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. To parse is a to check if some given input corresponds to a certain language's grammar, and in this case context-free programming languages' grammars.

1.1.1 Incrementality

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

1.1.2 Parallelism

1.1.3 Parsing

1.1.4 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.

1.2 Lexing

Shortly describe LexGen and its relevance.

1.3 Context-free grammars

1.3.1 Chomsky Normal Form

1.4 Parsing

1.4.1 CYK algorithm

1.4.2 Improvement by Bernardy & Claessen

1.5 Dependently typed programming

What is dependently typed programming, and how can it be used in Haskell.

1.5.1 Kinds, Types and Values

Chapter 2

Implementation

2.1 Finger trees

2.2 Measuring and Monoids

2.2.1 Pipeline of measures

An illustration would be good here

2.3 Lexing

2.3.1 Position information

2.4 Parsing

2.4.1 BNFC

2.4.2 Dependently typed programming with charts

2.4.3 Oracle and unsafePerformIO

2.5 Final product

2.5.1 Testing

Chapter 3

Results

3.1 Measurements

How fast is it? What is the complexity?

Chapter 4

Discussion

4.1 Implementation

4.1.1 Too many result branches

4.1.2 LexGen – Alex discrepancy

4.1.3 Position information

4.2 Improvements

4.3 Future work