UNIVERSITY OF GOTHENBURG

Implementing incremental and parallel parsing

A subtitle that can be rather long

Master of Science Thesis in Computer Science

TOBIAS OLAUSSON

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2014

Implementing incremental and parallel parsing
A subtitle here that can be quite long

Tobias Olausson

Examiner: Patrik Jansson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: an image that is used as a cover image

Department of Computer Science and Engineering
Göteborg, Sweden, May 2014

Abstract

This is an abstract

Contents

Chapter 1

Background

1.1 Introduction

The topic of this thesis is to do **parsing** in an **incremental** fashion that can easily be **parallelizable**, using a **divide-and-conquer approach**. In this section, I will give a brief explanation of the topics covered, and end with a motivation for why this is interesting to do in the first place.

1.1.1 Divide-and-conquer

Add stuff here from section 2 in parparse paper.

1.1.2 Incrementality

Doing something incrementally means that one does it step by step, and not longer than neccessary.

1.1.3 Parallelism

In modern day processors, rather than increasing the clock frequency, efforts are put into building processors with many cores, being able to run instructions in parallel. Programmers have for many years written code that runs several instructions seemingly simultaneous, even on single-core processors. With multi-core processors this can be done truly in parallel. Since divide-and-conquer algorithms usually work on several independent sub-problems, they are well-suited for parallelization. For this to become a reality, however, both the compiler and the source code must be written in a certain way to permit parallelization.

1.1.4 Parsing

To parse is a to check if some given input corresponds to a certain language's grammar, and in this thesis I will use **context-free grammars** for programming languages.

1.1.5 Motivation

In compilers, lexing and parsing are the two first phases. The output of these is an abstract syntax tree (AST) which is fed to the next phase of the compiler. But an AST could also provide useful feedback for programmers, already in their editor, if the code could be lexed and parsed fast enough. With a lexer and parser that is incremental and that can also be parallelized could real-time feedback in the form of an AST easily be provided to the programmer. Most current text editors give syntax feedback based on regular expressions, which does not yield any information about depth or the surrounding AST.

Something something about connecting to a type-checker.

1.2 Lexing

In compilers, a *lexer* reads input source code and groups the characters into sequences called *lexemes* so that each lexeme has some meaning in the language the compiler is built for [ALSU, p. 5, p. 109]. The lexemes are wrapped in *tokens* that denote what function and position each lexeme has. The tokens are then passed on to the parser for syntactic analysis.

For a language like C, the code in figure 1.1 would be valid, and can serve as an example of how lexing is done.

```
while(i < 5) {
    i++;
}
```

Figure 1.1: A while loop that would be valid in C

A lexer would recognise that *while* is a keyword and place it in its own token. It would also observe that (,), { and } are used for control grouping of code. Furthermore, i is an identifier and 5 is a number, < and ++ are operators and ; denote separation of statements. All of these will be forwarded as tokens to the parser.

1.2.1 LexGen

As a master's thesis, a generator for incremental divide-and-conquer lexers was developed in 2013 by Hansson and Hugo [HH14]. Since the aim of this thesis is to write an incremental divide-and-conquer parser, their work is well-suited as a starting point, and as something to build on. Their lexer utilised Alex for core lexing routines and relied heavily on the use of arrays and finger trees, which we will see more of later.

1.3 Context-free grammars

Context-free grammars are a way to describe formal languages, such as programming languages. They describe both the alphabet of a language and the rules for how to combine words in that language.

Formally, a context-free grammar is a 4-tuple: $G = (V, \Sigma, P, S)$ [HMU03, p.171]. V is a set of non-terminals, or variables. Σ is the set of terminal symbols, describing the content (or alphabet) that can be written in the language. P is a set of productions (or rewrite rules) that constitutes the recursive definition of the language. S is the start symbols, where $S \in V$.

The language recognized by a context-free grammar G is denoted $L(G)$ and is defined as

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

That is, all words in the language that can be derived using the rules from the grammar and starting from the start symbol [HMU03, p. 177]. A language L is said to be context-free if there is a context-free grammar G that recognizes the language, meaning that $L = L(G)$.

We can exemplify this using a simple made-up language of if-then-else clauses. The language terminals are *if, then, else* and *true and false*. There are two variables, I (for if) and R (for recursive) described by a total of four productions. The starting symbol is I - so just writing *true* would not be valid for this language. The formal definition of such a language can be seen in figure 1.2.

$$
\begin{aligned}
&G = (V, \Sigma, P, I) \\
&V = \{I, R\} \\
&\Sigma = \{true, false, if, then, else\} \\
&I \rightarrow if\ R\ then\ R\ else\ R \\
&R \rightarrow I \\
&R \rightarrow true \\
&R \rightarrow false
\end{aligned}
$$

Figure 1.2: Context-free grammar for a recursive if-then-else language

1.3.1 Backus-Naur Form

TODO: John Backus wrote a paper on this, cite it? See compling-essay.

Context-free grammars are often used to describe the syntax of programming languages. Such descriptions are often given in a **labelled Backus-Naur form**, where each rule is written on the following form:

$$Label.\ Variable ::= Production$$

Just link to BNFC here?

Grammars written in Labelled Backus-Naur Form are also used in the **BNF Converter (BNFC)**, a lexer and parser generator tool developed at Chalmers. Given such a grammar, BNFC generates, among other things, a lexer and a parser, implemented in one of several languages, for the language described in that grammar. According to its documentation, usage of BNFC saves approx 90% of source code work in writing a compiler front-end.

1.3.2 Chomsky Normal Form

Chomsky Normal Form (CNF) is a subset of context-free grammars that was first described by linguist Noam Chomsky. Productions in CNF are restricted to the following forms:

A	$\rightarrow$	BC,	A is a variable, B and C are productions
A	$\rightarrow$	a,	A is a variable, a is a terminal symbol

Figure 1.3: Rules allowed in Chomsky Normal Form

Since grammars in CNF are restricted to branches or single terminal symbols, they are well suited for usage in divide-and-conquer algorithms.

There are several existing algorithms to convert context-free grammars into CNF, so one does not have to write their grammars in CNF in the first place [LL09].

1.4 Parsing

1.4.1 CYK algorithm

1.4.2 Valiant

1.4.3 Improvement by Bernardy & Claessen

Running time analysis. Oracle for list.

1.5 Dependently typed programming

In a strictly typed programming language like Haskell, every value has a type that is enforced. Assigning an integer a floating-point value would not type-check and therefore would not compile. While this is useful and saves debugging time, it does not say anything about the contents of the values.

A motivating example often used is implementation of a vector type. Vectors in this case is a fixed-length list of some type. In a typical Haskell setting we may have a type as follows:

```
data Vec a = Nil | V a Vec

head :: Vec a -> a
head Nil = error "empty vector"
head (V a _) = a
```

Figure 1.4: Vector type and head function

This looks good, but it has the inherent problem that any code that tries to access the head of an empty Vec will compile but result in a runtime error.

In dependently typed programming, types may contain other types, acting as values, so that they relate the same way a value relates to its type. While Haskell is not a dependently typed programming language, there are ways to use dependent types even in Haskell. For our vector example that would look something like this:

```
data Nat = Z | S Nat
data Vec a n where
    Nil :: Vec a Z
    V :: a → Vec a s → Vec a (S s)

head :: Vec a (S b) → a
head Nil = error "empty vector" -- this does not type-check
head (V a _) = a
```

Figure 1.5: Dependently typed vector with head function

We created a new type Nat (for natural numbers) to keep track of the size of our vector. The new Vec type is *dependent* on the Nat type, while still holding values of some type `a`. What this code does not permit, however, is the Nil case for head. Because the type of head requires the Vec to be non-nil (with (S b) in its type signature) there is no need to check for a Nil vector here. In fact, the compiler will not pass the above code, as indicated by the comment, since the first case in head does not type check. The main advantage of this vector type is that any code that uses head and passes type-checking will be guaranteed to never encounter an empty vector. This way, even more bugs are caught at compile-time.

Chapter 2

Implementation

Before going into details about the implementation of this parser, there are a couple of libraries and programming techniques one has to be familiar with before moving forward. I will first describe those, and then move on to describe changes to the lexer I inherited, and the implementation of the parser.

2.1 Finger trees

A finger tree is a finite data structure with logarithmic time access and concatenation. The finger tree is similar to a general binary tree, where each branch has a couple of *fingers* (values) so that adding a new value does not neccessarily add a new branch to the tree. The tree structure makes the data structure suitable for a divide-and-conquer algorithm. TODO: Referens till paper om finger trees

A Haskell implementation suitable for everyday needs exists in the `Data.Sequence` package, and a more general structure is available in the `Data.FingerTree` package. The more general one is the one that will be used for this project, and is the one that was used for the LexGen project.

2.1.1 Measuring and Monoids

Two specific features in the general FingerTree data type are monoids and measuring. These are fundamental to the parser, so we will look more deeply into them here.

A *monoid* is a mathematical object with an identity element and an associative operation. In Haskell, this is provided by writing instances of this type class:

TODO: Src-referens

```
class Monoid a where
    -- Identity of mappend
    mempty  :: a
    -- An associative operation
    mappend :: a -> a -> a
```

Figure 2.1: The Monoid type class

This means that anything that is a Monoid has an identity element (that can be accessed with *mempty*) and an associative operation to append monoids together (*mappend*). A simple list example illustrates this:

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

Figure 2.2: Monoid instance for lists

TODO: Say something more about Monoids?

The FingerTree type has a notion of being able to *measure* its elements. In this case, to measure means to have a function that, given an element of the type the FingerTree contains, yields a value of some type – the measure of that element. Furthermore, any type that the elements can be measured to has to be a monoid. The existance of a measured instance is ensured by the FingerTree API.

TODO: src-referens

```
-- Things that can be measured
class Monoid v => Measured v a | a -> v where
    measure :: a -> v

-- FingerTrees are parametrized on both v (measures) and a (values)
data FingerTree v a

-- Create an empty finger tree
empty :: Measured v a => FingerTree v a
```

Figure 2.3: Measuring and the FingerTree type

This means that, in order to use the FingerTree, one need to fulfil a few criteria first. Let's say you want to have a FingerTree of Strings and that the measure should be the (combined) length of the Strings, then your type would be `FingerTree Int String`. For that to work, you first need to be able to convert between Strings and Int, by writing an instance of Measured for String Int. This would typically just be the length function. However, for that to work you need a Monoid instance for Ints. Perhaps it would look something like this:

```
type MyTree = FingerTree Int String
instance Measured String Int where
    measure = length
instance Monoid Int where
    mempty = 0
    mappend = (+)
```

Figure 2.4: One possible measure from String to Int

It should be noted however, that since instances cannot be hided, writing a general Monoid instance for Ints over addition is perhaps not the best idea. Wrapper types with instances over addition and multiplication are available in the `Data.Monoid` library.

2.2 Lexing

For lexing code into tokens, the results from the LexGen project was used. However, some modifications had to be done to LexGen in order to be easily generated from BNFC as an Alex file. One large change was done to the lexing code, however, which is due to the fact that not only the lexer, but also the parser, should work incrementally. In the LexGen code, the output structure is a `Sequence` of tokens. Since `Sequence` is a less general implementation of finger trees, they cannot be measured, and is therefore not as suitable to use in an incremental setting. Hence, instead of outputting tokens as a `Sequence`, the code was changed to output as another `FingerTree`, from which the tokens could then be measured (see Pipeline of measures below).

2.3 Parsing

TODO: Move this part down? Duplicate in BNFC TODO: An illustration here perhaps?

Observing that the lexer could easily be generated from BNFC, it was natural to plug it in, and given that, the parser code is so general that it can also be generated from any LBNF grammar.

2.3.1 BNFC

There was an existing reference implementation in BNFC for the optimisation to Valiant's algorithm, that could be accessed using the `--cnf` option. That option generated large tables needed for combining different tokens. Since the this project is similar to the reference implementation, it was natural to generate the new parser by using a new option.

For the Haskell backend, BNFC uses Alex as a lexer, as did LexGen, so it was easy to use the LexGen core and have BNFC and Alex generate the DFA needed for the lexer to work.

2.3.2 Pipeline of measures

TODO: Clarify that ONE char does not become the whole AST. Tree structure!

Using the `FingerTree` type, the lexer could use that data type to measure characters into an intermediate type for lexing. That intermediate `FingerTree` could then in turn be measured to the type used for parsing.

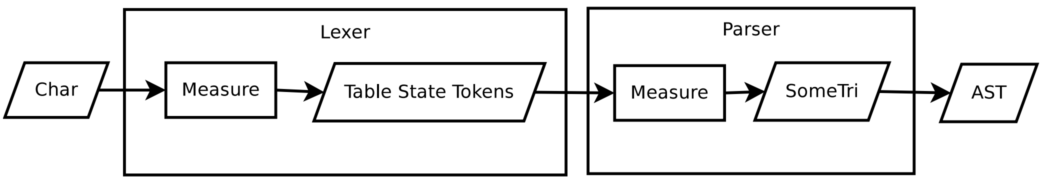

Figure 2.5: Diagram of measuring pipeline

Looking at simple testing code shows easily how the data progresses through the pipeline. `stateToTree` is an auxiliary function extracting a `FingerTree` from the internal lexer state.

```
test :: FilePath → IO (SomeTri [(CATEGORY,Any)])
test filename = do
    file ← readFile filename
    let mes = measure $ makeTree file
        tri = measure $ stateToTree mes
    return (results tri)
```

Figure 2.6: Code showing the measuring pipeline

2.3.3 Dependently typed programming with charts

The existing code. Illustration needed here.

2.3.4 Oracle and unsafePerformIO

The use of an oracle presented a bit of a problem in implementing a monoid instance for the parser, for the simple reason that it is very hard to simply pick a bool at random in Haskell without it being always `True` or always `False`. The call to `merge` in `mappend` illustrates this clearly:

```
instance Monoid (SomeTri a) where
    t0 `mappend` t1 = merge True t0 t1
```

Figure 2.7: Parser Monoid instance before random oracle

The call to `merge` requires a `Bool` acting as the oracle as an argument. Since a Monoid has no context outside its own type, it is hard, if not impossible, to generate a Bool using only the SomeTri type. One could argue for creating a newtype wrapper around a tuple of a SomeTri and a StdGen, used to generate the bool at each step, but that only moves the problem to the mempty call, where a fresh StdGen would have to be picked at each instance.

The solution to this problem came in the form of a call to unsafePerformIO. While this is controversial, it was also not completely obvious to implement. If an unsafe call to randomIO was made separately from the merge call, this call would be evaluated only once, rendering the solution useless. The trick here was to put the whole call inside an unsafe wrapper, so that the call to merge, and with that the call to randomIO, became dependent on the input.

```
instance Monoid (SomeTri a) where
    t0 `mappend` t1 = unsafePerformIO $ do
      b ← randomIO
      return $ merge b t0 t1
```

Figure 2.8: Parser Monoid instance with oracle

Now usually, for `unsafePerformIO` to be safe one should make sure that the call is free from side effects and *independent of its environment.* [Com]. Since none of those two requirements are fulfilled here this calls for discussion. In general, one does not want a call to unsafePerformIO to be evaluated more than once – but in this case this is a requirement for the code to behave as expected, and that's why it is indeed dependent on its environment. The only side effect in this snippet is the use of the global random number generator and that should not affect any other part of the program, and can thus be considered safe.

Chapter 3

Results

3.1 Final product

3.1.1 Testing

3.2 Measurements

How fast is it? What is the complexity?

Chapter 4

Discussion

4.1 Pitfalls

Look through the LOG to remember whatever happened. Describe sort of chronologically?

4.1.1 Too many result branches

Describe the reason for this, with the merge stuff.

4.1.2 Position information

* Tuple of monoids? * RingP instance? * Newtype for tuples?

4.2 Future work

Bibliography

[ALSU] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman.

[Com] The Glasgow Haskell Compiler. *Hackage - System.IO.Unsafe.* http://hackage.haskell.org/package/base-4.7.0.0/docs/System-IO-Unsafe.html.

[HH14] Christoffer Hansson and Jonas Hugo. A generator of incremental divide-and-conquer lexers. Master's thesis, Chalmers University of Technology, 2014.

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed).* Addison-Wesley, 2003.

[LL09] Martin Lange and Hans Leiß. To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm. *Informatica Didactica*, 8, 2009.