Parslers: A Staged Selective Parser Combinator Library for the Rust Programming Language

Jordan Hall

2023

Imperial Computing (BEng) Final Year Project Presentation

Introduction to Parsing

Parsing is the act of converting one data format to another.

Most commonly, converting a sequence of characters into data a computer can manipulate.

Types of Parsing libraries

- Handwritten Parsers
- Parser Generators
- Parser Combinators

Parser Combinators

Parser Combinators work by combining higher-order functions together. They are written in the host language, typically as a module.

Advantages of Parser Combinators

Parser combinators are very expressive, being Turing complete. It is possible to create full interpreters entirely expressed as parser combinators.

Parser combinators are embedded into the host language, unlike parser generators.

Disadvantages of Parser Combinators

- Parser combinators are generally considered slower than handwritten parsers and parser generators. This makes handwritten parsers the default route for high-performance parsing.
- Parser combinators typically have poor error messaging support, as they can’t generate tailored error messages, unlike parser generators.

Fixing the Structural Problem (cont.)

Most parser combinator libraries express their parsers as monads.

This poses many limitations on the optimisation of parsers.

Fixing the Structural Problem

A monad is a monoid in the category of endofunctors.

Fixing the Structural Problem (cont.)

A monad is a type `M` with the two following functions:

```
pure :: a -> M a
bind :: (M a) -> (a -> M b) -> (M b)
```

Fixing the Structural Problem (cont.)

Parser combinators are generally expressed as monads where:

`pure :: a -> Parser a` is the parser that returns a value and consumes no input.

`bind :: (Parser a) -> (a -> Parser b) -> (Parser b)` produces a new parser based on the output of the previous parser.

Fixing the Structural Problem (cont.)

`bind` can produce parsers whose structure is not known at compile-time.

Monadic parser combinators suffer from this lack of knowledge.

They can't be optimised like their handwritten parser or parser generator counterparts.

Fixing the Structural Problem (cont.)

We may only want only to parse numbers within a certain range (like an unsigned 8-bit integer):

```
u8 = bind parseNumber ( \x -> if 0 <= x <= 255 then pure x else Fail )
```

This is not possible using only the applicative parsers.

Fixing the Structural Problem (cont.)

The selective parser is defined as:

```
branch :: Parser (Either x y)
           -> Parser (x -> z)
           -> Parser (y -> z)
           -> Parser z
```

This allows the parser to make runtime decisions while maintaining a fixed structure, known at compile time.

Fixing the Structural Problem (cont.)

Now, the unsigned 8-bit integer parser is defined as:

```
u8 = branch (map select parseNumber) fail (pure id)
    where
        select :: Int -> Either () Int
        select x = if 0 <= x <= 255 then Right x else Left ()
```

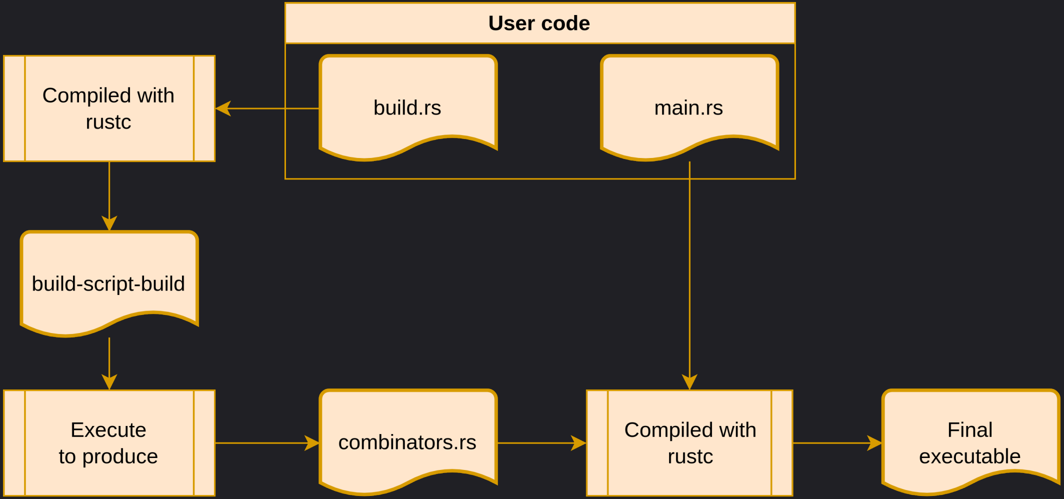
This parser’s structure can now be traversed at compile-time.

How Parslers Works

Parslers allows its users to write parsers in pure Rust as a domain-specific language.

It uses Rust’s staged compilation model to optimise user-written parsers and generate native Rust code at compile time.

How Parslers Works (cont.)



Main Contributions of the Project

Parslers stands as the fastest general-purpose parsing library for the Rust programming language.

This is because of the optimisations available at compile time both by Parslers and the Rust compiler.

Main Contributions of the Project (cont.)

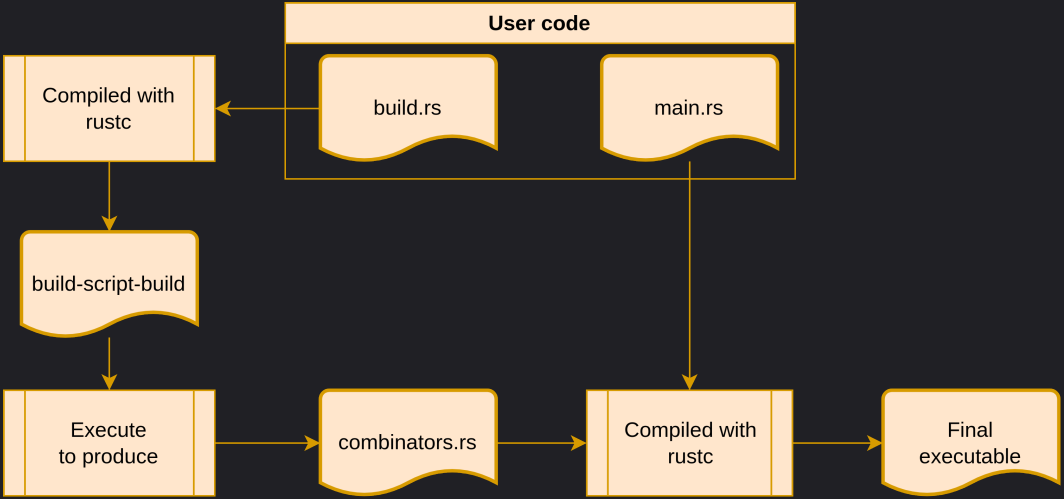
For Parslers to exist, a new library also had to be created.

The values produced by `pure` parsers exist in the build.rs file. The definitions are lost during the compilation of the build.rs file.

Even worse, the definitions of functions in the build.rs file can't be inspected at compile time in Rust.

If Parslers is staged, then the definition of functions and the values of `pure` parsers need to be preserved.

How Parslers Works (cont.)



Main Contributions of the Project (cont.)

This problem is not limited to Parslers.

Lacking reflection is the main reason staged domain-specific languages are not readily available in the Rust ecosystem.

Main Contributions of the Project (cont.)

The Reflect library acts as a general solution to the cross-staged persistence commonly needed in staged DSLs.

```
fn is_a(c: char) -> bool {
    c == 'a'
}
```

Main Contributions of the Project (cont.)

The Reflect library acts as a general solution to the cross-staged persistence commonly needed in staged DSLs.

```
#[reflect]
fn is_a(c: char) -> bool {
    c == 'a'
}
```

Main Contributions of the Project (cont.)

```rust
struct is_a;

impl FnOnce<(char,)> for is_a {
    type Output = bool;
    extern "rust-call" fn call_once(self, (c,): (char,)) -> bool {
        c == 'a'
    }
}
impl Reflect for is_a {
    fn reflect(&self) -> String {
        "fn is_a(c : char) -> bool { c == \'a\' }".to_owned()
    }
}
```

Main Contributions of the Project (cont.)

This method can also be used to retrieve the definitions of the values produced by `pure` parsers:

```rust
impl Reflect for u8 {
    fn reflect(&self) -> String {
        format!("{}u8", self)
    }
}
```

Main Contributions of the Project (cont.)

This method can also be used to retrieve the definitions of the values produced by `pure` parsers:

```rust
#[derive(Reflected)]
pub struct BranflakesProgram(pub Vec<Branflakes>);

#[derive(Reflected)]
pub enum Branflakes {
    Add,
    Sub,
    Left,
    Right,
    Read,
    Print,
    Loop(BranflakesProgram),
}
```

Main Contributions of the Project (cont.)

The Reflect library is generic to a restricted form of cross-staged persistence intended only for creating staged, domain-specific languages in Rust.

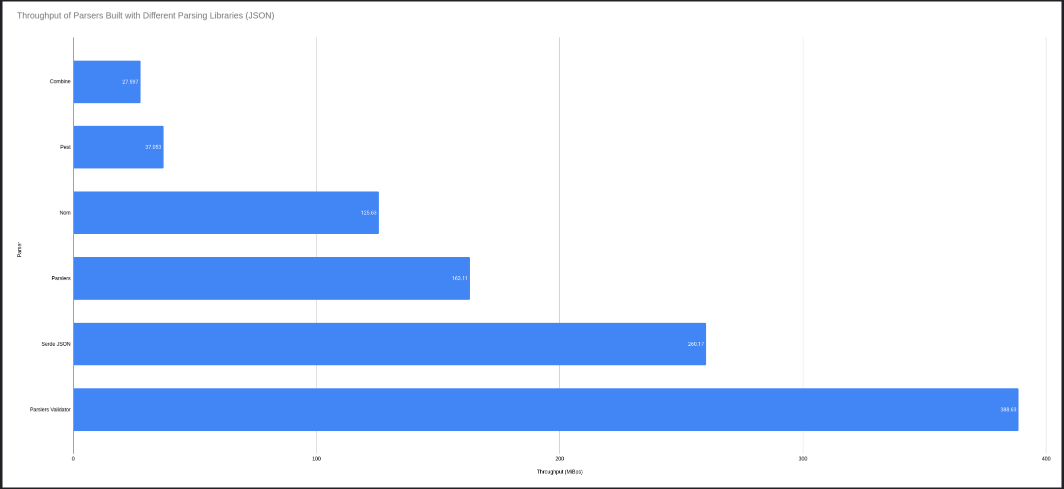
This provides the possibility for more staged DSLs to exist within the Rust ecosystem.

Demonstration

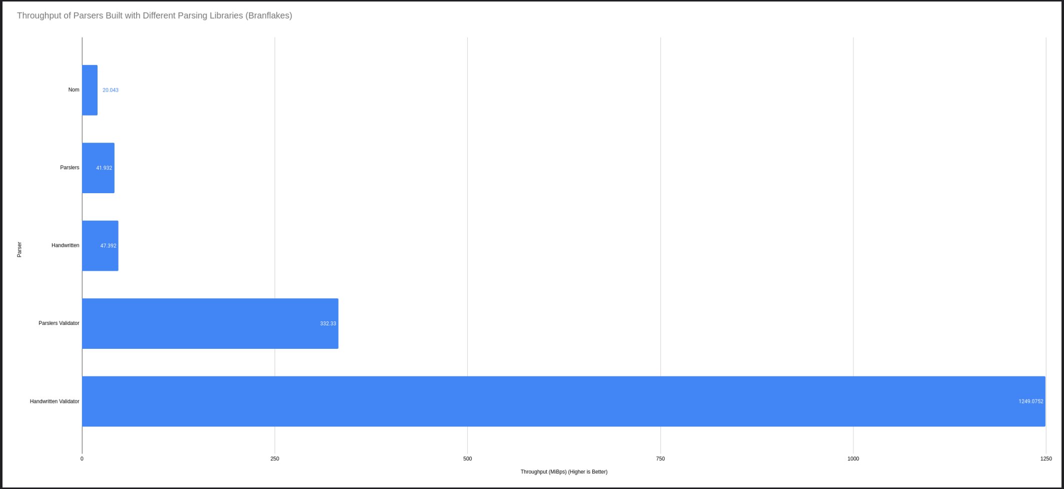
How is Parslers different from other parser combinator libraries?

What makes Parslers distinct from parser generator libraries?

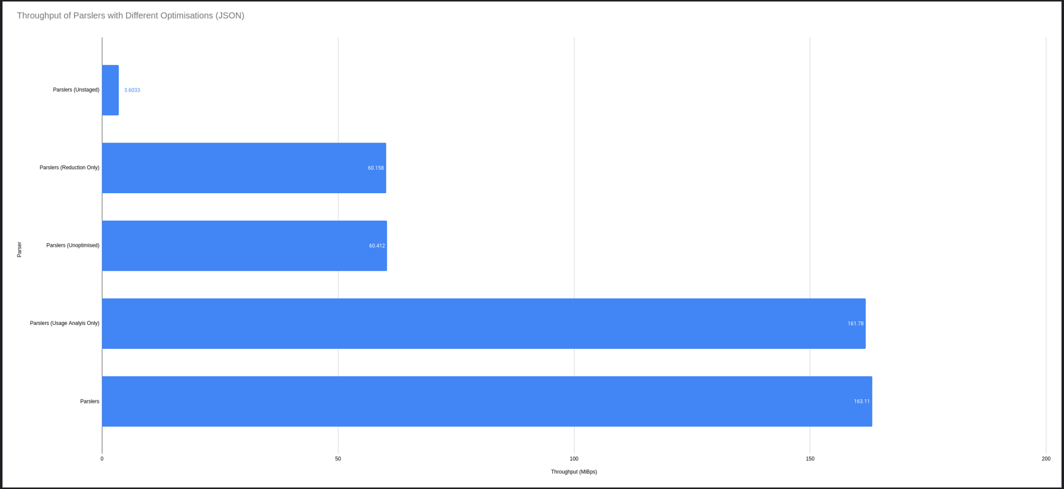
Performance of Parslers



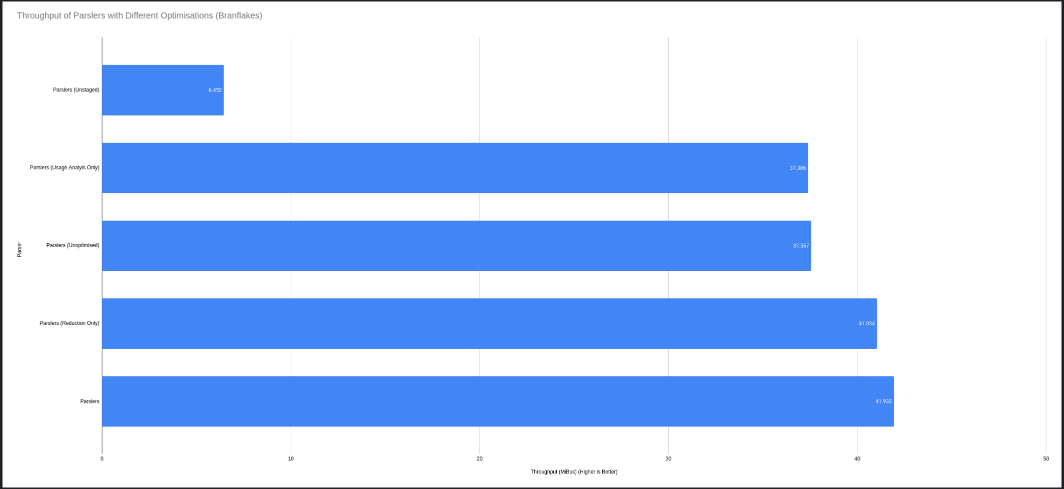
Performance of Parslers (cont.)



Performance of Parslers (cont.)



Performance of Parslers (cont.)



Conclusion

Parslers now stands as the fastest generalised parsing library for the Rust programming language.

The Reflect library lays the foundation for other staged domain-specific languages to be built for the Rust ecosystem.

Future Work

- Many more optimisation passes to be integrated into Parslers, further closing the gap between parsing libraries and hand written parsers.
- Automatic error messaging would make Parslers a fully ready-to-use parsing library.

Future Work (cont.)

- Adding buffered and asynchronous parsers should be possible in Parslers without any modification to the user-written parsers.
- Registers will increase the expressive power of parsers written with Parslers

The End

Thank you for your time!