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, April 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, April 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

Why is parallelism interesting and how does it apply in this case?

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 it 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

* Describe what lexing is * Shortly describe LexGen and its relevance.

1.3 Context-free grammars

Give a light-weight description here.

Formally, a context-free grammar is a 4-tuple: $G = (V, \Sigma, P, S)$. V is a set of non-terminals, or variables. Σ is the set of terminal symbols, describing the content that can be written in the language. P is a

1.3.1 Backus-Naur Form

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:

$$Label.Variable ::= Production$$

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 documentation, usage of BNFC saves approx 90% of source code 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 variable, B and C productions
A	$\rightarrow$	a,	A variable, a terminal symbol

Figure 1.1: 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.

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 dependently typed programming, types are dependent on values.

1.5.1 Kinds, Types and Values

Describe how kinds relate to types as types relate to values.

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.

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 the need for Measuring and Monoids. 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:

```
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*).

For the FingerTree type, anything that can be measured, has to also be a monoid. This is ensured not by the data type itself, but by every function operating on the finger tree.

```
-- 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 2.2: Measuring and the FingerTree type

This means that, in order to use the FingerTree type parametrized on some type a, one has to have:

1. Another type v, such that $a \rightarrow v$ is a functional dependency
2. A monoid instance of v
3. An instance measured v a

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.

2.2.1 Position information

2.3 Parsing

2.3.1 BNFC

2.3.2 Pipeline of measures

An illustration would be good here

2.3.3 Dependently typed programming with charts

2.3.4 Oracle and unsafePerformIO

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