University of Bristol

Jamie Willis
Supervised by Dr. Nicolas Wu

Parsley: The Fastest Parser Combinator Library in the West

Introduction

Parsing is an important problem in Computer Science. Many practical applications require some amount of parsing, for instance JSON or XML parsing.

When faced with parsing a grammar, we could make use of parser generating languages like *Antlr*. But these require us to work outside of the domain of the program and some potential for more powerful parsers is lost. We could also hand-write parsers, but this is cumbersome and error-prone.

Parser Combinators

Functional Programming has given us a third option: parser combinators. They keep the representation of the grammar in the implementation language itself allowing for full use of its features and syntax:

```
--some :: Parser a -> Parser [a]
--oneOf :: [a] -> Parser a
--(<$>) :: (a -> b) -> Parser a -> Parser b
nat :: Parser Int
nat = toNat <$> (some digit)
 where digit :: Parser Int
       digit = digitToInt <$> (oneOf ['0'..'9'])
       toNat :: [Int] -> Int
       toNat = foldl1' (\x n -> x * 10 + n)
```

They can represent the grammar quite directly and allow us to capture parsing results in the structure we desire:

```
data Expr = Add Expr Expr | Nat Int
expr :: Parser Expr
expr = chainl1 (Nat <$> nat) (Add <$ char '+')
```

The Problem

Parser combinator libraries have been traditionally quite slow! Depending on the style of parsing, they have also been known to leak space.

In particular, the programming language Scala's libraries are either quite cumbersome to use or are not performant, or both!

The Solution: Parsley

My contribution is Parsley, a combinator library written in Scala which is designed to provide the full sugar of the popular Parsec style of combinators found in Haskell with the fastest performance that I can possibly squeeze out.

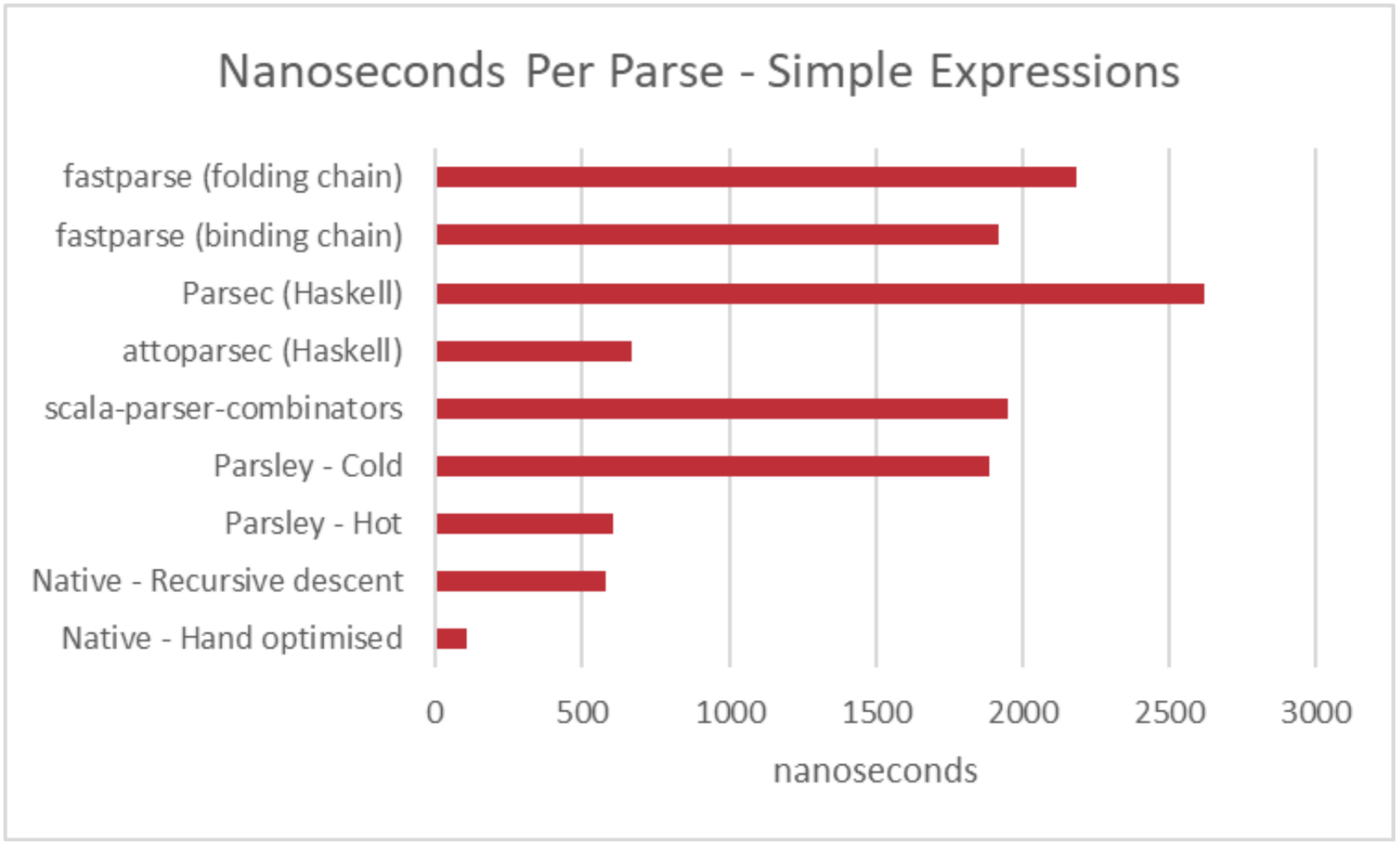
The secret is to actually compile the combinators at runtime to a *highly* optimised stack-based bytecode (also in Scala, in imperative style) as a DSL. This machine on its own has promising performance but the icing on the cake is the optimisations the compiler can perform on the combinators which kicks the speed through the roof. Many useful optimisations can be derived from the Applicative Functor laws alone, for example.

Challenges and Remaining Work

It's not a free lunch though; there are still a lot of work to be done. For instance, from a theoretical stand-point, it would be nice to be able to prove the correctness of the machine itself but this can be quite difficult.

There are also many practical challenges to be overcome. For instance, handling recursive parsers requires fixpoints; in order to even calculate these the parsers must be incredibly *lazy* and this can impact compilation performance. In addition, it would be nice to make efficient use of mutability to boost performance, but ensuring the correctness of the optimisations in a mutable environment is difficult. The *impurity* in Scala itself also poses a problem; optimisations often rely on the fact that parsers are *pure* but users can make *impure* parsers and this must disable certain optimisations with minimal user interaction.

Current Benchmarks


Nanoseconds Per Parse - Simple Expressions
fastparse (folding chain)
fastparse (binding chain)
Parsec (Haskell)
attoparsec (Haskell)
scala-parser-combinators
Parsley - Cold
Parsley - Hot
Native - Recursive descent
Native - Hand optimised
nanoseconds