion object
object CPSList {
  //create a fold from a list
  def fromList[A](ls: Rep[List[A]]) = new CPSList[A] {

    def apply[S](z: Rep[S], comb: Comb[A, S]): Rep[S] = {
      var tmpList = ls
      var tmp = z
      while (!tmpList.isEmpty) {
        tmp = comb(tmp, tmpList.head)
        tmpList = tmpList.tail
      }
      tmp
    }
  }
  ...
}
```

Figure 4.7 – `CPSList` as a staged abstraction

Figure 4.7 shows an implementation of staged `CPSList` in LMS. The enclosing trait `CPSListOps` mixes in some of LMS' building blocks which help in composing code generators [71]. Compared to staged implementations of pairs and sums (Figures 3.1, 3.3), we mix in `While` to allow usage of staged while loops.

Every instance of `CPSList` must implement an `apply` method, corresponding to the application of fold. As explained above, the type parameter `S` for this method corresponds to the eventual structure resulting from the fold.

We create a `CPSList` from a regular list using the `fromList` function. Since `CPSList` corresponds to the return type of the foldLeft function on lists, fromList is essentially a staged code generator for foldLeft.

Here we choose an implementation using loops instead of recursion. This is because the target languages for our code generation (Scala, Java or C) are better at executing while loops than recursive functions. This also explains our choice of foldLeft: contrary to foldRight, it can be implemented in a tail-recursive manner, hence easily written as a low-level loop.

Note that fromList takes as parameter a Rep[List[A]], and not a List[Rep[A]]. Indeed, the input list to a pipeline of folds is not usually known statically.

The API of staged CPSList

We now extend our staged CPSList implementation by adding a list-like API. Note that these methods can be added to an unstaged CPSList as well. The only difference is the use of staged types and unstaged functions. The API consists of the usual suspects, map, filter and flatMap, shown in Figure 4.8.

```
//as methods on CPSList
def map[B](f: Rep[A] => Rep[B]) = new CPSList[B] {
  def apply[S](z: Rep[S], comb: Comb[B, S]) =
    self.apply(
      z,
      (acc: Rep[S], elem: Rep[A]) => comb(acc, f(elem)))
}

def filter(p: Rep[A] => Rep[Boolean]) = new CPSList[A] {
  def apply[S](z: Rep[S], comb: Comb[A, S]) =
    self.apply(
      z,
      (acc: Rep[S], elem: Rep[A]) =>
        if (p(elem)) comb(acc, elem) else acc)
}

def flatMap[B](f: Rep[A] => CPSList[B]) = new CPSList[B] {
  def apply[S](z: Rep[S], comb: Comb[B, S]) =
    self.apply(
      z,
      (acc: Rep[S], elem: Rep[A]) => f(elem)(acc, comb)
    )
}
```

Figure 4.8 – The API of CPSList

CPSList.map. The map on CPSList is fairly straightforward and follows the classic implementation using foldLeft (Figure 4.6). The combine function applies (the unstaged function) f on the element it receives.

`CPSList.filter`. The `filter` function combines an element if it passes the unstaged predicate function `p`, and returns the accumulator unchanged otherwise. Note that the conditional expression is also a code generator since it takes a value of type `Rep[Boolean]` as the condition.

`CPSList.flatMap`. The type of the function argument `f` of `flatMap` deserves some elaboration. Expanding the type of `CPSList`, we get the following type for `f`:

```
f: Rep[A] => (Rep[S], Comb[B, S]) => Rep[S]
```

which is a curried, unstaged function. By fully applying this function, we inline not only the body of `f`, but also the body of the resulting `CPSList`. This way, we avoid generating code for an intermediate collection.

Note that the function passed to `flatMap` must return `CPSList`. If this `CPSList` is created from a call to `fromList`, this means that it makes use of a dynamic list after all: this list will not be fused. A programmer must therefore be careful how he creates a `CPSList` when using `flatMap`. We can naturally extend the library to have easier methods of creating static lists. In particular, there can be a method `fromStaticList` that takes a `List[Rep[B]]`, i.e. a static list of dynamic values.

It is also possible to create a `CPSList` from a range of numbers, so that there is no need for an initial list either:

```
//in the companion object of CPSList
def fromRange(a: Rep[Int], b: Rep[Int]) = new CPSList[Int] {
  def apply[S](z: Rep[S], comb: Comb[Int, S]) = {
    var tmpInt = a
    var tmp = z
    while (tmpInt <= b) {
      tmp = comb(tmp, tmpInt)
      tmpInt = tmpInt + 1
    }
    tmp
  }
}
```

Figure 4.9 – The `fromRange` function for constructing a `CPSList`

A code generation example. LMS takes as input a staged program, and generates a later-stage program. Consider an example that uses `CPSList`, shown in Figure 4.10. Given an integer interval, it creates nested intervals. It then sums all odd elements of the nested intervals, after

```
def cpsListExample(a: Rep[Int], b: Rep[Int]): Rep[Int] = {
  val fld = CPSList.fromRange(a, b)
  val flatMapped = fld flatMap {
    i => CPSList.fromRange(1, i)
  }
  val filtered = flatMapped filter (_ % 2 == 1)
  filtered.map(_ * 3).apply[Int](
    0, (acc, x) => acc + x
  )
}
```

Figure 4.10 – An example pipeline using the `CPSList` API

```
def generatedFunction(x0: Int, x1: Int): Int = {
  var x2: Int = x0; var x3: Int = 0
  while (x2 <= x1) {
    val x7 = x3; val x8 = x2
    var x9: Int = 1; var x10: Int = x7
    while (x9 <= x8) {
      val x14 = x10; val x15 = x9
      val x16 = x15 % 2
      val x17 = x16 == 1
      val x20 = if (x17) {
        val x18 = x15 * 3
        val x19 = x14 + x18; x19
      } else { x14 }
      x10 = x20
      val x22 = x15 + 1; x9 = x22
    }
    val x26 = x10; x3 = x26
    val x28 = x8 + 1; x2 = x28
  }
  val x32 = x3; x32
}
```

Figure 4.11 – The partially evaluated result of `cpsListExample` using LMS

having multiplied them by 3. Note that in the `flatMap` call, we pass a function that creates a fold from an interval, rather than from a list. Running LMS will partially evaluate the staged `CPSList` away, yielding code as in Figure 4.11. As we can see, we are left with two nested while loops, exactly what we wished for.

The power of staged CPSList. We now have a library over a staged fold abstraction, which enables us to write pipelines of operations over lists. Through partial evaluation, we generate code that is devoid of intermediate data structures. The main difficulty consisted in identifying the correct types for unstaged function arguments.

It is natural to wonder how many of the common operations over lists are fusible by this technique in practice. Our staged `CPSList`, being purely fold based, is as powerful as *foldr/build* fusion [29].

Partitioning and grouping

We have so far only considered list operations that output exactly one list as their result. It is not much of a surprise that such functions should be amenable to fusion since their composition will always result in "straight pipelines", i.e. functions which again take lists to lists. This is sometimes referred to as vertical fusion.

In this section we turn to operations that produce multiple outputs, and hence allow us to build forked pipelines. The main challenge consists in keeping all operations in the same pipeline, while avoiding the introduction of intermediate data structures to do so. This is also known as horizontal fusion. We start with the `partition` function.

Partition

The `partition` function on lists takes a list and a predicate, and returns two lists, one containing the elements satisfying the predicate, and the other containing those that do not. We can implement this function using `foldLeft` as defined in Section 4.2:

```
def partition[A](ls: List[A], p: A => Boolean): (List[A], List[A]) = {
  foldLeft[A, (List[A], List[A])](ls)(
    (Nil, Nil), {
      case ((trues, falses), elem) =>
        if (p(elem)) (trues ++ List(elem), falses)
        else         (trues, falses ++ List(elem))
    })
}
```

Figure 4.12 – The `partition` function on lists using `foldLeft`

The initial element is a pair of empty lists. Based on the predicate, we add each element of the input list to either the first of the second accumulating list. Here is an example usage of `partition`:

```
val myList: List[Int] = ...
val (evens, odds) = partition(myList, (x: Int) => x % 2 == 0)
(evens map (_ * 2), odds map (_ * 3))
```

In the context of fusion, we naturally want to avoid creating the `evens` and `odds` lists.

A naive attempt. One way to implement `partition` on `CPSList` is to have it return two separate `CPSList`s:

```
//as a method on CPSList
def partition(p: Rep[A] => Rep[Boolean]): (CPSList[A], CPSList[A]) = {
  val trues  = this filter p
  val falses = this filter (a => !p(a))
  (trues, falses)
}
```

Though we create a pair in the above code, it is unstaged and so is partially evaluated away. Moreover, we can access both `CPSLists` separately and further construct their pipelines separately.

Unfortunately, if both `trues` and `falses` are used later on, code for two separate traversals over the entire pipeline will be generated, which defeats the point of fusion. It is preferable to have a single traversal.

Partition with Either. If our objective is to generate a single traversal, we must fix the return type for `partition` to be `CPSList`, our current abstraction for loops. This particular `CPSList` does not see elements of type `A` anymore, but elements that have either passed a predicate, or not. The `Either` type captures this notion very well: instances of `Left` represent elements satisfying the predicate, instances of `Right` represent elements that do not. We can rewrite the example above as shown in Figure 4.13.

```
def partitionE[A](ls: List[A], p: A => Boolean): List[Either[A, A]] =
  ls map { elem => if (p(elem)) Left(elem) else Right(elem) }

val myList: List[Int] = ...
val partitioned = partitionE(myList, (x: Int) => x % 2 == 0)
val mapped = partitioned map {
  case Left(x) => Left(x * 2)
  case Right(x) => Right(x * 3)
}

foldLeft[Either[Int, Int], (List[Int], List[Int])](mapped)(
  (Nil, Nil), {
    case ((trues, falses), elem) =>
      elem.fold(
        x => (trues ++ List(x), falses),
        x => (trues, falses ++ List(x)))
  })
```

Figure 4.13 – The `partition` function with `Either`

The `partitionE` function is simply an application of `map`, turning an element of type `A` into an element of type `Either[A, A]`. It has the effect of *delaying* the creation of two separate lists to a later application of foldLeft. Between the final application and the partition point, we use the map function on Either to thread computations through to the actual values. Essentially, Either acts as a *box* that wraps underlying values.

Note that *eventually*, we are left with no option but to fork the pipeline into two lists, through a final call to foldLeft. Here, the combination operation concatenates elements to the resulting lists through the use of the fold function on Either.

The staged version of partitionE (Figure 4.14) is analogous. It uses the functions left and right, which create instances of Rep[Either].

```
//as a method on CPSList
def partitionBis(p: Rep[A] => Rep[Boolean]): CPSList[Either[A, A]] = {
  this map { elem =>
    if (p(elem)) left[A, A](elem) else right[A, A](elem)
  }
}
```

Figure 4.14 – The partition function on CPSList

The reader will surely object to this implementation. We have not really eliminated intermediate data structures. Rather, we have created new ones, in the form of instances of Rep[Either]. The insight is that we *know* exactly what type of boxes we create. We will discuss shortly how to eliminate them (Section 4.4). Before that, we discuss another multiple output producer function, groupBy.

GroupBy

The partition function on CPSList allows us to write pipelines so that no intermediate lists are created, and the single traversal requirement is met. We now focus our attention on a cousin of partition's, groupBy.

While partitioning splits a list into two groups, groupBy partitions a list into possibly many groups. This operation is also particularly interesting because it is a common query operation. It is of course used in query languages, but it is also not uncommon in spreadsheet-like languages to visualise results better. Recall the example in Section 4.1, where we group movies by people who like them, and then count the number of people per group.

For lists, the groupBy function can be implemented using foldLeft (Figure 4.15). It takes an input list, and a function f that attributes a key to a value. It returns a collection of key-value pairs, where the value is itself a collection of values from the input list ls. The initial element passed to the fold is an empty map (Map.empty). The combination operator adds a new key-value pair to the map if the key has not been created yet. Otherwise, it appends the element to the pre-existing list.

```
def groupBy[A, K](ls: List[A], f: A => K): Map[K, List[A]] = {
  foldLeft[A, Map[K, List[A]]](ls)(
    Map.empty[K, List[A]], {
      case (dict, elem) =>
        val k = f(elem)
        if (dict.contains(k))
          dict + ((k, dict(k) ++ List(elem)))
        else
          dict + ((k, List(elem)))
    })
}
```

Figure 4.15 – The `groupBy` function on lists using `foldLeft`

We can reimplement the example from the introduction using the above implementation of `groupBy`:

```
def movieCount(people2Movies: List[(String, List[String])]): Map[String, Int] = {
  val flattened = for {
    (person, movies) <- people2Movies
    movie            <- movies
  } yield (person, movie)

  val grouped = groupBy[(String, String), String](flattened, _._2)
  grouped map { case (movie, ls) => (movie, ls.size) }
}
```

Figure 4.16 – The `movieCount` function using a variant of `groupBy`

Note that we use the `map` function on the `Map` data structure after the call to `groupBy`. Once again, in terms of fusion, we would like to avoid creating the intermediate `HashMap[Int, List[Int]]`. One possibility for the above example is to implement a specific `reduceBy` function that takes an extra reduction function and applies it. Many collection libraries do indeed contain this alternative. We may however want to first group elements, perform group-specific operations on the values, and then reduce them. In which case a `reduceBy` will not suffice.

Delaying the application of the final fold. As in the case for partition, the key idea is to keep everything on a single fold pipeline for as long as possible. To achieve this, we once again resort to introducing an extra *box* type, through the use of a function named `groupWith`. This function is shown in Figure 4.17.

The result of applying a `groupWith` is a `CPSList` over key-value pairs. Values from the input fold are simply tagged with their group, and sent further down the pipeline. We can therefore finally fully implement the `movieCount` example using the staged `CPSList` API (Figure 4.18). One might

```scala
//as a method on CPSList
def groupWith[K](f: Rep[A] => Rep[K]): CPSList[(K, A)] =
  this map { elem => (f(elem), elem) }
```

Figure 4.17 – The `groupWith` function on `CPSList`

argue that this code is as difficult to write as the low-level loop version seen in Section 4.1, due to the added complexity of `Rep` and `CPSList` annotations. While this is admittedly true for our small example, writing hand-optimised loops is error-prone and does not scale to larger, more complex pipelines, especially those spanning multiple functions.

```scala
def repMovieCount(
    people2Movies: Rep[List[(String, List[String])]]): Rep[HashMap[String, Int]] = {

  val fld = CPSList.fromList[(String, List[String])](people2Movies)

  val flattened: CPSList[(String, String)] = for {
    elem  <- fld
    movie <- CPSList.fromList[String](elem._2)
  } yield (elem._1, movie)

  val grouped = flattened groupWith { elem => elem._2 }
  grouped.apply[HashMap[String, Int]](
    HashMap[String, Int](),
    (dict, x) =>
      if (dict.contains(x._1)) dict + (x._1, dict(x._1) + 1)
      else                     dict + (x._1, 1)
  )
}
```

Figure 4.18 – A staged version of `movieCount`, using the `CPSList` API

Summary. In this section, we integrated multiple output producers to the staged fold API. This was done by implementing variants of the functions that delay the final application of fold by boxing elements into a type that preserves information about the multiple output separation. We also preserve the `CPSList` representation in the process.

These extra boxes unfortunately manifest themselves in the generated code. In the next section we show how to eliminate this overhead.

Removing boxes

So far, we have successfully integrated `partition` and `groupBy` into the staged `CPSList` abstraction. Unfortunately this leads to the creation of boxes (`Either[A, B]` for partition, `(K, V)` for grouping) around elements. In this section, we discuss how to eliminate these boxes.

We observe that, inside the `CPSList` pipeline, we are free to choose any representation for our boxes, provided we can reconstruct the original representation at the end of the pipeline. In other words, we do not need to create instances of `Rep[Either[A, B]]` or `Rep[(K, V)]` until the final application of `CPSList`. In particular, by using CPS-encoded versions of the boxes inside the pipeline, we can delay their construction, much like we delay the construction of lists!

Tying the knot with staged products and sums

We can directly reuse Church pairs and sums seen in Sections 3.1 and 3.2, respectively. For context, here are their staged encodings (full implementations can be found in Figures 3.1 and 3.3):

```
abstract class CPSPair[A, B] {
  def apply[X](k: (Rep[A], Rep[B]) => Rep[X]): Rep[X]
}

abstract class CPSEither[A, B] {
  def apply[X](l: Rep[A] => Rep[X], r: Rep[B] => Rep[X]): Rep[X]
}
```

Getting back to `CPSList`, we can now implement `partition` using `CPSEither`. We face one final issue though. We may think that `partition` can be written as follows:

```
def partitionCPS(p: Rep[A] => Rep[Boolean]): CPSList[CPSEither[A, A], S] = {
  this map { elem =>
    if (p(elem)) mkLeft[A, A](elem)
    else         mkRight[A, A](elem)
  }
}
```

However, `CPSEither` expects a `Rep` type as its first argument: it expects a `Rep[CPSEither[A, A]]` but we provide a plain `CPSEither[A, A]`.

We are in the presence of nested CPS encodings once again (cf. Section 3.2.2). Just as with products, we also statically know all points at which an instance of `CPSEither` is created in the list pipeline: precisely in each branch of the partition predicate. Therefore we can once again use constant wrappers around `CPSEither`. Figure 4.19 shows the implementation of this wrapper. Every method on `CPSEither` has a corresponding method in `EitherWrapper` which simply forwards the implementation down.

```
trait CPSEitherOpsExp extends CPSEitherOps with ... {

  import CPSEither._
  //The wrapper acts as a Rep[CPSEither[A, B]]
  case class EitherWrapper[A, B](e: CPSEither[A, B]) extends Def[CPSEither[A, B]]

  def mkLeft[A, B](a: Rep[A]) = EitherWrapper(mkLeft[A, B](a))
  def mkRight[A, B](b: Rep[B]) = EitherWrapper(mkRight[A, B](b))

  def either_map[A, B, C, D](
      e: Rep[CPSEither[A, B]],
      lmap: Rep[A] => Rep[C],
      rmap: Rep[B] => Rep[D]): Rep[CPSEither[C, D]] = e match {
    case Def(EitherWrapper(sth)) =>
      EitherWrapper(sth map (lmap, rmap))
  }

  def either_apply[A, B, X](
      e: Rep[CPSEither[A, B]],
      lf: Rep[A] => Rep[X],
      rf: Rep[B] => Rep[X]): Rep[X] = e match {
    case Def(EitherWrapper(sth)) => sth.apply(lf, rf)
  }

  def __ifThenElse[A, B](
      cond: Rep[Boolean],
      thenp: => Rep[CPSEither[A, B]],
      elsep: => Rep[CPSEither[A, B]]): Rep[CPSEither[A, B]] = {

    (thenp, elsep) match {
      case (Def(EitherWrapper(t)), Def(EitherWrapper(e))) =>
        EitherWrapper(conditional(cond, t, e))
    }
  }
}
```

Figure 4.19 – `EitherWrapper`: LMS IR wrapper around `CPSEither`

Related work

Fusion, or deforestation, has been studied extensively. One of the first known techniques is Wadler's algorithm for eliminating intermediate trees [86]. For list-like pipelines, there are three main algorithms: `foldr/build` fusion [29], which is based on implementing list operations as folds. Its dual, `destroy/unfoldr` fusion, fuses multiple input consumers such as zips well [80]. Stream fusion [15, 14] converts list operations to operations on streams, and can handle both kinds of functions well. All three have been implemented using Haskell's rewrite rule system [40]. The technique presented here is an instance of, and therefore as powerful as, `foldr/build` fusion.

Fusion systems have also been studied theoretically. Meijer et al. [55] propose a theoretical framework for functional programs that are based on high-level recursive operations over algebras. The CPS-encoded datatypes (`FoldLeft`, `EitherCPS`) we use are instances of such algebras.

Hinze et al. provide a theoretical framework that unifies the above-mentioned fusion algorithms [34]. Ghani et al. generalise `foldr/build` fusion to other inductive datatypes [25]. Although we only treat lists, sums and pairs, their work suggests that our technique can be extended to other inductive datatypes.

LMS also proposes its own fusion algorithm for indexed loops [70]. This algorithm performs both horizontal and vertical fusion on representations of loops and provides facilities for heterogeneous code generation. However, while the framework embraces the "fusion as a library" approach, it also relies heavily on LMS' compiler infrastructure. Our goal here was to avoid this kind of dependency, and implement a simple library based entirely on partial evaluation.

Partial evaluation and multi-stage programming have been used with great success to optimise programs. The general idea is to apply the first Futamura projection to turn interpreters into compilers [24]. The LMS framework enables us to compose code generators; we effectively operate in a generative programming language [68].

Svensson et al. use defunctionalisation to unify push and pull arrays in an embedded DSL context [82]. Much like our approach, their representation effectively turns a CPS-encoded array into a code generator.

Conclusion

We have shown how to implement fold-based fusion as a library. The key is to represent data-structures using their CPS-encodings. As a result, composition over these data structures turns into function composition. We then partially evaluate function composition to achieve vertical fusion.

The technique readily extends to multiple output producers such as partitioning and grouping operations by introducing additional boxes. By CPS-encoding the box types, we are once again able to apply partial evaluation to eliminate intermediate data structures, and achieve horizontal fusion.

We used LMS as our staging/partial evaluation framework of choice: our implementation is available as an open-source project [41]. Our approach is, however, not tied to a particular framework. Indeed, any system capable of partially evaluating function composition is sufficient.

5 Stream fusion as a library

Introduction

Fold-based fusion is rather elegant. We simply Church-encode lists and are able to fuse quite a few operations on lists, as these can be expressed using `foldLeft`. Some functions are unfortunately not so straightforward to implement as a fold. The most popular example is the `zip` function, which takes two lists as inputs and outputs a list of paired elements from each list:

```
def zip[A, B](as: List[A], bs: List[B]): List[(A, B)] = (as, bs) match {
  case (Nil, _) => Nil
  case (_, Nil) => Nil
  case (a :: as2, b :: bs2) => (a, b) :: zip(as2, bs2)
}
```

Figure 5.1 – The `zip` function on lists

This function can be implemented using `foldLeft` as well:

```
def zip[A, B](as: List[A], bs: List[B]): List[(A, B)] = {
  val (res, _) = foldLeft[A, (List[(A, B)], List[B])](as)(
    (Nil, bs),
    { case ((abs, bs2), a) => bs2 match {
        case Nil => (abs, bs2)
        case b :: bs3 => (abs :+ (a, b), bs3)
    }})
  res
}
```

Figure 5.2 – The `zip` function on lists using `foldLeft`

The accumulator tracks both the state of the second list `bs` and the resulting list of pairs `abs`.

The combination function creates a pair if the second list still has elements.

There are two issues with this implementation that make fusion difficult. First of all, even if the second list `bs2` becomes empty, we still iterate over the rest of the elements of `as`. We could jump out of the function at that point by returning the accumulated list of pairs (**`return`** `abs`). But this creates two exit points for `zip`, and breaks the abstraction of a single pipeline.

Secondly, we manually inspect and extract from `bs`. Which means that the list must be materialised at this point, and thus becomes an intermediate structure in the process of creating the final list of pairs `abs`.

Inherently, `zip` takes more than one list as its input: this is not easily represented in terms of a fold. An alternate representation for list operations is via the dual operation to fold, unfold. Stream fusion [15] is based on such a representation in order to fuse pipelines of operations on lists. As a continuation of the main theme in this dissertation, we implement a variant of staged stream fusion in this chapter. While `zip`-like functions work well in an unfold setting, the implementations of `filter` and `flatMap` become more complicated [80] (Section 5.2).

List operations as unfolds

The dual to the `fold` function, which consumes elements from a list, is `unfold`, which produces elements from a (possibly infinite) source into a list:

```
def unfold[S, A](source: S)(step: S => Option[(A, S)]): List[A] = {
  def loop(tmpRes: List[A], tmpSeed: S): List[A] = step(tmpSeed) match {
    case Some((elem, rest)) => loop(tmpRes :+ elem, rest)
    case None => tmpRes
  }
  loop(Nil, source)
}
```

Figure 5.3 – The `unfold` function on lists

Given an initial `source`, and a stepper function, `unfold` extracts elements from the source until there are no more elements to be extracted, when `step` returns `None`.

```
def map[A, B](ls: List[A], f: A => B): List[B] = unfold[List[A], B](ls){
  case Nil => None
  case x :: xs => Some((f(x), xs))
}
```

Figure 5.4 – The `map` function on lists using `unfold`

An example of a list operation using `unfold` is `map`, shown in Figure 5.4. The `zip` function now

```
def zip[A, B](as: List[A], bs: List[B]): List[(A, B)] = {
  unfold[(List[A], List[B]), (A, B)]((as, bs)){
    case (Nil, _) => None
    case (_, Nil) => None
    case (a :: as2, b :: bs2) => Some((x, y), xs))
  }
}
```

Figure 5.5 – The `zip` function on lists using `unfold`

becomes easy to implement as well (Figure 5.5): the source is a pair comprising both initial lists. The stepper function extracts a pair if both lists have at least one element.

The `filter` and `flatMap` functions are not so straightforward however. Here is `filter`:

```
def filter[A](ls: List[A], p: A => Boolean): List[A] = {
  def inner(tmpSource: List[A]): Option[(A, List[A])] = tmpSource match {
    case Nil => None
    case x :: xs =>
      if (p(x)) Some(x, xs)
      else inner(xs)
  }

  unfold[List[A], A](ls)(inner _)
}
```

Figure 5.6 – The `filter` function on lists using `unfold`

The stepper function provided to `unfold` is itself recursive: it eagerly pulls elements from the source until is sees one that passes the predicate. This recursive nature makes it difficult to fuse `filter`, as it introduces nested recursion. Both `fold` and `unfold` are already recursion schemes, and it is in our interest to keep operations that use these non-recursive.

The `flatMap` function is even trickier (Figure 5.7). At all times in the stepper function, we must know whether to pull from the an inner list of type `List[B]`, or from the source list and apply `f`. Hence the use of two kinds of flags:

- The source is a pair of the source list as well as a potential inner list. We only pull from the outer list when the inner list is itself empty.
- The output type to the `unfold` function is `Option[B]`. When the inner list becomes empty, we must pull from the outer list. Rather than applying `f` directly and pulling from the resulting list (which would once again result in nested recursion), we produce a "bubble" value. These bubbles are removed by a final call to `foldLeft`.

```
def flatMap[A, B](ls: List[A], f: A => List[B]): List[B] = {
  val innerList: List[Option[B]] = unfold[(List[B], List[A]), Option[B]] {
    case (Nil, Nil) => None
    case (Nil, a :: as) => Some((None, (f(a), as)))
    case (b :: bs, as) => Some((Some(b), (bs, as)))
  }((Nil, ls))

  foldLeft[Option[B], List[B]](ls)(
    Nil,
    (acc, elem) => elem match {
      case None => acc
      case Some(b) => acc :+ b
    }
  )
}
```

Figure 5.7 – The `flatMap` function on lists using `unfold`

Staging streams

```
trait StreamOps extends CPSListOps with PairCPS with OptionCPS {
  abstract class Stream[A, Source] { self =>
    def source: Rep[Source]
    def atEnd(s: Rep[Source]): Rep[Boolean]
    def next(s: Rep[Source]): Rep[CPSPair[CPSOption[A], Source]]
}
```

Figure 5.8 – The interface of the staged streams library

Keeping in mind the difficulties with `filter` and `flatMap` above, we propose a staged stream implementation in this section. Its interface is given in figure 5.8: a `Stream[A, S]` sees elements of type `A`, from a source of type `S`. Its signature closely mirrors that of `unfold`. We decouple the stepper into two methods `atEnd` and `next`. The main difference is that `next` returns a value of type `OptionCPS[A]` rather than just `A`: this encodes the fact that we may add bubbles in the pipelines (see below). This decoupling reveals the closeness of streams to iterators in imperative programming, with the difference being that that ours have a functional interface. Note that we use Church-encoded pairs and options, to benefit from deforestation for these. We describe key methods from the API here below. We then treat `zip` and `flatMap` individually.

`Stream.map`. The `map` function is once again easy to implement. It creates a stream that has the same `source` and `atEnd` methods as the one it wraps around. The `next` method pulls an element and applies the mapping function `f` to an element, if there is one (Figure 5.9).

```
// as a method on Stream
def map[B](f: Rep[A] => Rep[B]) = new Stream[B, Source] {
  def source = self.source
  def atEnd(s: Rep[Source]): Rep[Boolean] = self.atEnd(s)
  def next(s: Rep[Source]): Rep[PairCPS[OptionCPS[B], Source]] = {
    val nextAndRest = self.next(s)
    nextAndRest.map(
      nxt => nxt map f,
      rst => rst
    )
  }
}
```

Figure 5.9 – The `map` function on streams

`Stream.filter`. The crucial bit with `filter` is to propagate a `None` downstream when the predicate is not satisfied. The solution of inserting a bubble helps in keeping the implementation non-recursive (Figure 5.10).

```
// as a method on Stream
def filter(p: Rep[A] => Rep[Boolean]) = new Stream[A, Source] {
  def source = self.source
  def atEnd(s: Rep[Source]): Rep[Boolean] = self.atEnd(s)
  def next(s: Rep[Source]): Rep[PairCPS[OptionCPS[A], Source]] = {
    val nextAndRest = self.next(s)
    nextAndRest.map(
      nxt => nxt filter p,
      rst => rst
    )
  }
}
```

Figure 5.10 – The `filter` function on streams

`Stream.toCPSList`. The final step in a steam pipeline is to collapse all desired elements into some structure. In terms of recursion schemes, the pipeline corresponds to a hylomorphism that needs a final catamorphism (fold) that is composed with the initial anamorphism (unfold). It is at this point that bubbles created by previous functions are eliminated: we pass the `unit` continuation to a `None` element (Figure 5.11). Also, thanks to the customised treatment of default values (`zeroVal`) and mutable variable assignment for Church-encoded structures (cf. Section 3.2.2), we are guaranteed that there is no intermediate object creation in the while loop.

```
//as a method on Stream
def toCPSList = new CPSList[A] {
  def apply[Sink](z: Rep[Sink], comb: Comb[A, Sink]): Rep[Sink] = {
    var tmpSource = source
    var tmpSink = z

    while (!atEnd(tmpSource)) {
      var optA = zeroVal[OptionCPS[A]]

      /* peeling out the pair and the option */
      next(tmpSource).apply { (optcps, src) =>
        optA = optcps
        tmpSource = src
      }

      optA.apply(
        _ => unit(()),
        e => tmpSink = comb(tmpSink, e)
      )
    }
    tmpSink
  }
}
```

Figure 5.11 – Closing the hylomorphism on streams with `toCPSList`

Zip

The implementation of `zip` for streams is more complex than its `unfold` counterpart (see Figure 5.5), because of the extra `OptionCPS` wrapper around an element. Its signature is the following:

```
def zip[B, S2](
  that: Stream[B, S2]
): Stream[PairCPS[A, B], PairCPS[OptionCPS[A], PairCPS[Source, S2]]]
```

The `source` for the resulting stream combines both the left and right sources. In addition there is an intemediate state holding an `OptionCPS[A]`. It allows us to decide whether to pull from the left or the right source, depending on whether it is empty or contains a value, respectively. Initially it is naturally empty:

```
def source = mkPair(mkNone[A], mkPair(self.source, that.source))
```

A zipped stream becomes empty when either the left or right stream are empty. We must also check first whether the intermediate state itself is empty, lest it be possible to create another pair (Figure 5.12).

The `next` method creates a pair of values once they are availabe. For this to happen, the

```
def atEnd(p: Rep[InnerSource]): Rep[Boolean] = {
  p.apply { (optA, sources) => sources.apply { (s1, s2) =>
    (self.atEnd(s1) && !optA.isDefined) || that.atEnd(s2)
  }}
}
```

Figure 5.12 – The `atEnd` method on a zipped stream

intermediate state should contain a value, and pulling from the stream to the right must also return a value. In other cases, a bubble is created, and modifications are made to the remaining source if the intermediate state is empty and the left stream yields an element:

```
def next(s: Rep[InnerSource]): Rep[PairCPS[OptionCPS[PairCPS[A, B]], InnerSource]] = {
  s.apply { (optA, sources) => sources.apply { (s1, s2) => optA.apply(
    /** if ‘optA‘ not defined we pull from ‘s1‘ */
    _ => {
      val nextAandRest = self.next(s1)
      nextAandRest.apply { (newOptA, s1Rest) =>
        mkPair(mkNone[PairCPS[A, B]], mkPair(newOptA, mkPair(s1Rest, s2)))
      }
    },

    /** if ‘optA‘ is defined we pull from ‘s2‘ */
    a => {
      val nextBandRest = that.next(s2)
      nextBandRest.apply { (optB, s2Rest) =>
        optB.apply(
          /** if ‘optB‘ not defined we continue */
          _ => mkPair(mkNone[PairCPS[A, B]], mkPair(optA, mkPair(s1, s2Rest))),

          /** if ‘optB‘ is defined we can create a pair */
          b => mkPair(mkSome(mkPair(a, b)), mkPair(mkNone[A], mkPair(s1, s2Rest)))
        )
      }
    }
  )}}
}
```

Figure 5.13 – The `next` method on a zipped stream

Thanks to `zip`, it is now possible to fully deforest the classic functional implementation of the dot product function:

```
def dotProduct(a: Stream[Int, Int], b: Stream[Int, Int]): Rep[Int] = {
  val dotted = (xs zip ys).map(pair => pair.apply((x, y) => x * y))
  dotted.toFold.apply[Int](unit(0), (acc, x) => acc + x)
}
```

Partial evaluation of this function generates code with no intermediate data structures, as a single loop that iterates over two indices, one for each input stream.

FlatMap

The `flatMap` function is given by the following signature:

```
def flatMap[B, S2](
  f: Rep[A] => Stream[B, S2]
): Stream[B, PairCPS[Source, OptionCPS[PairCPS[A, S2]]]]
```

It takes a function which for an element of type `Rep[A]` yields a new static stream, whose element and source types are *both* known. The resulting stream naturally yields elements of type `Rep[B]`. Its source consists of two elements, the outer source of type `Source`, and internal state that marks both which current outer element we are flattening, as well as the current inner source:

```
def source = mkPair(self.source, mkNone[PairCPS[A, S2]])
```

We have reached the end of a `flatMap` stream if the outer source is empty, and there is nothing left to flatten:

```
def atEnd(s: Rep[TotalSource]): Rep[Boolean] = s.apply { (outer, inner) =>
  self.atEnd(outer) && !(inner.isDefined)
}
```

To compute a next element, if the internal state is defined, it means that we can pull from the inner source, otherwise that we must pull from the outer source. The flattened stream reaches its end if both the outer stream has reached its end, and if the inner state is empty (Figure 5.14).

Discussion

The stream implementation presented here reveals the type of the source very early. A programmer must therefore provide a source type early when declaring a pipeline. An alternate implementation would be to have the source parameter as a type member:

```
abstract class Stream[A] {
  type S
}
```

The main reason for not doing so is due to the difficulty of deforesting `flatMap`. Indeed, we restrict the function `f` passed to `flatMap` to return a stream whose source is of the same type for *every* element `a: A` of the initial stream. In the general case, the type of a stream's source could depend on the value of an element.

```
def next(s: Rep[TotalSource]): Rep[PairCPS[OptionCPS[B], TotalSource]] = {
  s.apply { (outer, optInner) =>
    optInner.apply(
      /** if no inner source, pull from outer */
      _ => {
        val nextAndRest = self.next(outer)
        nextAndRest.apply { (optA, rest) =>
          mkPair(mkNone[B], mkPair(rest, optA map (a => mkPair(a, f(a).source))))
        }
      },

      /** if the inner source is defined we pull from it */
      inner => inner.apply { (innerA, innerSource) =>

        /** calling the innerStream, to have access to the functions */
        val innerStream = f(innerA)

        if (innerStream.atEnd(innerSource)) {
          mkPair(mkNone[B], mkPair(outer, mkNone[PairCPS[A, S2]]))
        } else {
          val nextAndRest = innerStream.next(innerSource)
          nextAndRest.apply { (optB, rest) =>
            mkPair(optB, mkPair(outer, mkSome(mkPair(innerA, rest))))
          }
        }
      }
    )
  }
}
```

Figure 5.14 – The next method on a flatmapped stream

Such cases are far more difficult to optimise. In the original stream fusion work by Coutts et al. [15], optimising this function involves complex compiler optimisations such as the case of case transformation, and even then, not all boxes can be always removed [14]. Coutts proposes a weaker version of concatMap (Haskell's name for flatMap), known as the name-capturing flatten, that corresponds to the version of flatMap presented here. Farmer et al. propose a tranformation from flatMap to flatten (when possible) using the HERMIT framework [19]. We choose instead to restrict the operation by design. In practice, pipelines of streams using flatMap tend to have a single inner source type, and therefore this is not, we believe, a major restriction. Moreover, it allows us to benefit from deforestation without having to implement more complex analysis algorithms for the body of a general flatMap.

We saw implementations of partitioning and grouping functions that wrap an element as a sum or a product, respectively (Section 4.3). These implementations are a simple use of the map function itself. As a result they directly carry over to streams.

The `toCPSList` method on streams also demonstrates the ease with which two a priori distant deforestation algorithms can be connected for great benefit. One might wonder, as a result, why we need to implement functions such as `flatMap` and `filter` in a stream context. Is it not enough to call `toCPSList` first and only then use these functions? Unfortunately some pipelines may require performing these operations *before* the call to a `zip` function. Shaikhha et al. further explore the relation between fold and unfold fusion in query engines of database systems, and the quality and performance of generated code in both cases [77].

Staged Parser Combinators Part II

In Part I we implemented shortcut fusion by encoding lists as either folds or unfolds, and partially evaluating function composition. This is but one half of "Specialising Parsers to Queries". In this part we study parsers, in particular parser combinators. Among many advantages described below, parser combinators are classically implemented as functions themselves. Which is ideal for partial evaluation and performance purposes. While practical parser combinator implementations handle recursive-descent parsing, we show that the techniques also apply to bottom-up parsing, in particular in the context of dynamic programming.

Part of this work has been previously published. The work on dynamic programming was done in collaboration with Thierry Coppey [42]. Eric Béguet developed a first macro-based implementation of recursive-descent parser combinators [7]. The presentation in this part is more principled and considerably simplifies the treatment of both recursion and code generation issues.

6 Staged parser combinators for data processing

Introduction

Parser combinators [87, 48, 56] 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 poor performance (see Section 6.4) inherent to their implementation. There is a heavy penalty to be paid for the expressivity 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 [84, 4]. From a software engineering standpoint, big chunks of low-level code 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.

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 (Figure 6.1).

```scala
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) }
```

Figure 6.1 – A HTTP parser implemented using parser combinators

In this example, we observe the following:

- the `status` parser looks for the code in a status message. It uses the sequencing 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 10 lines of code we have defined a parser for HTTP responses that is declarative and fairly easy to understand. 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 10 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.

Contributions

As is the theme in this dissertation, the answer to performance with parser combinators lies in judiciously using partial evaluation. By doing so, we create 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) [69]. 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 6.3).

- 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 6.3.2).

- 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. (Section 6.4).

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.

In a parser combinator library, parsers are represented as functions from some input to a parse result. Complex parsers are created from simpler ones through the use of special operators, called combinators. They essentially combine parsers through function composition. Figure 6.2 shows an interface for parser combinators in Scala. The implementation is contained inside a `Parsers` module. The abstract class `Parser[T]` represents a parser that returns results of type `T`. It is a function from an abstract `Input` type to a `ParseResult` datatype. An input represents a (finite) stream yielding elements of type `Elem`. A parse result is either a `Success`, in which case it contains a value of type `T`, and the rest of the input from where to continue parsing. Or it is a `Failure`, in which case it contains the input from where we can recover. We use a helper function `mkParser` to reduce boilerplate when creating new parsers.

We distinguish three main sorts of combinators, sequencers, alternatives, and repetition.

Sequencers. The parser `lhs ~ rhs` uses the sequencing combinator `~`. It succeeds if `lhs` succeeds, *and* `rhs` succeeds on the remaining input. The result is a pair containing the left and right results, respectively (Figure 6.3.

```
trait Parsers {
  type Elem
  abstract class Input {
    def atEnd: Boolean
    def first: Elem
    def rest: Input
  }

  abstract class ParseResult[T] {
    def map[U](f: T => U): ParseResult[U] = this match {
      case Success(t, rest) => Success(f(t), rest)
      case _                => this
    }
    def flatMapWithNext[U](f: T => Input => ParseResult[U]) = this match {
      case Success(res, next) => f(res)(next)
      case _                  => this
    }
  }
  case class Success(res: T, next: Input) extends ParseResult[T]
  case class Failure(next: Input) extends ParseResult[T]

  def mkParser[T](f: Input => ParseResult[T]) = new Parser[T] {
    def apply(in: Input) = f(in)
  }
  abstract class Parser[T] extends (Input => ParseResult[T]) { ... }
}
```

Figure 6.2 – An unstaged interface for parser combinators

```
//as a method on Parser[T]
def ~[U](that: Parser[U]): Parser[(T, U)] = mkParser { in => this(in) match {
  case Success(t, rest) => that(rest) match {
    case Success(u, rest2) => Success((t, u), rest2)
    case _ => Failure(in)
  }
  case _ => Failure(in)
}}
```

Figure 6.3 – The sequencing combinator for parser combinators

Variants of this combinator are the operators <~ and ~>. The parser `lhs <~ rhs` succeeds if both the left hand side and right hand side succeed, and returns only the left result, resulting in a parser of the type of `lhs`. Analogously, `lhs ~> rhs` returns the right result.

Alternatives. The parser `lhs | rhs` uses the alternative combinator `|`, and succeeds if either of `lhs` or `rhs` succeed. Our notation follows the PEG grammar notation, which means that there is a notion of priority in parsing an alternative: parsing `rhs` is attempted only if `lhs` fails.

```
//as a method on Parser[T]
def | (that: Parser[T]): Parser[T] = mkParser { in => this(in) match {
  case s @ Success(_, _) => s
  case _ => that(in)
}
```

Figure 6.4 – The alternative combinator for parser combinators

Repetition. The parser `rep(p)` parses the underlying parser `p` repeatedly until failure. It aggregates the results of each successful parse into a list. An empty list is returned if `p` does not succeed even once. Note that our implementation of `rep` uses a tail-recursive `loop` function.

```
def rep[T](p: Parser[T]): Parser[List[T]] = {
  def loop(curIn: Input, curRes: List[T]): ParseResult[T] = {
    p(curIn) match {
      case Success(t, rest) => loop(rest, curRes.append(t))
      case Failure(_)       => Success(curRes, curIn)
    }
  }
  mkParser[List[T]] { in => loop(in, Nil) }
}
```

Figure 6.5 – The `rep` combinator for parser combinators

We could have implemented repetition using sequencers and alternatives too:

```
def success[T](t: T): Parser[T] = mkParser(in => Success(t, in))
def rep[T](p: Parser[T]) =
 ((p ~ rep(p)) map { (x, xs) => x :: xs }) | success(Nil)
```

The `success` combinator creates successful parse results without consuming input, and the `map` combinator transforms results of a parser. This implementation uses recursion at the grammar level. We however prefer to treat repetition as full first-class citizen in the library, because this enables us to treat such parsers as list-like recursion schemes. We refer to Section 8.2 for more details.

Common variants of the repetition combinator are `repN(p, min, max)`, which succeeds if `p` succeeds between `min` and `max` times, and `repsep(p, q)`, which repeatedly parses occurrences of `p`, interleaved with occurrences of `q`. The latter is useful, for example, for parsing comma-separated values, where a comma parser takes the place of `q`.

Transforming a parse result. The above three combinators allow creating more complex parsers based on simple ones. In addition we also have a map combinator which we use to transform a parse result. This combinator is also commonly known as a means to specify a semantic action. In particular, we will use this combinator to perform a query on data resulting from a successful parse.

```
//as a method on Parser[T]
def map[U](f: T => U): Parser[U] = mkParser { in => this(in) map f }
```

Figure 6.6 – The map combinator for parser combinators

Context-dependent parsing. The flatMap combinator enables the creation of a parser based on the result of a previous parser, effectively enabling context-dependence. It is used, for instance, in defining the header and respWithPayload from the above HTTP example.

```
//as a method on Parser[T]
def flatMap[U](f: T => Parser[U]): Parser[U] = mkParser { in =>
  this(in) flatMapWithNext f
}
```

Figure 6.7 – The flatMap combinator for parser combinators

Note that we can re-implement all sequencing combinators using flatMap. This gives us a much more intuitive for-comprehension notation:

```
//as methods on Parser[T]
def ~[U](that: Parser[U]): Parser[(T, U)] = for (l <- this; r <- that) yield (l, r)
def ~>[U](that: Parser[U]): Parser[U]     = for (_ <- this; r <- that) yield r
def <~[U](that: Parser[U]): Parser[T]     = for (l <- this; _ <- that) yield l
```

Figure 6.8 – The sequencing combinators for parser combinators, using for-comprehensions

Base parsers. As mentioned, we can construct complex parsers from simpler ones, but we still need to create the simplest possible parsers. The simplest one, acceptIf, accepts an element of type Elem if it passes a predicate. We can then define a helper function accept for single elements (Figure 6.9).

In theory, any parser can be built from acceptIf. In practice, and mainly for performance reasons, we define additional base parsers for handling token parsers for identifiers and string literals.

```
def acceptIf(p: Elem => Boolean): Parser[Elem] = mkParser { in =>
  if (!in.atEnd && p(in.first)) Success(in.first, in.rest)
  else                          Failure(in)
}
def accept(e: Elem) = acceptIf { (x: Elem) => x == e }
```

Figure 6.9 – Base parsers `acceptIf` and `accept`

Recognisers. If a parser succeeds on a certain input and does not return any value, we refer to it as a recogniser. They have the type `Parser[Unit]`, since success or failure is already captured in the `ParseResult` datatype. For example, a recogniser for `acceptIf` is defined as follows:

```
def acceptIfReco(p: Elem => Boolean): Parser[Unit] = mkParser { in =>
  if (!in.atEnd && p(in.first)) Success((), in.rest)
  else                          Failure(in)
}
```

Figure 6.10 – Base recogniser for `acceptIf`

A recogniser for a string literal would succeed by verifying that the input is a sequence of characters wrapped in quotes, while a parser for the same would return the literal in question.

The overhead of abstraction

Similar to overhead incurred by products, sums and list pipelines, this functional parser combinator implementation also suffers from poor performance:

- 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 plug- gable VM, such as Truffle/Graal [92], 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.

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 this penalty.

Staging Parser Combinators

As mentioned before, in a parser combinator library parsers are represented as functions from and input to a parse result. Moreover, our guiding principles tell us that partially evaluating function composition effectively removes intermediate structures. Based on the above, we can systematically add `Rep` types to the parser combinator API:

```
abstract class Parser[T] extends (Rep[Input] => CPSParseResult[T]) {
  def map[U](f: Rep[T] => Rep[U]): Parser[U]
  def flatMap[U](f: Rep[T] => Parser[U]): Parser[U]

  def ~[U](that: Parser[U]): Parser[(T, U)]
  def ~>[U](that: Parser[U]): Parser[U]
  def <~[U](that: Parser[U]): Parser[T]

  def |[T](that: Parser[U]): Parser[U]

}
```

Figure 6.11 – A staged interface for parser combinators

Note in particular the signature of `flatMap`, which takes a function that returns a `Parser` (instead of `Rep[Parser]`). This is indeed analogous to the staged `flatMap` counterparts for lists and streams: a staged parser is already a code generator (i.e. a staged function), so the semantics and flow are preserved. The implementations of these remain *exactly the same* to their unstaged counterparts, since they redirect to the corresponding implementations on parse results.

Church-encoded parse results

The reader will have noticed the use of `CPSParseResult`, which seems natural in a partial evaluation context. A parse result is a glorified `Option`, therefore Church-encoding is fairly straightforward (Figure 6.12). A parse result waits for two continuations, `success` and `failure`. One of these is called based on whether we create a successful of failing result. The `map` and `flatMap` methods follow directly from their non-CPS counterparts, and do not require much description. For completeness sake we provide their implementation in Figure 6.13.

```
abstract class CPSParseResult[T] { self =>
  def apply[X](
    success: (Rep[T], Rep[Input]) => Rep[X],
    failure: Rep[Input] => Rep[X]
  ): Rep[X]

  def mkSuccess[T](t: Rep[T], next: Rep[Input]) = new CPSParseResult[T] {
    def apply[X](
      success: (Rep[T], Rep[Input]) => Rep[X],
      failure: (Rep[Input]) => Rep[X]
    ): Rep[X] = success(t, next)
  }

  def mkFailure[T](next: Rep[Input]) = new CPSParseResult[T] {
    def apply[X](
      success: (Rep[T], Rep[Input]) => Rep[X],
      failure: (Rep[Input]) => Rep[X]
    ): Rep[X] = failure(next)
  }
}
```

Figure 6.12 – An interface for Church-encoded parse results

```
//as methods on CPSParseResult
def map[U](f: Rep[T] => Rep[U]) = new CPSParseResult[U] {
  def apply[X](
    success: (Rep[U], Rep[Input]) => Rep[X],
    failure: Rep[Input] => Rep[X]
  ): Rep[X] = self.apply(
    (t: Rep[T], in: Rep[Input]) => success(f(t), in),
    failure
  )
}

def flatMapWithNext[U](f: (Rep[T], Rep[Input]) => CPSParseResult[U]) = {
  new CPSParseResult[U] {
    def apply[X](
      success: (Rep[U], Rep[Input]) => Rep[X],
      failure: Rep[Input] => Rep[X]
    ): Rep[X] = self.apply(
      (t: Rep[T], in: Rep[Input]) => f(t, in).apply(success, failure),
      failure
    )
  }
}
```

Figure 6.13 – An implementation of `CPSParseResult`

As with `EitherCPS` we once again specialise conditional expressions for parse results by overriding the virtualised `__ifThenElse` method. We fully benefit from customised variable creation as well (Figure 6.14).

```
def __ifThenElse[T](
    cond: Rep[Boolean],
    thenp: => CPSParseResult[T],
    elsep: => CPSParseResult[T]) = new CPSParseResult[T] {
  def apply[X](
      success: (Rep[T], Rep[Input]) => Rep[X],
      failure: Rep[Input] => Rep[X]): Rep[X]): Rep[X] = {

    var tmp = zeroVal[CPSParseResult[T]]
    val successK = (t: Rep[T], rest: Rep[Input]) => { tmp = mkSuccess(t, rest) }
    val failK = (rest: Rep[Input]) => { tmp = mkFailure(rest) }

    if (cond) thenp.apply(successK, failK)
    else      elsep.apply(successK, failK)

    if (tmp.isSuccess) success(tmp.value, tmp.rdr) else failure(tmp.rdr)
  }
}
```

Figure 6.14 – Specialised conditional expressions for `CPSParseResult`

In addition, we add a `orElse` method that simplifies the implementation of the alternative combinator for parser combinators: if the left result is a success it is returned, otherwise the right result is returned. This is the trickiest function to implement (Figure 6.15). There are two join points involved here, first with the result on the left, and then with the potential result on the right. Hence the need to not only use temporary variables for the left part, but also to use the customised conditional expressions for the right hand side, and to call the `apply` method at the end.

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 would be unfolded and inlined during staging time. We are forced to stop the recursion by generating a staged function for recursive parsers. This is done by explicitly declaring a parser needs using the `rec` combinator. This combinator performs the lifting as necessary. It works using the classical memoization scheme. The first time it sees a parser, it stores, in a static map, 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.

Consider for instance a simple `digitAdder` parser that reads a sequence of digits and maintains

```
def orElse(that: => CPSParseResult[T]) = new CPSParseResult[T] {
  def apply[X](
      success: (Rep[T], Rep[Input]) => Rep[X],
      failure: Rep[Input] => Rep[X]): Rep[X] =  {

    var tmp = zeroVal[CPSParseResult]
    val successK = (t: Rep[T], rest: Rep[Input]) => { tmp = mkSuccess(t, rest) }
    val failK = (rest: Rep[Input]) => { tmp = mkFailure(rest) }
    self.apply(successK, failK)

    (if (tmp.isSuccess) mkSuccess(tmp.value, tmp.rdr) else that).apply(success, failure)
  }
}
```

Figure 6.15 – An implementation of `CPSParseResult`

a running sum:

```
def adder: Parser[Int] =
  (digit2Int ~ adder) map { case (x, y) => x + y } | digit2Int
```

In a staged setting, we wrap the definition around a `rec` combinator:

```
lazy val adder: Parser[Int] = rec {
  (digit2Int ~ adder) map { case (x, y) => x + y } | digit2Int
}
```

Note that we use lazy evaluation to bind the instance of `adder` properly: when the implementation of `rec` sees `adder` for the second time it will know to generate a function call rather than define a new function. The definition of `rec` is given in Figure 6.16. It is implemented at the IR level (in the `Exp` world). When a parser is seen for the first time, a new symbol for a staged function `Rep[Input => ParseResult[T]]` is created. Since this is a function boundary, we must materialise the parse result, and cannot use Church-encodings here, hence the call to `toParseResult`. The `doLambdaDef` function reifies an unstaged function: it converts a `Rep[T] => Rep[U]` to a `Rep[T => U]`, and is available by mixing in the `FunctionsExp` trait. The symbol and its corresponding definition are stored in a cache, and reused when the parser `p` is seen again.

Evaluation

Our evaluation of staged parser combinators attempted to answer two questions: how much does staging itself impact performance, and how does a fully staged version compare to hand-optimised parsers. We ran our Scala/Java benchmarks using Scalameter [66], a benchmarking and performance regression testing framework for the JVM. This framework handles JIT warm-ups as well as running a benchmark until performance stabilizes.

We answered the first question by taking a profiler-based approach on a simple parser that

```
trait StagedParsersExp extends StagedParsers ... with FunctionsExp {
  val store = new scala.collection.mutable.HashMap[Parser[_], Sym[_]]

  def rec[T](p: Parser[T]) = Parser[T] { in =>
    val myFun: Rep[Input => ParseResult[T]] = store.get(p) match {
      case Some(f) =>
        val realf = f.asInstanceOf[Exp[Input => ParseResult[T]]]
        realf

      case None =>
        val funSym = fresh[Input => ParseResult[T]]

        store += (p -> funSym)
        val f = p.toParseResult
        val g = createDefinition(funSym, doLambdaDef(f))
        store -= p

        funSym
    }

    val res: Rep[ParseResult[T]] = myFun(in)
    if (res.isEmpty) mkFailure(res.next),
    else             mkSuccess(res.get, res.next)
  }
}
```

Figure 6.16 – The rec combinator for recursion on staged parser combinators

parses a sequence of comma-separated boolean values. Its parser is given by:

```
 def p = (accept('[') ~>
    repsep((accept("true") | accept("false")), skipWs(accept(',')))
  <~ accept(']'))
```

We started out with a naive parser combinator library, and progressively added staging for various data types until we got results within range of a hand-written recursive version of the same parser. We added the optimisations based on the data type whose instances were the highest, based on information from the profiler. Here are the optimisations, in order:

A mutable collection to fold into. An *order of magnitude* in performance was obtained by using a mutable list buffer to gather results rather than immutable lists in the implementation of rep and repsep. This seems obvious in hindsight. Yet the standard library use immutable lists, and also implements rep using a right fold, i.e. it first accumulates closures. With this simple change, the standard library combinators would perform much better. A fair baseline is therefore a parser combinator library that “does the reasonable thing” for repetition parsers. For this benchmark, despite using mutable data structures we are still *4-5x slower* than the hand-optimised version.

Staged parsers and Church-encoded parse results. The data structure that appears most often next is the parse result. Indeed, an instance is created for every simple parser. By staging these (which naturally implies staging parsers themselves) we get to within *1.5x* of the hand-optimised version.

Staged input readers. We finally stage the input reader as well. Since we know at compile time what structure we will be parsing, this amounts to specialising the reader for the parser. In this benchmark we load an array of characters in memory and use that as an input. This final optimisation even *surpasses* the hand-written recursive implementation. We are now *1.5x faster*. We suspect this is because the staged implementation generates a while loop that seems to have a better shape for the JVM to further optimise.

We answered the second question by benchmarking a HTTP and a JSON parser. 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.

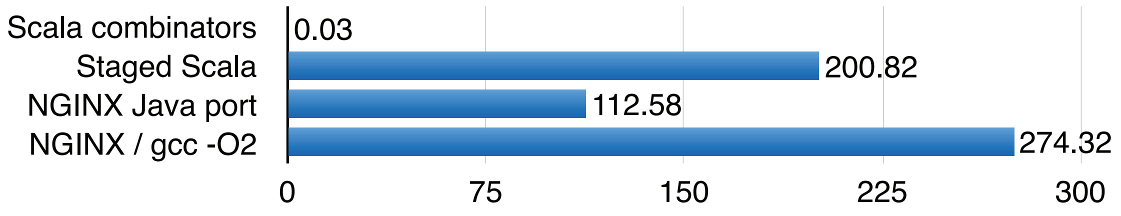


Figure 6.17 – HTTP parser throughput in MB/s

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 6.17 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 faster than staged parser combinators. We presume that the O2 compiler optimizes switch/case statements efficiently.

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

  value
}
```

Figure 6.18 – A JSON parser using parser combinators

JSON parser Our second evaluation (Figure 6.19) compares parsing of the JSON payload of the previous messages. The JSON grammar is given in Figure 6.18. For simplicity sake we have omitted the `accept` and `acceptIf` wrappers around characters and strings. This is achievable in real code using the implicit classes (colloquially known as the pimp my library pattern). This JSON parser looks very similar to a standard parser combinator implementation [59, 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 [16]. 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.

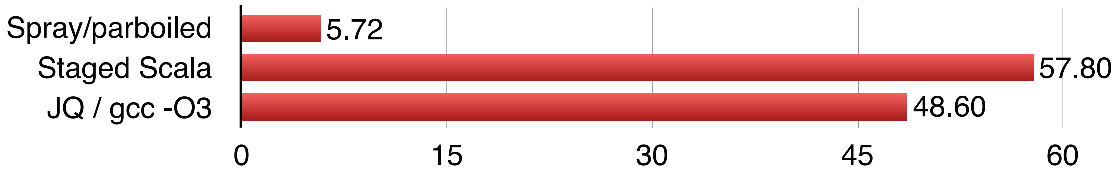


Figure 6.19 – JSON parser throughput in MB/s

Related work

Tools for parsing Parser combinators and their implementations are popular in functional programming [22, 37, 87]. Their original implementations led to mainstream libraries, like

Parsec [48] in Haskell and the Scala parser combinator library [56]. These libraries focus on producing a single result. Koopman et al [47] use a continuation-based approach to eliminate intermediate list creation.

On the other hand are parser generators like Yacc [39], Antlr [61] and Happy [30]. 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.

Metaprogramming and compiler technology Staged parser combinators are a specific instance of partial evaluation [24]. We make use of the well known technique of multi-stage programming [85]. Sperber et al. also use partial evaluation for optimizing LR parsers which are implemented as a functional-style library [78].

7 Dynamic programming on sequences with staged parser combinators

Introduction

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) [27], can be used to describe sequence structures as grammars in a parser combinator library.

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.

In this chapter we present a staged parser combinator library for algebraic dynamic programming. As expected, staging allows the removal of intermediate data structures. Staging also allows us to generate code for various types of hardware. The added value of this chapter is in using this feature to generate GPU code in order to perform dynamic programming in parallel:

- 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 7.3.2, 7.3, 7.3.1).

- Imposing a bottom-up order also opens up the possibility to parallelize processing, as independent intermediate results can be computed independently [75, 79]. Our use of staging coupled with generative programming enables us to generate code that runs on the GPU for dynamic programs (Section 7.4). The GPU code computes an optimal cost function efficiently. In Section 7.5, 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 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.

Algebraic Dynamic Programming

Consider the following classic dynamic programming example: 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. 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:

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

Figure 7.1 – A recurrence relation describing the matrix multiplication problem

We can memoize intermediate results in a matrix, and look them up when we need them. The recurrence relations above do not capture the *structure* of the problem in an intuitive manner though. We can express the problem as a grammar instead, as shown in Figure 7.2. 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.

Note that we have not shown the implementations of `mult`, `single` and `h`. Indeed, they are customisable. 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.

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

Figure 7.2 – The matrix multiplication problem as a grammar

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

Based on the above, the matrix-chain problem has the following signature:

```
trait MatMultSig {
  type A, S
  def single(a: A): S
  def mult(l: S, r: S): S
  def h(xs: List[S]): List[S]
}
```

Figure 7.3 – A signature for the matrix-chain multiplication problem

In the above, the abstract types `A, S` designate the input alphabet and sort symbol, respectively. The `single` and `mult` methods correspond to operators on the signature, and `h` is the aggregation function.

An implementation of a signature and two algebras for the matrix-chain multiplication problem are given in Figure 7.3. In addition to the aggregation function `h`, we define two operations, `single` representing a single matrix and `mult` representing the multiplication of two matrices. Note that the arguments to `mult` are of the generic sort type `S`.

This signature can spawn many algebras, of which we show two interesting ones here. The first one is the cost algebra, which seeks to minimize the cost of multiplying matrices. The input alphabet is a pair of integers representing the dimensions of the matrix, and the output is a triple of integers, for the rows, optimal cost and columns of the final result, respectively. As with the recurrence relation above (Figure 7.1) the cost of a `single` matrix is 0, while multiplying two matrices (the `mult` function) has an associated non-zero cost. The crucial part is the aggregation function, which selects the result with the minimal cost among a list of results.

```
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))
}
```

Figure 7.4 – An algebra computing the optimal cost for a chain of matrices

Another useful algebra prints all possible configurations. In this case the sort is just a `String`, and the implementations of `single` and `mult` print the matrix dimensions, with brackets as appropriate. The aggregation function simply returns all elements it receives as input.

```
trait PrettyPrintAlgebra 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
}
```

Figure 7.5 – An algebra printing various chaining alternatives

We already saw the grammar describing the chain above. It is in fact contained in a trait that mixes the signature `MatMultSig` and `ADPParsers`, which provide a parser combinator library for these grammars (Figure 7.6). The `MatMult` object ties a grammar to a *concrete* algebra to a grammar using mix-in composition.

A staged parser combinator library

Algebraic Dynamic Programming enables a modular way to express semantic operations on a (potentially) ambiguous grammar. As we have seen in the previous section, this can be easily reworked into an executable program. We have however only hinted at how the parsers themselves are implemented. In this section we present an implementation for these. Since by this point, the reader will be familiar with staging techniques, we add `Rep` types directly as well.

```
trait MatMultGrammar extends ADPParsers with MatMultSig {
  def chain = tabulate((
    singleMatrix map single
  | (chain ~ chain) map mult
  ) aggregate h)
}
object MatMult extends MatMultGrammar with CostAlgebra
```

Figure 7.6 – A grammar for matrix chain multiplication, and tying an algebra to this grammar

Note that the aggregation function `h` operates on a set of results, not a single one. Indeed with ambiguous grammars it is possible to build multiple parse trees for a single input sequence. Hence parsers now become functions from an input to a list of results. Figure 7.7 provides an interface for these parsers. The `ADPParsers` trait has an abstract `input` sequence that will be parsed. A parser is tasked with computing all possible results over a subsequence of this input. Hence it is a function from a pair of integers (representing the beginning and end of the subsequence) to a list of results. We use `CPSList`: this is in line with grammar specifications. Since we do not know at grammar construction time what algebra will be applied, we do not need to create intermediate lists.

```
trait ADPParsers extends CPSLists with ... {
  type Elem
  abstract class Input(val source: Rep[List[Elem]]) {
    def elemAt(i: Rep[Int]): Elem
  }
  def input: Input
  abstract class Parser[T] extends ((Rep[Int], Rep[Int]) => CPSList[T]) { ... }

  def mkParser[T](f: (Rep[Int], Rep[Int]) => CPSList[T]) = new Parser[T] {
    def apply(i: Rep[Int], j: Rep[Int]) = f(i, j)
  }
}
```

Figure 7.7 – An interface for ADP-based parser combinators

As with recursive-descent parsers, we implement combinators for sequencing, alternation, and transforming of results. With lists, sequencing corresponds to a cross-product of all possible pairings resulting from the left and the right parsers, and alternation corresponds to list concatenation. In addition to these we provide a combinator for filtering, and one for aggregating results. The latter simplifies the use of the aggregation function `h` from a signature.

A basic parser is created using the `el` combinator, which is analogous to `acceptIf`. It returns a single element if the indices have the correct interval (Figure 7.9).

```
//as methods on Parser
def ~[U](that: Parser[U]) = mkParser[(T, U)] { (i, j) =>
  if (i < j) for {
    k <- fromRange(i, j - 1)
    x <- this(i, j)
    y <- that(k, j)
  } yield (x, y)
  else mkCPSNil()
}
def | (that: Parser[T]) = mkParser[T] { (i, j) => this(i, j) ++ that(i, j) }

def map[U](f: Rep[T] => Rep[U]) = mkParser[U] { (i, j) => this(i, j) map f }
def filter(p: (Rep[Int], Rep[Int]) => Rep[Boolean]) = mkParser[T] { (i, j) =>
  if (p(i, j)) this(i, j) else mkCPSNil()
}
def aggregate(h: CPSList[T] => CPSList[T]) = mkParser[T] { (i, j) => h(this(i, j)) }
```

Figure 7.8 – Main combinators for ADP-based parsers

```
def el = mkParser[Elem] { (i, j) =>
  if (i + 1 == j) mkCPSList(input.elemAt(i))
  else            mkCPSNil()
}
```

Figure 7.9 – An implementation of parser combinators for dynamic programming

Staging and memoisation

With recursive-descent parsers, we introduced the rec combinator for handling recursion. The `tabulate` combinator plays an analogous role for memoisation 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 (Figure 7.10).

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

Figure 7.10 – The `tabulate` combinator, for memoisation

Top-down to bottom-up evaluation

Running the `chain` parser 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 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]): Unit = {
  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
    }
  }
}
```

Figure 7.11 – Processing a parser in bottom-up order

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.

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, 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 Smith-Waterman: we want to find the best alignment of *two* sequences. While ADP can support dynamic programs on multiple sequences [75], state of the art implementations for sequence alignment face challenges that are orthogonal to its representation as a grammar. Not only does 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 [74, 73].

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

Mutually recursive parsers

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 7.12), 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).

GPU parallelisation

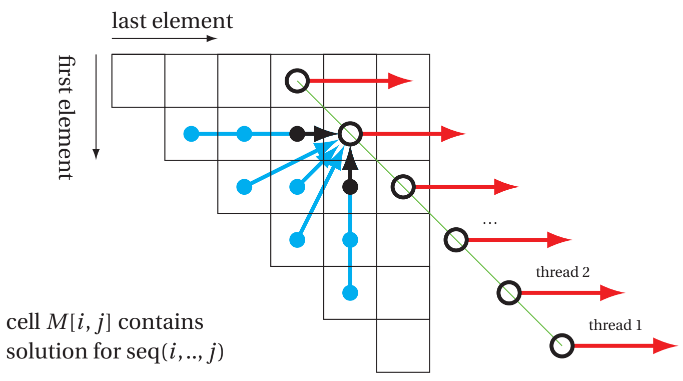


Figure 7.12 – Threads progress jointly along matrix diagonal. Dependencies can be reduced to immediately preceding matrix cells.

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 divergence 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

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

power of GPUs outperforms the memory bandwidth such that (global) memory access 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 [93].

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.

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 7.13 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 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 7.6), 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.

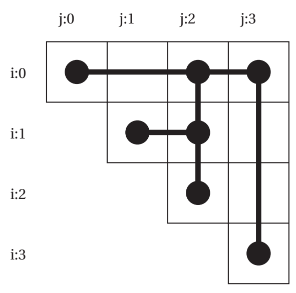
Therefore, at every cell, we store a pair `(ruleId, concats)`, where `ruleId` corresponds to a chosen alternative, and `concats` is a list of splits due to concatenation. We attribute a `ruleId` for every alternative in a `tabulate` combinator in a prior grammar analysis phase.

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 7.13 depicts this representation.

Matrices:
$A : 3 \times 2$
$B : 2 \times 4$
$C : 4 \times 2$
$D : 2 \times 5$

Solution:
$(A \cdot (B \cdot C)) \cdot D$



Backtrace:
(i, j), (rule, concats)

(3,3), (`singleMatrix`, []),
(2,2), (`singleMatrix`, []),
(1,1), (`singleMatrix`, []),
(1,2), (`chain`, [2]),
(0,0), (`singleMatrix`, []),
(0,2), (`chain`, [1]),
(0,3), (`chain`, [3])

Figure 7.13 – A backtrace for an instance of the matrix multiplication problem

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 [75, 36]. 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 ($O(n^2)$).

Evaluation

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 [38, 35]. For both algorithms, we generate C code from staged parser combinators. For parallelisation, we generate CUDA, additionally.

The Nussinov algorithm

The Nussinov algorithm maximises the number of matching basepairs. Its running time complexity is $O(n^3)$. Its grammar is given in Figure 7.14. There are 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.

```
def 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 7.14 – A grammar for the Nussinov sequence alignment problem

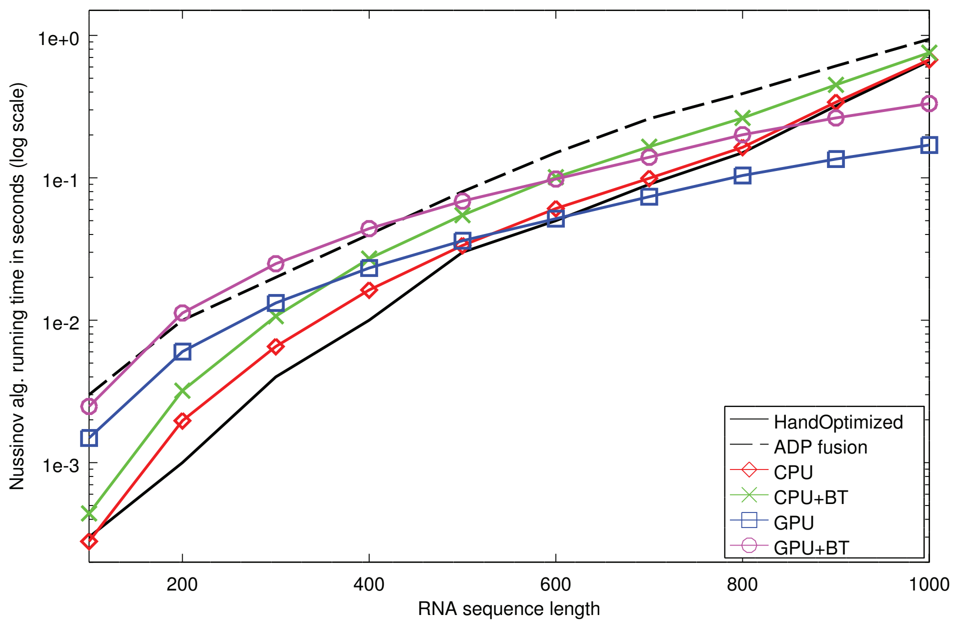


Figure 7.15 – Nussinov algorithm running time

Our evaluation results are displayed in Figure 7.15. We show the running time for four variants of our generated code, along with a hand optimised C version, and an ADPfusion version:

- ADPfusion is an embedded Haskell DSL for ADP. It uses stream fusion [15] to eliminate intermediate list creation, and performs close to hand-optimised 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-optimised C code. Indeed, manual inspection of the generated code reveals very close implementations. The ADPfusion version is about 2× slower, which is in line with some remaining overhead that stream fusion is unable to completely eliminate [36, 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 parallelising 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 maximising the number of basepairs, it minimises free energy. Because the free energy is computed from hundreds 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 7.16, 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 [35], a highly optimised 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 optimisations.

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 7.16 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 7.17). The Nussinov algorithm scales better on GPU, offering speedups from 4× to 40× (respectively for 1000 and 5000 elements sequences). The Zuker lookup tables hamper GPU performance more

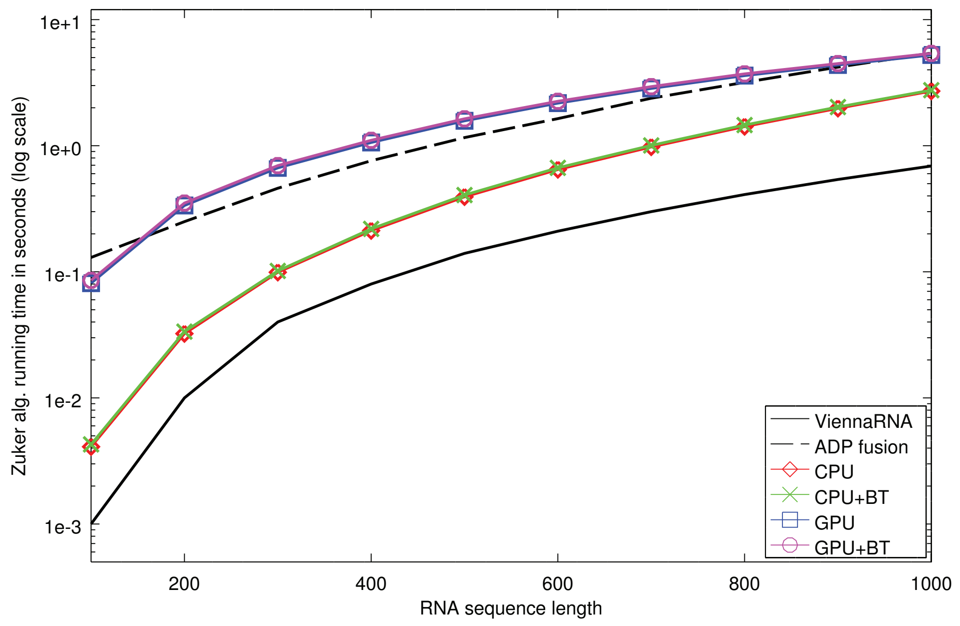


Figure 7.16 – Zuker algorithm running time

significantly, as multiple random memory accesses are required for each scoring, thereby delivering at best 3.3× speedup on the CPU for this algorithm.

Related work

Parser combinators and memoization are common knowledge. Frost et al. introduce this technique [23, 21]. In libraries, techniques like packrat parsing [20] are supported. They are also known to support left recursion [88].

Our work on dynamic programming parsers is inspired by Algebraic Dynamic Programming [27]. The original implementation was a Haskell library, and later a external DSL known as Bellman's GAP was developed [75]. Bellman's GAP also includes analysis techniques for verifying some soundness properties, and optimizing memory consumption and running time through yield-size analysis [28]. 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 resulting from the optimal cost (forward) phase. Both these approaches

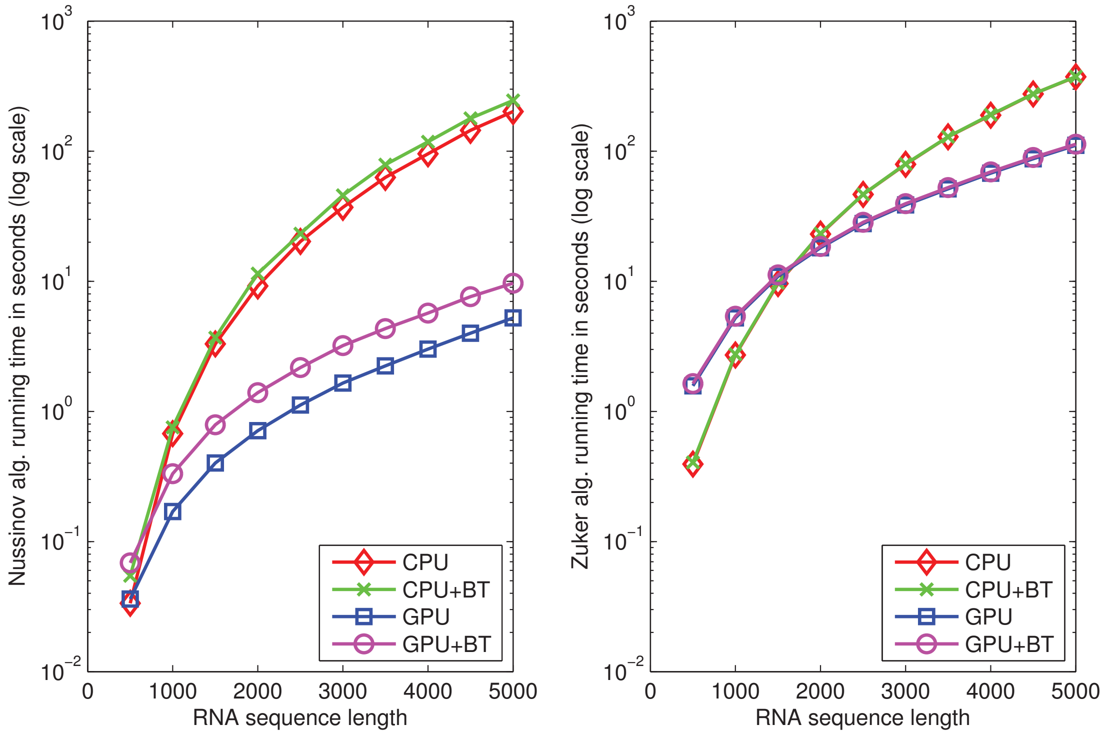


Figure 7.17 – Nussinov and Zuker algorithms scalability

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 [83] (programs are expressed using classic recurrences) and Dyna [18] (a logic-programming style language prevalent in the field of NLP/stochastic grammar parsing).

Cartey et al propose an intermediate DSL for recurrence relations, which they analyze and generate GPU programs from [11]. 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 [79]. 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 [91, 13].

Much work on parallelizing dynamic programming has focused on the Smith-Waterman sequence alignment problem [50]. CudAlign [72, 73] matches sequences much larger than the GPU memory size, using a hybrid divide-and-conquer and dynamic programming approach. We have focused on folding problems rather than alignment problems.

Specialising Parsers for Queries Part III

In Parts I and II we implemented staging-based libraries for list operations and parsers which are rid of abstraction overhead, and therefore exhibit close to hand-written performance. In this part we bring these two libraries together to attack another source of performance overhead, namely the creation of intermediate and unwanted parse results.

8 Specialising parsers for queries

Introduction

Suppose we are interested in analysing contributions and revisions made by users on Wikipedia. Before performing any of the core data analysis, our first task is to curate, or preprocess, full Wikipedia data to select only what we need, and load that into a database so that we can move on to our main task.

The classic way of performing the preprocessing task is to first partition the data into multiple chunks and then do preprocessing on a cluster of machines. If we were using Scala, we would opt for Apache Spark for distributing the work. The core program reads an XML file (the format in which the Wikipedia dumps are created), parses it, runs a query that selects data we need, and dumps this selected data into a database, as in Figure 8.1.

```
case class WikipediaPage(title: String,
  redirectTitle: String,
  timestamp: String,
  lastContributorUsername: String,
  text: String)

def parseXMLtoWikipediaPage(xml: String): WikipediaPage = {
  val elem = scala.xml.XML.loadString(xml)
  new WikipediaPage(
    (elem \ "title").text,
    (elem \ "redirect" \ "@title").text,
    (elem \ "revision" \ "timestamp").text,
    (elem \ "revision" \ "contributor" \ "username").text,
    (elem \ "revision" \ "text").text.replaceAll("\n", " ")
  )
}
val wikipediaDataFrame = sc.textFile(wikiDumpFile).map(parseXMLtoWikipediaPage)
```

Figure 8.1 – Preprocessing XML metadata

In this example, the `parseXMLtoWikipediaPage` function is a query over an XML file. We first parse the `xml` string using a standard XML parsing library, and use an XPath-like language for the query itself. The query succeeds because we *know* the schema for Wikipedia dumps.

Yet we could do better. The problem with the above code is that we iterate over the same data twice: once for parsing it, and the second time for preprocessing. What is worse, we parse a lot of data that we potentially ignore later on. The above query only shows the portion that we are interested in. The schema for Wikipedia's metadata contains many other fields and subfields. In short, we waste both time and space, and all this before getting to the meat of the data analysis itself. What if we could preprocess the data, and skip over unnecessary parts *while* we parse it?

Indeed, we could write a specialised version of the XML parser that skips tags we do not require. Of course, doing so by hand is tedious, error-prone, and non-scalable. We could instead use an event-based (SAX) parser. This allows us to specify actions to perform when seeing certain types of tags. In pseudocode, the implementation would look like the following:

```
def onTagSeen(tagName: String, beginPos: Int, endPos: Int) = {
  if (tagName == "title") collectTitle(beginPos, endPos)
  else if (... other tags we want to collect ...) ...
  else skip (default case)
}
```

While this is already a huge improvement on writing such specialised parsers by hand, we have lost the declarative and parser-agnostic nature of the query: we are forced to think of a query in terms the underlying parser that reads data in.

Even high-level functional programming tools like parser combinators do not escape this issue. We can write a well-typed parser for the Wikipedia schema, i.e. `Parser[WikipediaPage]`. To do so, we write a parser for the general schema, and then insert skipping combinators at places where we do not need the result. Essentially we end up writing two parsers, one for the general Wikipedia metadata, and one for parsing `WikipediaPage`.

In this chapter, we describe a technique that takes a parser (implemented using a parser combinator library), and a query on the parsed structure, and returns a new parser that is specialised to the query in question. Our technique applies to queries on nested, non-recursive data structures; many prevalent and popular raw data sources adhere to such schemas. This technique hinges on two insights:

- Converting a parser to a recogniser drastically decreases both running time and memory consumption.
- Treating repetition parsers as folds over parsers allows us to express query-like operations on these folds.

Contributions

- We describe a transformation from a tree representing a parser for a data format to a specialised parser for a given query (Section 8.3). We treat queries that are a combination of the selection, projection and aggregation operators on a nested relational algebra. The transformation includes the conversion of skipped parsers to recognisers.
- The transformation relies on an efficient representation of repetition parsers. We can treat repetition parsers as (Church-encoded) lists (Section 8.2), and therefore build a list-like API for these. By composing over repetition parsers we push query operations directly to the point where a single element is parsed, and we can therefore process or discard it very early.
- Our transformation is semantics preserving: if a non-transformed parser succeeds on an input, a transformed parser succeeds with the same result (Section 8.3.1). The correctness argument relies on correct implementations of recognisers for base parsers. We also discuss optimisation properties of the transformation (Section 8.3.2).
- For meaningful performance improvements, the transformation by itself is not enough. We must have efficient implementations of both parser combinators and list operation pipelines. In the previous chapters we have seen precisely how to achieve these partial evaluation based implementations. The transformation presented in this chapter sits atop a Scala macro version of a staged parser combinator library (Section 8.4).
- We evaluate the performance of our transformed parsers (Section 8.5). Our benchmarks evaluate various degrees of parsing vs. recognising over a sequence of large tuples, from full recognisers to full parsers. As the size of the data increases, parsing everything stresses memory, to the point where long garbage cycles and swapping is required. This translates into running times that get relatively worse as the data size increases.

We discuss our implementation choices in Section 8.6 and related work in Section 8.7.

Repetition parsers as lists

Recall the parser combinator implementation presented in Figure 6.5: the classic repetition combinator accumulates results into a list. Often, we are interested in performing some semantic action on the resulting list. Consider for instance `personParser`, which parses information for a person, represented by their first name, last name, and age:

```
case class Person(first: String, last: String, age: Int)
def personParser: Parser[Person] = ...
```

If we wish to only count the number of people, we can take the length of the resulting list:

```
def numAdults: Parser[Int] = rep(personParser).map(_.length)
```

If we were instead interested in the first names of people who are adults, we would provide the following semantic function:

```
def namesOfAdults: Parser[String] = rep(personParser) map { ls =>
  ls.filter(_.age >= 18).map(_.first)
}
```

For both these queries we first create an intermediate data structure `ls` which we subsequently process. In this section we show that the creation of `ls` can be avoided altogether by introducing a more general repetition combinator `repFold`. Such combinators are usually present in many combinator libraries. We then abstract over `repFold` so that repetition parsers can themselves be treated as lists. As a result we can compose over repetition parsers using operations similar to those on traditional lists.

The `repFold` combinator. The repetition combinator `rep` folds its results into a list. The choice of returning a list is an arbitrary one. We could have chosen any other foldable structure. We can generalise this idea to create a `repFold` function as follows:

```
def repFold[T, R](p: Parser[T])(z: R, combine: (R, T) => R): Parser[R] = {
  def loop(curIn: Input, curRes: R): ParseResult[R] = {
    p(curIn) match {
      case Success(t, rest) => loop(rest, combine(curRes, t))
      case Failure(_)       => Success(curRes, curIn)
    }
  }
  mkParser[R] { in => loop(in, z) }
}
```

Figure 8.2 – The `repFold` combinator

This combinator is parametric over the type of the input parser, as well as the type of the result. It takes an initial element `z`, which is the base value for the repetition parser. Successful parses of `p` are combined with the accumulated result via the `combine` function. The traditional `rep` function can be expressed using `repFold`; the initial element `z` is the empty list, and `combine` appends an element to the accumulator:

```
def rep[T](p: Parser[T]): Parser[List[T]] = {
  repFold(p)(Nil, (acc, elem) => acc.append(elem))
}
```

Figure 8.3 – The `rep` combinator implemented using `repFold`

CPS-encoded lists, all over again. Essentially, `repFold` is just like the `foldLeft` function on lists, the difference being that it folds over a parser, and eventually returns another parser. The similarity is clearer when we look at both function signatures side by side:

```
def repFold[T, R](p: Parser[T])(z: R, combine: (R, T) => R): Parser[R]
def foldLeft[T, R](ls: List[T])(z: R, combine: (R, T) => R): R
```

We have indeed recovered a particular form of `CPSList` (see Section 4.2), `CPSListParser`:

```
type CPSListParser[T, R] = (R, (R, T) => R) => Parser[R]
```

The difference between a `CPSList` and a `CPSListParser` lies only in the eventual return type. Since parser combinators satisfy the functor laws [87], we can create a more generic type that accomodates for both, and build a list-like API over it:

```
type CPSListF[T, R, F[_]] = (R, (R, T) => R) => F[R]
```

The extra higher-kind parameter `F` represents the functor that wraps the eventual result value. For the classic `foldLeft` this is the identity functor, whereas in `repFold`'s case it is `Parser`.

```
type Combine[T, R] = (R, T) => R
abstract class CPSListF[T, F[_]: Functor] {
  private def self = this
  def fold[R](z: R, combine: Combine[T, R]): F[R]

  def map[U](f: T => U) = new CPSListF[U, F] {
    def fold[R](z: R, combine: Combine[U, R]): F[R] = self.fold(
      z,
      (acc: R, elem: T) => combine(acc, f(elem))
    )
  }

  def filter(p: T => Boolean) = new CPSListF[T, F] {
    def fold[R](z: R, combine: Combine[T, R]) = self.fold(
      z,
      (acc: R, elem: T) => if (p(elem)) combine(acc, elem) else acc
    )
  }

  def length: F[Int] = fold(0, (acc, elem) => acc + elem)
  def toList: F[List[T]] = fold(Nil, (acc, elem) => acc.append(elem))
}
```

Figure 8.4 – An API for `CPSListF`

Figure 8.4 gives an implementation of `CPSListF`. The notation `F[_]: Functor` specifies that the higher-kinded type `F` requires evidence of a `Functor` typeclass. Note that `CPSListF` is not parameterised by `R` anymore. This type is instead delayed to the `fold` function. As a result we only need to specify the eventual result type when the `fold` function is applied.

We create a `CPSList`, or a `CPSListParser`, from a list, or a parser, respectively (Figure 8.5). The implementation of the generic `fold` function simply forwards to the specific, and concrete, foldLeft, or repFold, respectively. The API of CPSListF closely mirrors the API for CPSList we have previously seen.

```
type CPSList[T] = CPSListF[T, Id]
type CPSListParser[T] = CPSListF[T, Parser]

def fromList[T](ls: List[T]): CPSList[T] = new CPSListF[T, Id] {
  def fold[R](z: R, combine: Combine[T, R]) = foldLeft(ls)(z, combine)
}

def fromParser[T](p: Parser[T]): CPSListParser[T] = new CPSListF[T, Parser] {
  def fold[R](z: R, combine: Combine[T, R]) = repFold(p)(z, combine)
}
```

Figure 8.5 – Creating variants of CPSListF, for classic lists or parsers

We have already previously used partial evaluation over CPSList to achieve fusion. The more generic CPSListF tells us that a pipeline of operations on a CPSList has an equivalent pipeline over CPSListParser. Fusion properties carry over as well. This allows us to rewrite above examples as below, and benefit from deforestation.

```
def numAdults2: Parser[Int] = fromParser(personParser).length
def namesOfAdults2: Parser[String] =
  fromParser(personParser).filter(_.age >= 18).map(_.first).toList
```

Naturally, one would not expect to manually convert parsers written in the namesOfAdults style to the namesOfAdults2 style. Indeed, the original parser seems more natural to write, since it separates the composing of parsers from semantic actions (queries) performed on the result. This is all the more visible if we sequence personParser with some other parser later on. In the next section, we describe a transform that mechanically rewrites the more declarative version of a parser to its interleaved, but more efficient, counterpart.

Interleaving parsers and queries

Treating repetition parsers as lists gives us more expressivity from a combinator library point of view. By using the fromParser combinator and operations over CPSListParser we push operations close to the creation of a single parse result.

This is however not enough: a user is still required to *manually* place a query close to the parser it operates on. Consider that in addition to people parsers by personParser, the raw data file contains a list of countries. A parser for consuming both lists would sequence personParser with countriesParser. If we wish to filter out the adults and all countries in Asia, the natural way to write the parser is given in Figure 8.6.

Using CPSListParser we would instead write a filtered2 function (Figure 8.7). Performing

```
case class Person(first: String, last: String, age: Int)
case class Country(name: String, continent: String)
def personParser: Parser[Person] = ...
def countryParser: Parser[Country] = ...

def peopleAndCountries = rep(personParser) ~ rep(countryParser)
def filtered = peopleAndCountries map { t =>
  (t._1.filter(_.age >= 18), t._2.filter(_.continent == "Asia"))
}
```

Figure 8.6 – Idiomatic filtering of a sequence of list parsers

the conversion by hand breaks the declarative nature of the query. If we wanted to slightly change it, we would only ever have to rewrite the anonymous function in `filtered`. Changing `filtered2` is much more involved, and requires us to think about the flow of a parser rather than just about the query.

```
def filtered2 = (
  fromParser(personParser).filter(_.age >= 18).toList ~
  fromParser(countryParser).filter(_.continent == "Asia").toList
)
```

Figure 8.7 – Filtering a sequence of list parsers, close to the list creation

Fortunately is it possible to automatically convert `filtered` like queries into their `filtered2` like counterparts. In this section we describe a set of rules which perform this transformation. We then argue that the transformation is correct, i.e. that it preserves semantics (Section 8.3.1), and finally discuss optimisation properties (Section 8.3.2). For the rest of this section we change the type of the `rep` combinator to return a `Parser[CPSList[T]]`:

```
def rep[T](p: Parser[T]): Parser[CPSList[T]]
```

This modification ensures that the transformation of repetition parsers is restricted to operations over `CPSList`. Such operations `CPSList` *must* be concluded with a final call to the `fold` function, to create an actual data structure.

The transformation operates in a purely functional subset of Scala with products and sums, with strict evaluation semantics. The core rule T transforms as AST representing a `Parser [T]` to another AST of `Parser[T]`. We restrict these ASTs to be non-recursive; this keeps the transformation rules simple and clear. We discuss recursion in more detail in Section 8.6.

We subdivide the transformation into three parts. In these, metavariables a, b, p designate parsers and f_i designate ASTs corresponding to pure functions on simple types, with appropriate arity.

MAIN TRANSFORMATION $T : AST[\texttt{Parser[T]}] \rightarrow AST[\texttt{Parser[T]}]$

Sequencing

$$\begin{aligned}
T[\![a \sim b]\!] &= T[\![a]\!] \sim T[\![b]\!] && \text{(S-0)} \\
T\left[\!\!\left[\begin{array}{l}(a \sim b)\ \texttt{map}\ \{ \\ \quad \texttt{t => } f_3(f_1(\texttt{t.\_1}), f2(\texttt{t.\_2})) \\ \}\end{array}\right]\!\!\right] &= \begin{array}{l}(T[\![a\ \texttt{map}\ f_1]\!] \sim T[\![b\ \texttt{map}\ f_2]\!])\ \texttt{map}\ \{ \\ \quad \texttt{t => } f_3(\texttt{t.\_1}, \texttt{t.\_2}) \\ \}\end{array} && \text{(S-1)} \\
T[\![(a \sim b)\ \texttt{map \{ t => } f(\texttt{t.\_1})\ \texttt{\}}]\!] &= T[\![a\ \texttt{map}\ f]\!]\ \texttt{<\~{}}\ T_{\text{RECO}}[\![b]\!] && \text{(S-2A)} \\
T[\![(a \sim b)\ \texttt{map \{ t => } f(\texttt{t.\_2})\ \texttt{\}}]\!] &= T_{\text{RECO}}[\![a]\!]\ \texttt{<\~{}}\ T[\![b\ \texttt{map}\ f]\!] && \text{(S-2B)}
\end{aligned}$$

Repetition

$$\begin{aligned}
T\left[\!\!\left[\begin{array}{l}\texttt{rep}(p)\ \texttt{map}\ \{ \\ \quad \texttt{ls => } g(\texttt{ls}).\texttt{fold}(z, f_c) \\ \}\end{array}\right]\!\!\right] &= T\left[\!\!\left[\begin{array}{l}T_{\text{FUNC}}[\![g]\!](\texttt{fromParser}(p)). \\ \quad \texttt{fold}(z, f_c)\end{array}\right]\!\!\right] && \text{(R-1)} \\
T\left[\!\!\left[\begin{array}{l}g(\texttt{fromParser}(p)\ \texttt{map}\ f). \\ \quad \texttt{fold}(z, f_c)\end{array}\right]\!\!\right] &= T[\![g(\texttt{fromParser}(p\ \texttt{map}\ f)).\texttt{fold}(z, f_c)]\!] && \text{(R-2)} \\
T[\![g(\texttt{fromParser}(p)).\texttt{fold}(z, f_c)]\!] &= g(\texttt{fromParser}(T[\![p]\!])).\texttt{fold}(z, f_c) && \text{(R-3)}
\end{aligned}$$

Alternation

$$\begin{aligned}
T[\![a \mid b]\!] &= T[\![a]\!] \mid T[\![b]\!] && \text{(A-0)} \\
T[\![(a \mid b)\ \texttt{map}\ f]\!] &= T[\![a\ \texttt{map}\ f]\!] \mid T[\![b\ \texttt{map}\ f]\!] && \text{(A-1)}
\end{aligned}$$

Other

$$\begin{aligned}
T[\![p\ \texttt{map}\ f\ \texttt{map}\ g]\!] &= T[\![p\ \texttt{map}\ (g \circ f)]\!] && \text{(MAP-MAP)} \\
T[\![p]\!] &= p && \text{(DEFAULT)}
\end{aligned}$$

Figure 8.8 – The main transformation

Main transformation T**.** This transformation mainly handles rewriting parsers that perform semantic actions, i.e. of the form `p map f` (Figure 8.8). For parsers that are not of this form, we either propagate the transformation to the underlying parsers (S-0, A-0) or perform an identity rewrite (DEFAULT). The latter applies to base parsers, as described in Section 6.2.

The S-1 rule pushes semantic functions as close as possible to the creation of a parse result. By doing so we create possibilities for other rules to be applicable, and optimise parsing. Two such rules are S-2A and S-2B: if we identify that we only use one of the halves of a sequence parser, we can convert the other half into a recogniser, and ultimately discard the result, using the variants ~> and <~, respectively.

For repetition parsers, the R-1 rule performs the functor change for `CPSListF` (see Section 8.2), from a `CPSList[T, Id]` to a `CPSList[T, Parser]`. This is done by invoking the T_{FUNC} tranformation. We invoke T on the result of T_{FUNC} to trigger further rewrite possibilities. The R-2 rule is one of them. It converts a `map` on a `CPSListParser` to a `map` on the underlying parser. Finally R-3 pushes T to the underlying parser.

The transform rule for alternation is straightforward, it simply pushes T to both alternatives. Finally, the MAP-MAP rule pushes function composition under the parser functor. From a partial evaluation point of view this rules is redundant, since inlining will automatically produce code that composes functions correctly. From a practical point of view it is nonetheless useful, in particular when extending the transformation to recursive data types (Section 8.6).

CPSLIST TO CPSLISTPARSER

$$T_{\text{FUNC}} : AST[\texttt{CPSList[T] => CPSList[U]}] \\ \rightarrow AST[\texttt{CPSListParser[T] => CPSListParser[U]}]$$

$$\begin{array}{rcll}
T_{\text{FUNC}} \left[\!\!\left[\begin{array}{l} \texttt{(ls: CPSList[T]) =>} \\ \quad g\texttt{(ls) map } f \end{array} \right]\!\!\right] & = & \begin{array}{l} \texttt{(ls: CPSListParser[T]) =>} \\ \quad T_{\text{FUNC}}[\![g]\!]\texttt{(ls) map } f \end{array} & (\text{FUNC-MAP}) \\
T_{\text{FUNC}} \left[\!\!\left[\begin{array}{l} \texttt{(ls: CPSList[T]) =>} \\ \quad g\texttt{(ls) filter } f \end{array} \right]\!\!\right] & = & \begin{array}{l} \texttt{(ls: CPSListParser[T]) =>} \\ \quad T_{\text{FUNC}}[\![g]\!]\texttt{(ls) filter } f \end{array} & (\text{FUNC-FILTER}) \\
T_{\text{FUNC}}[\![\texttt{(ls: CPSList[T]) => ls}]\!] & = & \texttt{(ls: CPSListParser[T]) => ls} & (\text{FUNC-ID})
\end{array}$$

Figure 8.9 – Converting `CPSLists` to `CPSListParsers`

T_{FUNC}**: from `CPSList` to `CPSListParser`.** This set of rules recursively transfers a pipeline of operations on a `CPSList` to a pipeline of calls on `CPSListParser`.The metavariable g designates functions from `CPSListF` to `CPSListF`, as per the API in Figure 8.4.

PARSER TO RECOGNISER

$$T_{\text{RECO}} : AST[\texttt{Parser[T]}] \rightarrow AST[\texttt{Parser[Unit]}]$$

$$\begin{array}{rcll}
T_{\text{RECO}}[\![a \texttt{ map } f]\!] & = & T_{\text{RECO}}[\![a]\!] & (\text{RC-MAP}) \\
T_{\text{RECO}}[\![g(\texttt{fromParser}(p))\texttt{.fold}(z, f_c)]\!] & = & \begin{array}{l} \texttt{fromParser(}T_{\text{RECO}}[\![p]\!]\texttt{).} \\ \quad \texttt{fold((), (acc, \_) => acc)} \end{array} & (\text{RC-REP}) \\
T_{\text{RECO}}[\![a \sim b]\!] & = & T_{\text{RECO}}[\![a]\!] \;\texttt{\textasciitilde>}\; T_{\text{RECO}}[\![b]\!] & (\text{RC-SEQ}) \\
T_{\text{RECO}}[\![a \mid b]\!] & = & T_{\text{RECO}}[\![a]\!] \mid T_{\text{RECO}}[\![b]\!] & (\text{RC-ALT}) \\
T_{\text{RECO}}[\![a : \texttt{Parser[Unit]}]\!] & = & a & (\text{RC-BU}) \\
T_{\text{RECO}}[\![a]\!] & = & \mathit{recognise}(a) & (\text{RC-BASE})
\end{array}$$

Figure 8.10 – Converting parsers to recognisers

T_{RECO}**: from `Parser` to `Recogniser`.** This rule is key in the optimisation process. Once we have identified that we do not require the results of a certain parser, we can convert it into a recogniser.

The RC-MAP rule discards the application of a `map`, and converts the underlying parser into a recogniser. RC-ALT pushes the transformation through, while RC-SEQ also replaces the sequence combinator `~` with its discarding variant `~>`. The rule for repetition is more interesting. Indeed, it not only transforms the underlying parser into a recogniser, but it also folds the results into `Unit`. The consequence of this rewrite is that a potentially large accumulation into a list is now but a recognition of the underlying parser followed by an immediate discard.

All the above rules depend on RC-BASE, which operates on base parsers. This rule depends on a recogniser available in scope (`recognise(p)`). Unfortunately, we must require such implementations to be available. Indeed, it is not easy to "guess" what a recogniser for a base parser can be. However, it is not only possible to provide base implementations for recognisers, it is also possible to enforce, from a library point of view, that any base parser be implemented with a recognising counterpart. In our implementation we introduce an additional abstract method `def recognise: Parser[Unit]` in the `Parser` class. The compiler will therefore complain if any parser does not provide an implementation for this method. An alternative implementation is to require an evidence for a `Recogniser` typeclass for every parser. The base case is naturally the id transformation for a parser that is already a recogniser, i.e. a `Parser[Unit]`. This is reflected in the RC-BU rule. As a result, T_{RECO} does not require a meta-language for its implementation.

Our intuitive transformation of `filtered` into `filtered2` at the beginning of the section can now be derived using the rules (Figure 8.11).

```
  peopleAndCountries map { t =>
    (t._1.getAdults.toList, t._2.getAsian.toList)
  }
```

= `(rep(personParser) map getAdults).toList ~ (rep(countryParser) map getAsian).toList` (S-1)

= $T_{\text{FUNC}}[\![$`getAdults`$]\!]$ `(fromParser(personPaser)).toList ~` $T_{\text{FUNC}}[\![$`getAsian`$]\!]$`(fromParser(countryParser)).toList` (R-1)

```
=  (fromParser(personPaser).
     filter(_.age > 18)).toList
     ~ (fromParser(countryParser).
         filter(_.continent == "Asia")).toList
```
(FUNC-FILTER, FUNC-ID)

Figure 8.11 – From `filtered` to `filtered2`, mechanically

Correctness

The transformation T does not change the semantics of the parser it transforms. By semantics, we mean the following:

- If a parser p succeeds on an input `in`, then $T\llbracket p \rrbracket$ succeeds on `in` with the same result.
- If p fails on a given input, so will $T\llbracket p \rrbracket$.

The justification hinges on the correct implementation of base recognisers (The RC-BASE rule). We can show that our recognisers, such as `acceptIfReco`, do indeed preserve semantics, but we cannot guarantee that base parsers added externally do the same. Therefore, we argue preservation of semantics modulo correct implementation of base recognisers.

The rest of the argument is a reasoning by structural induction on the shape of `p`. The base cases involve base parsers, which are handled by the DEFAULT and RC-BASE rules. We discussed the latter just above. The former is the identity transform, and therefore trivially preserves semantics.

Many of the other rules are straightforward applications of induction. The less obvious cases are S-1, S-2A, S-2B, and R-2. The first group uses functorial properties of products along with those of parsers. Indeed, in S-1 the expression $f_3(f_1(\texttt{t.\_1}), f2(\texttt{t.\_2}))$ corresponds, in a pure functional setting to applying a `map` on the pair `t`, followed by a `fold` on the pair. For S-2 we have an application of `fold` on pairs that discards one of the elements.

We satisfy `R-2` by expanding both the left and right hand sides as per the definitions of `fromParser`, `map` on `CPSList`. The derivations arrive at matching expressions in the success pattern match of the `loop` function of `repFold`.

The other key rules involve R-1 and the FUNC rules. These preserve semantics due to the functorial properties of `Parser` over `CPSListF`, as seen in Section 8.2.

Optimisation

By performing the transformation, we hope to ultimately achieve better performance. From a formal point of view, the notion of optimisation is captured more accurately in terms of allocation of fewer objects.

For most of the rules presented above, this optimisation property does hold. As with the correctness argument, this depends on the correct implementation of base recognisers. For `acceptIfReco`, on successfully passing the predicate we assign the unique instance of `Unit` instead of an `Elem`; we do not allocate anything as a result.

The rules that do make a difference (in creating fewer objects) are:

- R-1: By moving from `CPSList` to `CPSListParser`, we avoid creating the list `ls` on the left hand side of the rule.
- RC-REP: We are converting a fold operation into a data structure into a fold operation into `Unit`. The amount of allocations we save depends on the structure we previously

fold into. Typically we fold into a list, or another type of collection. The construction of that collection is fully avoided here.

- RC-MAP: We avoid allocating a value of the resulting type of the `map` function.
- RC-SEQ: We avoid creating a pair value to contain results of `a` and scodeb.

Unfortunately this property breaks for S-1. If the sequencing parser `a ~ b` fails globally after succeeding for `a`, bringing an expensive function `f1` will create more allocations than in the non-transformed version.

Even if we relax optimisation properties to apply only for successful parses, we cannot guarantee it due to the alternative combinator and the A-1 rule. Consider for instance the following parser:

```
def aParser = ((a ~ b) | (c ~ d)) map { t => (f1(t._1), f2(t._2)) }
```

According to the transformation rules, A-1 pushes the `map` function to both alternatives, and in turn S-1 gets triggered, pushing `f1, f2` close to their creation site. An input that succeeds on `a` but fails the first alternative `a ~ b` will compute `f1`. If `f1` is an expensive function, we will end up creating many more objects than required.

The issues shown here apply more generally to inlining and conditional expressions. In Section 3.2.2 we mitigated the problem by introducing explicit join points. The rules presented here operate on parser ASTs, a meta-level higher than the code itself. The options we have are therefore either to perform the transformation or not to. In the negative, we might be missing optimisations opportunities further down the line.

At present, in practice, we systematically perform the transformation whenever it applies. A possibly better solution is to base the transformation on heuristics, which we leave for future work.

Implementation

In the previous section we described transformation rules that convert a parser to a specialised version. Unutilised parsers are converted into recognisers. We saw that one of the rules, T_{RECO} (see Figure 8.8), could be implemented directly in the host system as a method on the `Parser` class. For the other two rules, we choose to use Scala macros to implement them [8]. The transformation T is transforms trees of parsers to trees of parsers. For true performance gains, it is therefore not only important that the transformation creates fewer objects (Section 8.3.2), it is also crucial that the underlying parser combinator implementation is efficient itself.

We have seen how to eliminate overhead abstraction by partially evaluating parser combinators in Chapter 6. The implementation was in LMS. In this section we present a macro-based implementation of the same. By doing so we illustrate that the optimisation as a library approach can be easily ported beyond a specific staging framework.

The library we describe here is the successor of previously published work [7]. The main addition is the transformation phase described earlier. While implementing this phase we also redesigned large parts of the library in order to be closer to the design principles described in this dissertation. At the time of writing not all features present previously have been fully ported. We hope to do so in the near future.

```
val aParser = optimise {
  def a: Parser[A] = ...
  def b: Parser[B] = ...

  def finalParser = rep(a ~ b) map { ls => /* query on ls */ }
  finalParser
}
```

Figure 8.12 – A typical usage of the `optimise` macro

This library, named `parsequery`, is used by scoping parsers inside an `optimise` macro, as shown in Figure 8.12. The macro accepts a sequence of parser declarations followed by a final parser. The latter is typically a call to the `map` combinator where the `f` function is a query on the parsed data structure.

The parser declarations are implemented as a basic, unstaged parser combinator library, as presented in Section 6.2. This means that they can be used even without being wrapped in the `optimise` macro. They will yield the same result given the same input, but since they are not deforested, will run slower. The implementation of the `optimise` macro consists of various phases:

- **Step 1:** construct a compile-time AST for each parser declared in scope.
- **Step 2:** transform each parser in scope according to the rules.
- **Step 3:** stage each parser in order to optimise it.
- **Step 4:** create an optimised version for each parser by partially evaluating it.

In the background section we mentioned that macros operate on typed trees (Section 2.3.2). In other words, any program wrapped inside the macro is well typed. Trees representing the program can be accessed in their typed form, as expressions of type `T` (`Expr[T]`). They are also accessible in their untyped form `Tree`. In the latter case the reflection API can be used to recover types of original expressions, since we are guaranteed of receiving well-typed programs as input to the macro: every tree has a member `tpe` of type `Type`. For the `parsequery` library we have chosen the latter representation: the implementation in practice was considerably simplified thanks to this choice.

Step 1: Lifting parsers to an AST.

```
abstract class Grammar(val tpe: Type)

/** base parsers */
case class AcceptIf(path: List[Tree], p: Tree => Tree)
  extends Grammar(realElemType)
case class AcceptStr(path: List[Tree], s: String)
  extends Grammar(typeOf[String])
case class PIdent(name: Ident) extends Grammar(name.tpe)

/** combinators */
case class Mapped(g: Grammar, f: Tree => Tree, tpe2: Type) extends Grammar(tpe2)
case class Concat(l: Grammar, r: Grammar, tpe2: Type)
  extends Grammar(appliedType(typeOf[Tuple2[_, _]], List(l.tpe, tpe2)))
case class ConcatLeft(l: Grammar, r: Grammar, tpe2: Type) extends Grammar(l.tpe)
case class ConcatRight(l: Grammar, r: Grammar, tpe2: Type) extends Grammar(tpe2)
case class Or(l: Grammar, r: Grammar, tpe2: Type) extends Grammar(tpe2)

case class Rep(g: Grammar, tpe2: Type)
  extends Grammar(appliedType(typeOf[List[_]], tpe2))
case class Repsep(g: Grammar, g2: Grammar, tpe2: Type, u: Type)
  extends Grammar(appliedType(typeOf[List[_]], tpe2))
```

Figure 8.13 – A compile-time AST for parser combinators

A def macro accepts any arbitrary Scala program. Since we operate in the domain of parsers and queries we restrict our language further. An `optimise` scope should contain a sequence of parser declarations, followed by a final expression of higher-kinded type `Parser`. The parser declarations themselves should be declared as method definitions (**`def`**, not **`val`**) that take no parameters, have a return type `Parser`, and have a right-hand side expression that can be lifted to a compile-time, domain-specific AST, `Grammar`. For any scope not matching this specification we return a domain-specific compile-time error.

To check these properties we use quasiquotes. A parser declaration is matched as follows:

```
val parserType = typeOf[Parser[_]]
stmt match {
  case q"def ${name: TermName}[..$tparams]: ${retType: Type} = ${g: Grammar}"
    if retType <:< parserType => ...
}
```

A value is captured by the unquote operator `$`. The type ascription syntax allows us to bind values to more specific data types than the generic `Tree`. For example, we require `retType` to be a `Type` and `name` to a `TermName` in the above. This operation of binding to more specific types is known as *unlifting* (the dual operation lifting enables unquoting of specific data types). By default quasiquotes support unlifting for primitive types, constants, and some special tree data types from the scala reflection API.

Note that the `Grammar` data type is however not supported by default. Indeed, it is specific to our domain. It is possible to extend quasiquote support for lifting and unlifting of user-specified data types as well. As a result we effectively convert trees representing arbitrary Scala code into trees representing a more specific domain (or fail with a compile-error in the process). Figure 8.13 gives the main trees we unlift to. We have trees representing the principal combinators for sequencing, alternation, repetition and mapping, as well as trees for base parsers. We have a specific grammar tree for named parsers (`PIdent`), which occur if a parser calls another one declared in scope (possibly recursively).

One may wonder whether we need to create a domain-specific tree for every combinator we encounter. This can be tedious since there are many combinators implemented in terms of others, such as for example `accept` (which uses `acceptIf`). Unlifting must be done for every combinator, but it is not necessary to map it to a dedicated tree. We can use the knowledge of implementation of a combinator to *desugar* it into a less specific tree. Therefore unlifting an instance of `accept` yields an `AcceptIf` tree with the appropriate function as its parameter. In essence, unless a combinator is essential, or can be optimised in a specific manner, we desugar it to an existing tree during the unlifting phase.

Function values passed to combinators (`acceptIf, map`) are also handled specially: for `map`, the function literal is of type `Tree` as seen from the macro scope:

```
q"${subg: Grammar}.map[${t: Type}]($f)"
```

Yet the AST counterpart `Mapped` has a member of type `Tree => Tree`: this is equivalent to a static function type on dynamic input and output values (`Rep[T] => Rep[U]`). By applying this function to a dynamic input (of type `Tree`) we effectively inline the body of the function. Indeed, during the lifting phase we already stage functions. If `f` is a literal (i.e. an anonymous function), we create an unstaged function whose body is `f`'s body, with the arguments substituted appropriately. Otherwise we simply eta-expand it. We could do better if `f` is declared in the `optimise` scope, since we would have access to its body as well. At present however, this is not possible, since we restrict the block to contain parser declarations only. Growing the variety of expressions supported by the macro is part of future work.

Note that `Grammar` is not polymorphic in the traditional sense. We “cheat” by providing a field representing the type of the grammar instead. For all intents and purposes these are equivalent.

At the end of the lifting phase, we have a set of names associated to domain-specific `Grammar` trees. The rest of the optimisations are based on these trees.

Steps 3-4: Staging parsers and partially evaluating them

```
abstract class StagedParser(val elemType: Type) extends (CharReader => ParseResult) {
  def map(t: Type, f: Tree => Tree): StagedParser
  def flatMap(t: Type, f: Tree => StagedParser): StagedParser
  def ~(that: StagedParser): StagedParser
  def or(t: Type, that: StagedParser): StagedParser
}
def rep(elemType: Type, p: StagedParser): StagedParser
def repsep(p: StagedParser, sep: StagedParser): StagedParser
def acceptIf(elemType: Type, p: Tree => Tree): StagedParser

abstract class Reader(elemType: Type) {
  def first: Tree
  def atEnd: Tree
  def rest: Reader
}
abstract class CharReader extends Reader(typeOf[Char]) { ... }

abstract class ParseResult(val elemType: Type) { self =>
  def apply(success: (Tree, CharReader) => Tree, failure: CharReader => Tree): Tree
  def map(t: Type, f: Tree => Tree): ParseResult
  def flatMapWithNext(t: Type, f: Tree => CharReader => ParseResult): Tree
  def orElse(t: Type, that: ParseResult): Tree
}
def mkSuccess(elemType: Type, elem: Tree, rest: CharReader): ParseResult
def mkFailure(rest: CharReader): ParseResult
def cond(elemType: Type)(test: Tree, thenp: ParseResult, elsep: ParseResult): ParseResult
```

Figure 8.14 – An interface for staged parser combinators, with Scala macros

Right after the lifting phase (and after the core transformation phase) we are in position of a set of `Grammar` trees, representing parsers declared in scope. We could simply convert them back into user-level parsers: this would either correspond to the identity transformation (if done before the transformation phase) or a library-level implementation of the interleaved parser. In both cases, execution still suffers from poor performance as we have not partially evaluated function composition yet.

To do so we first stage every `Grammar`:

```
def stage(g: Grammar): StagedParser
```

A `StagedParser` is a data type representing a staged parser. Its interface is given in Figure 8.14. We recognise an almost direct port of the LMS implementation in Figures 6.11 and 6.12:

- A parser is a static function from an input to a parse result. In this case we have already specialised the input to be a character reader.
- A parse result is a Church-encoded data type, taking continuations for success and

failure

- A `CharReader` is a CPS-encoded reader whose methods `first` and `atEnd` return dynamic values.

The main difference is the usage of unstaged functions on generic `Tree` types instead of unstaged functions on `Rep` types.

The implementation of the `stage` function is fairly straightforward: to every `Grammar` variant corresponds a combinator in the `StagedParser` world. So it is sufficient to pattern match on the variants, and stage grammars recursively. The main difference being with recursive parsers. Rather than using a dedicated `rec` combinator as with LMS, we rely on parser names recovered from the lifting step: whenever we pattern match a `PIdent` we create a `StagedParser` whose body is a call to the named parser in question.

Once each `Grammar` has been staged, the final step in the macro is to recreate a user-level parser definition, but based on the optimised, staged equivalents. In step 3 we created a `StagedParser` for each user-level parser declared in the macro step. We now perform the reverse transformation, by creating a user-level parser definition. The difference lies in the body. A `StagedParser` is an unstaged function over staged types: the right-hand side of the user-level parser is the result of (statically) applying this function to a (freshly generated) symbolic argument. In other words, the new body of the user-level parser is the deforested equivalent of the combinator-based counterpart that the user initially wrote.

Step 2: Transforming parsers

The core transformation takes a `Grammar` as input, and returns a `Grammar`. As mentioned before, we follow the rules provided in Figures 8.8, 8.9 and 8.10. Many of these rules are fairly straightforward to implement. As suggested by the rules, we match the relevant left-hand side, and turn it into the corresponding right-hand side. The tricky parts lie in the variety of patterns we recognise and transform. With the $T_{\text{FUNC}}[\![.]\!]$ rules for instance, we perform the transformation only if the expression matches syntactically, i.e. the body is a single expression containing chains of calls to `map` and `filter`. We plan to tolerate a wider variety of expressions in the future (see Section 8.6 for more details).

The trickiest rules to implement are S-1, S-2A, S-2B, which take as input expressions of the form `(a ~ b) map f`. Here we analyse the body of `f` in order to identify largest independent application subtrees for each argument of the function: subtrees where only one of the results of `a` or `b` are manipulated. If we map each result to exactly one such subtree, we extract these into their own function bodies before propagating them to their corresponding individual parsers.

In case we do not recognise specific forms, we default to the identity transformation (the DEFAULT rule). To help the user of the library we output compiler warnings where we suggest changing the syntactic form of the function in order to enable more optimisation opportunities.

Efficient base parsers

For good performance we need efficient implementations of base parsers and recognisers other than `acceptIf` and `acceptIfReco`. Our library provides these for identifiers, string matchers, string literals, digit and number parsers by default. In particular, identifiers and string matchers seek to parse a string passed as input. We hoist this string out into a static value in scope in order to not recreate it every time the parser runs.

Evaluation

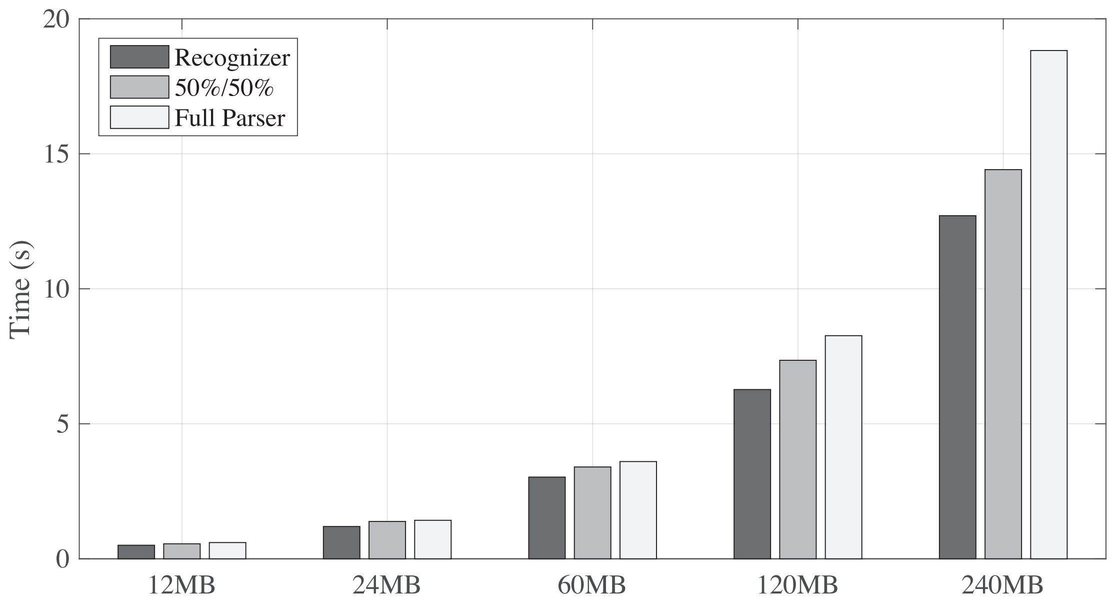


Figure 8.15 – Running time as we vary the file size from 12 to 240 MB.

We evaluate our transformation as follows: we measure the running time, for a sequence of tuples, each one a key-value pair of string literals, of recognising the tuples versus. parsing increasing percentages of these tuples. Each element contains 17 key-value pairs.

We use the Scalameter [66] benchmarking suite for our evaluation. This framework handles JIT warm-ups as well as running a benchmark until performance stabilizes. The benchmarks are executed on a computer running Ubuntu 12.04, with 32GB DDR3 of RAM memory and an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz.

In Figures 8.15, 8.16, and 8.17, we benchmark the running time over the file size for three cases: full parsing, full recognition and a mixed approach in which we parse half of the tuples

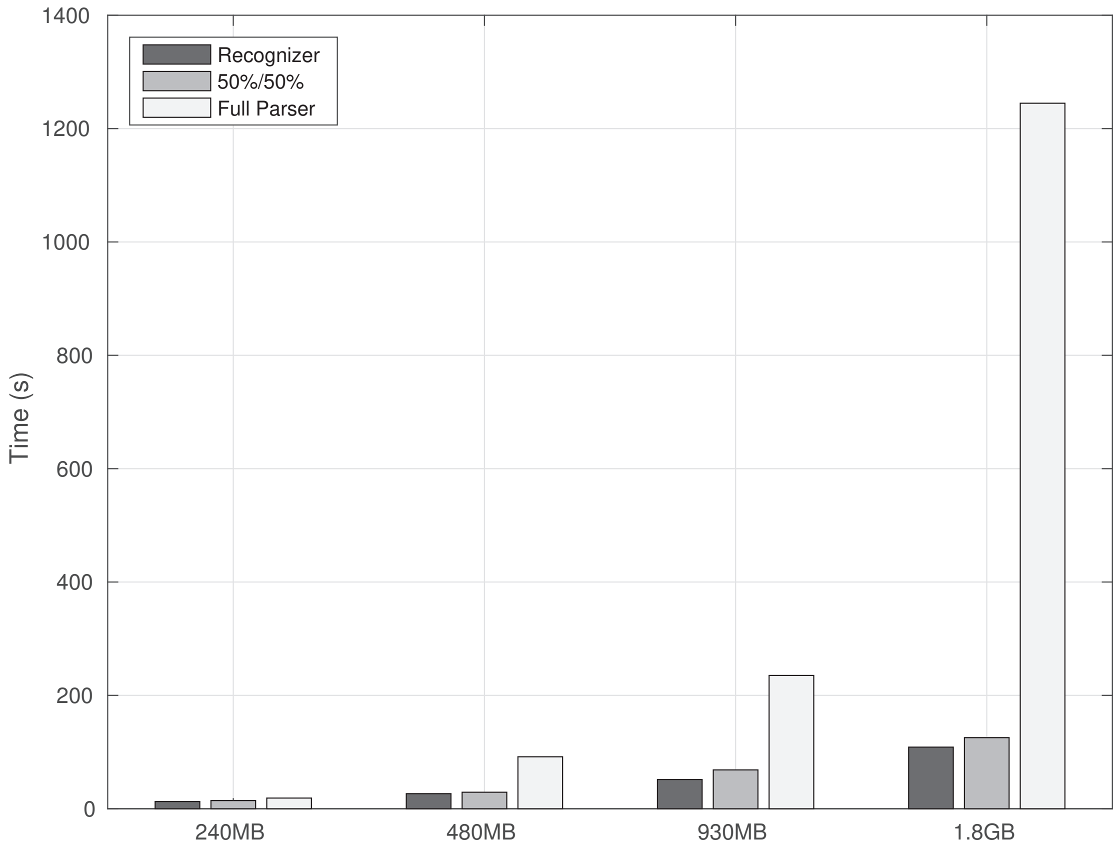


Figure 8.16 – Running time as we vary the file size from 240 to 1800 MB.

dataset described above and recognise the other half. We vary the size by doubling the number of tuples each iteration from 12MB to 1800MB. This is separated into three figures for clarity: Figure 8.15 varies the file size from 12 to 240MB, Figure 8.16 varies the file size from 240 to 1800MB. Figure 8.17 shows all these in log scale.

Figure 8.18 varies the percentage of tuples that are parsed. It shows 5 data points computed on the largest file size, 1800MB. These data points represent parsing 0, 25, 50, 75, 100%. That is, the “25%75%” data point in the figure represents the case where we recognize 25% of the dataset and parse 75%.

The figures show that recognising really pays off. For a dataset of only 1.8 GB we get speedups of up to 11x. Even if we parse 50% of the original data we are not significantly worse off than recognising alone (10x faster than full parsing). This suggests that there is a threshold after which the running time worsens significantly on large datasets. We believe this is due to effects of the garbage collector, which kicks in an excessive number of times since the object creation rate is very high and surpasses the expected rate of the JVM. Therefore, the JVM heuristics make the garbage collector fully clean the memory before an `OutOfMemoryError` is thrown.

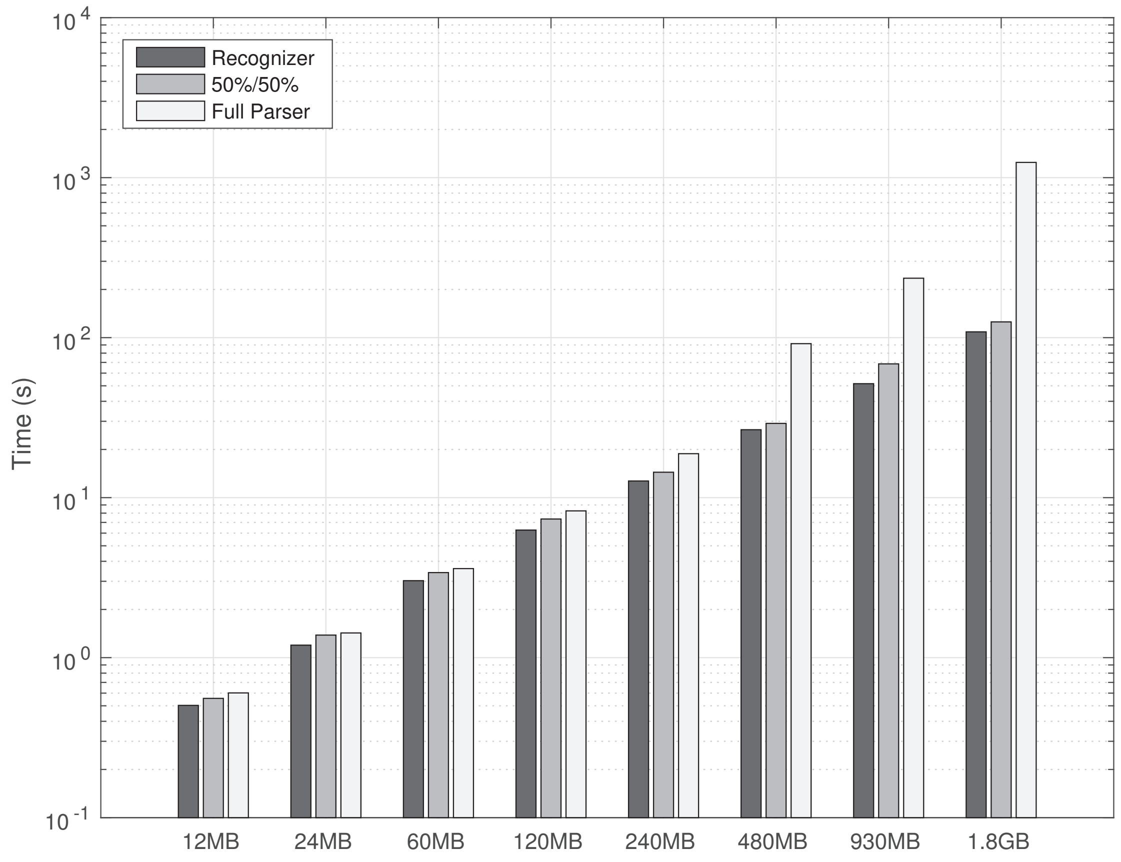


Figure 8.17 – Running time as we vary the file size from 12 to 1800 MB in log scale.

Discussion

Evaluation of our transformation shows that it really pays off to skip as early, and as often as possible. In this section we discuss potential additions to the transformation rules that can potentially trigger further optimisation possibilities. We also discuss other implementation alternatives and additions for the staging and transformation steps.

Recursive parsers and queries

The transformation rules presented previously apply to queries on nested, non-recursive data structures. Many prevalent and popular data formats satisfy these properties.

Yet this is a restriction in theory and in practice. For instance, queries such as "count all variable declarations in this program snippet" are not covered. Also, in practice, it is common to use generic recursive parsers, such as JSON and XML, to parse a data structure before either a) decoding it into a precise schema or b) writing a dynamic query. We sketch a technique to extend our transformation rules for such queries and parsers.

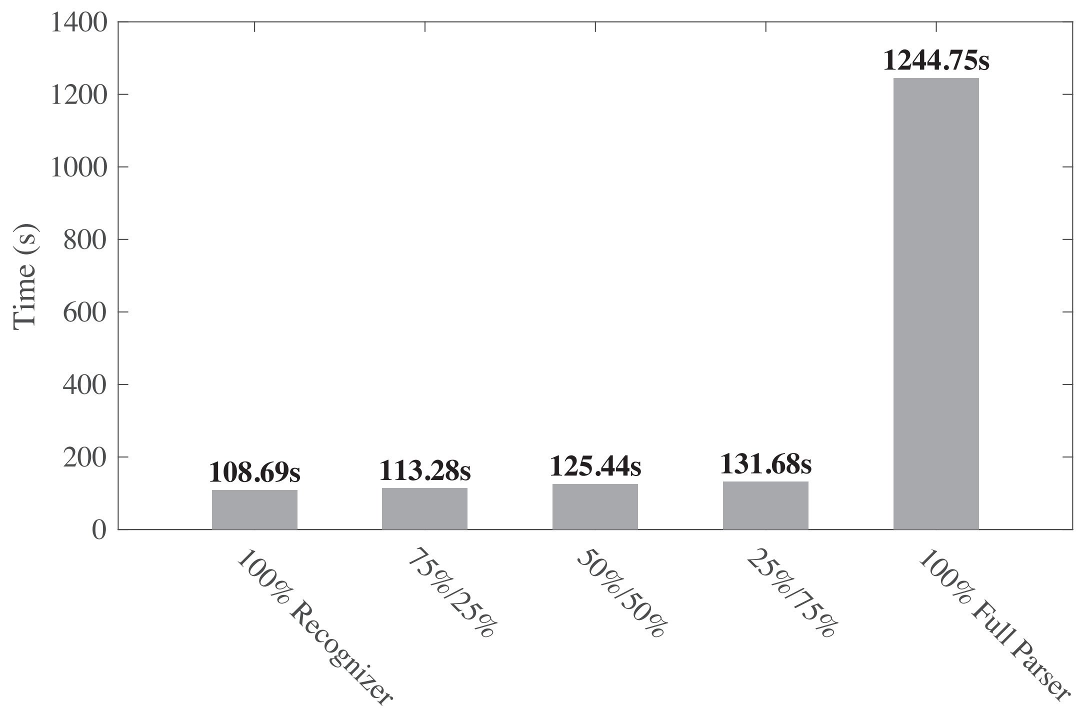


Figure 8.18 – Varying the percentage of recognising vs. parsing on the largest file size, 1800MB

We first restrict ourselves to non-recursive queries but recursive parsers where the first operation on the parsed data structure is to decode it into a more precise non-recursive data structures. The structure of this parser matches the MAP-MAP rule, i.e. has the form `p map decode map query`, where `decode` and `query` are the decoding and querying functions, respectively. The parser `p` itself constructs an arbitrary ADT. Therefore `decode` contains all information required to specialise the recursive parser `p`. Essentially, the decoding function converts variants of the recursive data type from `p` by matching on their structure. These structures are themselves constructed by alternating parsers that combine to yield `p`. That is, there is mapping between data type constructions and their parsers. Therefore we only need to unfold the `decode` function in a top-down manner. Whenever a data type constructor is matched, we create recursively create specialisations for the parsers the map to. This process will terminate since we know that the datatype we decode to is finitely nested.

Converting a recursive parser into a recognizer becomes easy. We follow the RECO rules until we encounter a parser that we are already transforming, and exchange the call to the untransformed parser by a recursive call to its corresponding recogniser.

For recursive queries the transformation is similar to that for recognisers. Since a matching field can be nested at any level the idea it to specialise parsers that yield particular variants of the data type, and substitute their original versions for these these in the global parser `p`.

Implementation-wise this transformation is a special case of the call-pattern specialisation optimisation in Haskell [62]. The heuristics for deciding when to specialise or not would differ from those for general recursive programs, and be more domain-specific.

Specialisation of filters

A common use case is, while parsing a sequence, to conditionally keep the result based on a predicate on one of the underlying parsers. For instance, in the `personParser` example from before, we could first recognise the triple, check that the age is above 18 and only *then* parse the input. Such a rule can be easily added to the set of rules from Figure 8.8. But the efficacy of this rule is dependent on where the predicate sits, and how many elements it is applied on.

Sharing parsers between multiple scopes.

The `optimise` macro operates on parsers declared in its scope, but does not work beyond the boundary. This situation is not ideal because it reduces modularity and violates the don't repeat yourself (DRY) principle. We can re-use a parser declared in a different scope, but transformation and staging will not interleave this parser with others. One would have to implement the whole parser in scope.

There are two possible solutions to this. In the previous version of the library we used the `@saveAST` annotation to have access to the trees in the macros itself, so that they could be used and expanded as and when required [7]. A more principled solution would be to implement reusable parsers directly in the staged world, in separate components that can be mixed in with general staged parsers. We would also need to extend lifting and unlifting, as well as the `Grammar` data type. This follows the modular approach advocated by LMS for developing staged libraries.

Alternative implementations

Our choice of macros for the transformation is driven by the underlying efficient implementation of parser combinators. Any other efficient implementation or parser combinators that inlines function application would also do. For this we can either rely on the quality of a compiler's heuristics or use a use a partial evaluation framework [69].

We nonetheless need to inspect bodies of pure functions (for the SEQ) rules. We could restrict the parser combinator library to only use `map` and `fold` on products and sums, and specialize the functions passed to `CPSList`. The transformation could be written as a host language function. The disadvantage of such restrictions is the introduction of prohibitive overhead for a library user. It is preferable for us to do a little bit extra work by analysing function bodies in order to make the library have a more natural look and feel.

Related work

Stream-based parsing. The popularity of the XML format led to the development of SAX parsers, which are event-based parsers for this format [57]. With these, we can specify, using callbacks, what actions to perform when the parser encounters certain types. In the functional programming community, many stream processing libraries based on iteratees [45] can be used to build incremental parsers, which are a variant of event-based parsers. Among these, generators [46] are very popular in dynamic languages like Python as well. While these techniques improve performance on parsing fully upfront, they forsake the declarative nature of writing a query. Our transformation could be added on top of such parser combinators, thereby automatically generating efficient generators which skip unwanted input at source.

Query languages for data formats. There is a plethora of implementations of query languages for raw data formats. The most basic one is of course `awk` [1], which is particularly well suited for flat CSV files. XQuery [89] and XPath [90] are languages for querying XML files. There are also DSLs and libraries that help with JSON. JQ [17] is a command-line JSON processor, and Aeson [60] is a Haskell library for parsing JSON data into well typed datastructures. All of these languages need data to be fully loaded in memory. Therefore they support joins as well. We focus in this dissertation purely on preprocessing, where join-like queries are not prevalent.

There has been work on optimising XPath queries in a streaming setting [3, 12, 33, 31]. The goal in this work to be able to perform queries on the fly, as seen from events triggered by a SAX parser. The challenge is to handle many statically known queries at the same time. To achieve this a set of queries are analysed and converted into different types of state machines, with index structures that allow to identify early enough which queries are matched. Our work differs in that it does not depend on a specific format of a parser. While some of the work uses schema knowledge to perform lazy optimisations, we believe better parsing time can be achieved by introducing this knowledge into the parser directly. Moreover, the XML streaming work tries handles multiple queries, which we do not handle yet. Techniques proposed here can be used as a basis for future work.

Ludäscher et al. compile XQuery queries into a stream based paradigm called XSM [51]. They perform schema-based optimisations by creating specific events for tags that are part of the query. Also noteworthy is their composition optimisation which is reminiscent of fusion on list operations. The solution we propose, beyond applying to other data formats, is arguably simpler: instead of performing manual type inference for a query we delegate this job to the compiler by virtue of embedding. And we get fusion for free thanks to Church encodings and partial evaluation.

Marian et al. propose a method to pre-process an XML document to create a subset that is relevant for future processing [52]. Their solution is close in spirit to ours, in that they first define a domain-specific language for queries that they then statically analyse in order to project out required subsets of a document. Our approach shows that this can be taken to a streaming context, and that knowledge of language embedding can further simplify the analysis.

More recently, the NoDB [2] system allows querying of large, raw datasets that cannot fit into memory. They generate code that skips large sections of the data that is not needed. They mostly handle flat (CSV) like data. Our tranformation rules show that this is easily generalised to nested data structures as well. Our library can therefore be used as a middleware on top of which the rest of the query engine is built. Since NoDB assumes that queries can be executed more than once on the same dataset, they make use of positional maps to amortise future query times. Positional maps, from a combinator point of view, are simply augmented recognisers that return start and end positions for a given production.

Conclusion and future work

In this chapter we presented a set of transformation rules that specialise a query for a parser of a data format. This transformation seems worthwhile for two reasons. First, it is good programming practice to separate the flow of a parser from a query on the structures parsed. Second, as data sizes grow, it is really important, for good preprocessing performance, that unwanted data is thrown out as early as possible. Preprocessing is, all things considered, not the most crucial step in a data analysis framework (insights from the data are more important), and therefore should not consume prohibitive resources.

As immediate future work, we plan to measure the trade-offs in filter specialization (see Section8.6) more precisely to warrant the implementation of the extra rule. A natural consequence will be to release parsers for common data formats, so that the open-source community can benefit from this technique.

At present, many programmers still use generic parsers such as JSON or XML despite having a well specified data format. While augmenting our transformation to include recursion would alleviate their issues, a more principled approach seems to make it easier to write well typed parsers for a given format. Parser combinators seem to finally offer promise of performance, even for large datasets. We plan to explore the direction generating parsers for given data formats, especially by example, following recent progress in the field [49].

Synopsis

In this dissertation we set out to show that it is possible to implement efficient libraries for parser combinators and pipelines of operations on list-like collections such that the optimisations themselves can be expressed as libraries. Moreoever, we aimed to automatically interleave a query into a parser from a decoupled description of the same. We posited that these transformations would result in increased performance.

We achieved the first objective by developing a principled approach that combines Church encodings of data types with partial evaluation. We reduced dependence on the partial evaluator to function composition only by extracting optimisations of nested data types and conditional expressions to the library level. We then illustrated the applicability of this approach to a range of fusion algorithms on lists, and to two parser combinator libraries on far ends of the parsing spectrum. We showed that the approach was portable thanks to two implementations, each using a vastly different metaprogramming environment.

We achieved the second objective by treating repetition parsers as lists and hence recovering fusion properties discovered above. We then provided a set of transform rules that specialise a parser for a query. We implemented these rules in a macro. We evaluated this library to show that deforestation does pay off in practice as well.

Many programming languages have libraries and frameworks that allow to handle and operate on large amounts of data. While some languages' compilers can execute these programs efficiently, in many cases performance remains a major issues. Our hope is that developers can reuse the ideas presented here to implement optimised libraries in their ecosystem, such that they can avoid depending on a state-of-the-art optimising compiler.

Beyond parsers and queries, the work had us dive deep into the world of metaprogramming and program embedding in Scala. Our experience with these tools reveals that many tasks that are so easily expressed formally and in the literature remain difficult to implement: they either require writing boilerplate heavy code or a lot of (too much) knowledge of compiler internals. We are hoping that the implementation of the latest metaprogramming framework, `scala.meta`, will go far in alleviating these headaches. Often it is clear, for a given class of programs, which optimisations apply best. Yet a general purpose optimising compiler could only speculate, and potentially yields a sub-optimal result. Ideally a domain-specific library would selectively pick the optimisations most relevant to it. A good meta-programming tool would help democratise this process, by enabling future implementations of optimisations as libraries.

Bibliography

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[2] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’12, pages 241–252, New York, NY, USA, 2012. ACM.

[3] M. Altinel and M. J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB ’00, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

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

[5] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[6] E. Axelsson and M. Sheeran. Feldspar: Application and implementation. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP’11, pages 402–439, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] E. Béguet and M. Jonnalagedda. Accelerating parser combinators with macros. In *Proceedings of the Fifth Anuual Scala Workshop on - SCALA ’14*, pages 7–17, New York, New York, USA, 2014. ACM Press.

[8] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA ’13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

[9] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.

[10] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, Sept. 2009.

[11] 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.

[12] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. *The VLDB Journal*, 11(4):354–379, Dec. 2002.

[13] 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.

[14] D. Coutts. *Stream Fusion: Practical shortcut fusion for coinductive sequence types.* PhD thesis, University of Oxford, 2010.

[15] 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.

[16] M. Doenitz. spray-json, a lightweight, clean and efficient json implementation in scala. https://github.com/spray/spray-json, 2016.

[17] S. Dolan. jq: a lightweight and flexible command-line json processor.

[18] 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.

[19] A. Farmer, C. Hoener zu Siederdissen, and A. Gill. The hermit in the stream: Fusing stream fusion's concatmap. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 97–108, New York, NY, USA, 2014. ACM.

[20] 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.

[21] 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.

[22] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *Comput. J.*, 32(2):108–121, Apr. 1989.

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

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

[25] N. Ghani, P. Johann, T. Uustalu, and V. Vene. Monadic augment and generalised short cut fusion. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 294–305, New York, NY, USA, 2005. ACM.

[26] J. Gibbons and N. Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 339–347, New York, NY, USA, 2014. ACM.

[27] 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.

[28] 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.

[29] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

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

[31] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. *Processing XML Streams with Deterministic Automata*, pages 173–189. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[32] M. Guerrero, E. Pizzi, R. Rosenbaum, K. Swadi, and W. Taha. Implementing dsls in metaocaml. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 41–42, New York, NY, USA, 2004. ACM.

[33] A. K. Gupta and D. Suciu. Stream processing of xpath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 419–430, New York, NY, USA, 2003. ACM.

[34] R. Hinze, T. Harper, and D. W. H. James. Theory and practice of fusion. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 19–37, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[36] 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.

[37] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 007 1992.

[38] 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.

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

[40] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc, 2001.

[41] M. Jonnalagedda. Staged fold fusion, 2015. https://github.com/manojo/staged-fold-fusion.

[42] M. Jonnalagedda, T. Coppey, S. Stucki, T. Rompf, and M. Odersky. Staged parser combinators for efficient data processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '14*, pages 637–653, New York, New York, USA, 2014. ACM Press.

[43] M. Jonnalagedda and S. Stucki. Fold-based fusion as a library: a generative programming pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala - SCALA 2015*, pages 41–50, New York, New York, USA, 2015. ACM Press.

[44] A. Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 177–190, New York, NY, USA, 2007. ACM.

[45] O. Kiselyov. Iteratees. In *Proceedings of the 11th International Conference on Functional and Logic Programming*, FLOPS'12, pages 166–181, Berlin, Heidelberg, 2012. Springer-Verlag.

[46] O. Kiselyov, S. L. P. Jones, and A. Sabry. Lazy v. yield: Incremental, linear pretty-printing. In *APLAS*, pages 190–206, 2012.

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

[48] 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.

[49] A. Leung, J. Sarracino, and S. Lerner. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 565–574, New York, NY, USA, 2015. ACM.

[50] 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.

[51] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based xml query processor. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 227–238. VLDB Endowment, 2002.

[52] A. Marian and J. Siméon. Projecting xml documents. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 213–224. VLDB Endowment, 2003.

[53] S. Marlow and S. Peyton Jones. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications, Volume 2*. Lulu, 2012.

[54] E. Meijer, B. Beckman, and G. Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[55] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

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

[57] P. e. a. Murray-Rust. SAX. http://www.saxproject.org/.

[58] M. Odersky, J. Olson, P. Phillips, and J. Suereth. Sip-15 - value classes. http://docs.scala-lang.org/sips/completed/value-classes.html, 2012.

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

[60] B. O'Sullivan. Aeson: A json parsing and encoding library optimized for ease of use and high performance. https://hackage.haskell.org/package/aeson.

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

[62] S. Peyton Jones. Call-pattern specialisation for haskell programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 327–337, New York, NY, USA, 2007. ACM.

[63] S. Peyton Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12(5):393–434, July 2002.

[64] S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for haskell. *Sci. Comput. Program.*, 32(1-3):3–47, Sept. 1998.

[65] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1 edition, Feb. 2002.

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

[67] 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.

[68] T. Rompf, K. Brown, H. Lee, A. Sujeeth, M. Jonnalagedda, N. Amin, Y. Klonatos, M. Dashti, C. Koch, and K. Olukotun. Go meta! for a fundamental shift towards generative programming and dsls in performance critical systems. In *Proceedings of the Inaugural Summit on Advances in Programming Languages*, SNAPL 2015, 2015.

[69] 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.

[70] 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.

[71] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented dsls. In *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011.*, pages 93–117, 2011.

[72] 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.

[73] 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.

[74] 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.

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

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

[77] A. Shaikhha, M. D. Dashti, and C. Koch. Push vs. pull-based loop fusion in query engines. 2016.

[78] 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.

[79] 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.

[80] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 124–132, New York, NY, USA, 2002. ACM.

[81] J. Svenningsson, E. Axelsson, A. Persson, and P. A. Jonsson. Efficient monadic streams. In *Trends in Functional Programming*, 2015.

[82] B. J. Svensson and J. Svenningsson. Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 43–52, New York, NY, USA, 2014. ACM.

[83] 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.

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

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

[86] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, Jan. 1988.

[87] 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.

[88] 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.

[89] World Wide Web Consortium. W3c xml query (xquery).

[90] World Wide Web Consortium. Xml path language (xpath).

[91] 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.

[92] 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.

[93] 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.

[94] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

MANOHAR JONNALAGEDDA

PERSONAL INFORMATION

email	manohar.jonnalagedda@gmail.com
blog	http://manojo.github.io

INTERESTS

Functional Programming,
Domain Specific Languages,
Abstractions for Data Processing,
the Border between Theory and Practice

EDUCATION

Sep 2011–Present EPFL

PhD in Computer Science

Programming Languages Group · *Advisor:* Prof. Martin Odersky
I research embedded domain specific languages, their expressivity and performance. I have mostly worked on optimizing data processing pipelines by combining parsing, queries and generative programming. I currently maintain a blog with regular write-ups on these research efforts.

Fall 2006–Fall 2010 EPFL, UC Berkeley (exchange year)

Master in Computer Science

GPA: 5.75/6 · *Specialization in Foundations of Software*
I won the Prix de la Société Suisse d'Informatique for the second highest GPA in the Computer Science section.

PROJECTS

Packrat Parsing in Scala

Implemented and integrated packrat parsing, a novel parsing technique to the parser combinator library in Scala. The results of the project are now part of the Scala standard library.

Staged Fold Fusion

Implemented various fusion algorithms on list-like data structures using multi-staged programming, and discovered that these optimizations can be implemented as libraries (as opposed to compiler internals). These techniques form the basis of the current data processing and parsing research.

Parsequery

A macro-based Scala library that specialises data-processing queries for parsers written in a parser-combinator library. With this library it is easy to skip unwanted parts of the data instead of parsing it, resulting in significant memory savings and performance improvements.

Staged Parser Combinators

A parser combinator library that eliminates abstraction overhead for parsers. As a result, parsers written in Scala exhibit performance close to hand-tuned parsers written in C.

Lightweight Modular Staging

Contributed struct-based optimizations to the Lightweight Modular Staging Framework, a runtime code-generation approach for building DSLs.

ExPASy

Fully redesigned and developed ExPASy, one of the main bioinformatics resources portal for proteomics in the world.

EMPLOYMENT

Sep 2011–Present PhD Student

EPFL, Lausanne, Switzerland

Supervisor: Martin Odersky
Research student on the academic side of the Scala programming language team. My main research topic is DSLs and their optimisation. I also supervise student projects and am head TA of the undergraduate Functional Programming class.

Jun 2013–Sep 2013 Research Intern

Oracle Labs, Lausanne, Switzerland

Supervisor: Tiark Rompf
During this internship I came up with the initial design, and implemented a prototype, of the staged parser combinator project. Post internship, I developed benchmarks and published the results.

Sep 2010–Dec 2011 Software Engineer, Civil Service

Swiss Institute of Bioinformatics, Lausanne, Switzerland

Supervisors: Heinz Stockinger, Aurélien Grosdidier
Worked on various projects with the Swiss Institute of Bioinformatics. The most important result of my work was a full re-design and development of Expasy, one of the main bioinformatics resources for proteomics in the world.

Feb 2010–Sep 2010 Intern, Master's Thesis

Siemens Corporate Technology, Munich, Germany

Supervisors: Michael Jaeger, Gerald Kaefer
Worked on a cloud-computing related Masters thesis: "Achieving application symmetry between on-premises and cloud platforms". This project resulted in key guidelines that Siemens used for adopting the cloud for current/future projects.

TEACHING EXPERIENCE

MOOCs

Co-designed and mentored the course on Functional Programming in Scala, taken by $>$ 200000 students to date. My main contribution was the design and implementation of the popular Bloxorz homework on lazy evaluation.

Project Supervision

Supervised 5 master level students on various programming language research related projects, yielding 2 peer- reviewed publications in international conferences.

Teaching Assistant

As head TA for the undegraduate functional programming course, I organized and led recitation sessions, homework projects and exam sessions for 3 consecutive years, each iteration of the course followed by more than 150 students. This course was the mirror of the Functional Programming MOOC.

SELECTED PUBLICATIONS

Scala '15 Fold-based Fusion as a Library: A Generative Programming Pearl
Co-authors: Sandro Stucki

OOPSLA '14 Staged Parser Combinators for Efficient Data Processing
Co-authors: Thierry Coppey, Sandro Stucki, Tiark Rompf, Martin Odersky

Scala '14 Accelerating Parser Combinators with Macros
Co-authors: Eric Béguet

April 13, 2016