agdarsec – Total Parser Combinators

Guillaume Allais[1]

Radboud University, Nijmegen, The Netherlands
guillaume.allais@ens-lyon.org

Abstract

Parser combinator libraries represent parsers as functions and, using higher-order functions, define a DSL of combinators allowing users to quickly put together programs capable of handling complex recursive grammars. When moving to total functional languages such as Agda, these programs cannot be directly ported: there is nothing in the original definitions guaranteeing termination.

In this paper, we will introduce a 'guarded' modal operator acting on types and show how it allows us to give more precise types to existing combinators thus guaranteeing totality. The resulting library is available online together with various usage examples at https://github.com/gallais/agdarsec.

1 Introduction

Parser combinators have made functional languages such as Haskell shine. They are a prime example of the advantages Embedded Domain Specific Languages provide the end user. She not only has access to a library of powerful and composable abstractions but she is also able to rely on the host language's existing tooling and libraries. She also gets feedback from the static analyses built in the compiler (e.g. type and coverage checking) and can exploit the expressivity of the host language to write generic parsers thanks to polymorphism and higher order functions.

However she only gets the guarantees the host language is willing to give. In non-total programming languages such as Haskell this means she will not be prevented from writing parsers which will unexpectedly fail on some (or even all!) inputs. Handling a left-recursive grammar is perhaps the most iconic pitfall leading beginners to their doom: a parser never making any progress.

We start with a primer on parser combinators and follow up with the definition of a broken parser which is silently accepted by Haskell. We then move on to Agda [12] and introduce combinators to define functions by well-founded recursion. This allows us to define a more informative notion of parser and give more precise types to the combinators commonly used. We then demonstrate that broken parsers such as the one presented earlier are rejected whilst typical example can be ported with minimal modifications.

2 A Primer on Parser Combinators

2.1 Parser Type

Let us start by reminding ourselves what a parser is. Although we will eventually move to a more generic type, Fritz Ruehr's rhyme gives us the *essence* of parsers:

A Parser for Things
is a function from Strings
to Lists of Pairs
of Things and Strings!

This stanza directly translates to the following Haskell type (we use a **newtype** wrapper to have cleaner error messages):

```
newtype Parser a = Parser { runParser
   :: String          -- input string
   → [(String, a)] }  -- possible values + leftover
```

It is naturally possible to run such a parser and try to extract a value from a successful parse. Opinions may vary on whether a run with leftover characters can be considered successful or not. We decide against it. This is not crucial as both styles can be mutually emulated by either providing an 'end of file' parser guaranteeing that only runs with no leftovers are successful or by extending a grammar with a dummy case consuming the rest of the input string.

```
parse :: Parser a → String → Maybe a
parse p s = case filter (null ∘ fst) (runParser p s) of
  [(_, a)] → Just a
  _        → Nothing
```

2.2 (Strongly-Typed) Combinators

The most basic parser is the one that accepts any character. It succeeds as long as the input string is non empty and returns one result: the tail of the string together with the character it just read.

```
anyChar :: Parser Char
anyChar = Parser $ λs → case s of
  []      → []
  (c : s) → [(s, c)]
```

However what makes parsers interesting is that they recognize structure. As such, they need to reject invalid inputs. The parser only accepting decimal digit is a bare bones example. It can be implemented in terms of *guard*, a higher order parser checking that the value returned by its argument abides by a predicate which can easily be implemented using functions from the standard library.

```
guard :: (a → Bool) → Parser a → Parser a
guard f p = Parser $ filter (f ∘ snd) ∘ runParser p

digit :: Parser Char
digit = guard (∈ "0123456789") anyChar
```

These two definitions are only somewhat satisfactory: the result of the *digit* parser is still *stringly-typed*. Instead of using a predicate to decide whether to keep the value, we can opt for a validation function of type $a \to Maybe\ b$ which returns a witness whenever the check succeeds. To define this alternative *guardM* we can once more rely on code already part of the standard library.

In our concrete example of recognizing a digit, we want to return an `Int` corresponding to it. Once more the standard library has just the right function to use together with *guardM*.

```
guardM :: (a → Maybe b) → Parser a → Parser b
guardM f p = Parser $ catMaybes ∘ fmap (traverse f)
                    ∘ runParser p

digit :: Parser Int
digit = guardM (readMaybe ∘ (:[])) anyChar
```

2.3 Expressivity: Structures, Higher Order Parsers and Fixpoints

We have seen how we can already rely on the standard library of the host language to seamlessly implement combinators. We can leverage even more of the existing codebase by noticing that the type constructor `Parser` is a `Functor`, an `Applicative` [10], a `Monad` and also an `Alternative`.

Our first example of a higher order parser was *guardM* which takes as arguments a validation function as well as another parser and produces a parser for the type of witnesses returned by the validation function.

The two parsers *some* and *many* turn a parser for elements into ones for non-empty and potentially empty lists of such elements respectively. They concisely showcase the power of mutual recursion, higher-order functions and the `Functor`, `Applicative`, and `Alternative` structure.

```
some :: Parser a → Parser [a]          many :: Parser a → Parser [a]
some p = (:) <$> p <*> many p          many p = some p <|> pure []
```

Remark: Non-Commutative The disjunction combinator is non-commutative. As such the definitions of *some* and *many* will try to produce the longest list as possible as opposed to a flipped version of many which would start by returning the empty list and slowly offer longer and longer matches.

3 The Issue with Haskell's Parser Types

The ability to parse recursive grammars by simply declaring them in a recursive manner is however dangerous: unlike type errors which are caught by the typechecker and partial covers in pattern matchings which are detected by the coverage checker, termination is not guaranteed here.

The problem already shows up in the definition of *some* which will only make progress if its argument actually uses up part of the input string. Otherwise it may loop. However this is not the typical hurdle programmers stumble upon: demanding a non empty list of nothing at all is after all rather silly. The issue manifests itself naturally whenever defining a left recursive grammar which leads us to introducing the prototypical such example: `Expr`, a minimal language of arithmetic expressions.

```
Expr ::= <Int> | <Expr> '+' <Expr>
```

The intuitive solution is to simply reproduce this definition by introducing an inductive type for `Expr` and then defining the parser as an alternative between a literal on one hand and a sub-expression, the character '+', and another sub-expression on the other.

```
data Expr = Lit Int | Add Expr Expr
```

```
expr :: Parser Expr
expr = Lit <$> int <|> Add <$> expr <* char '+' <*> expr
```

However this leads to an infinite loop. Indeed, the second alternative performs a recursive call to *expr* even though it hasn't consumed any character from the input string.

The typical solution to this problem is to introduce two 'tiers' of expressions: the *base* ones which can only be whole expressions if we consume an opening parenthesis first and the *expr'* ones which are left-associated chains of *base* expressions connected by '+'.

```
base :: Parser Expr
base = Lit <$> int <|> char '(' *> expr' <* char ')'

expr :: Parser Expr
expr = base <|> Add <$> base <* char '+' <*> expr'
```

This approach can be generalised when defining more complex languages by having even more tiers, one for each *fixity level*, see for instance **??**. An extended language of arithmetic expressions would for instance distinguish the level at which addition and substraction live from the one at which multiplication and division do.

Our issue with this solution is twofold. First, although we did eventually managed to build a parser that worked as expected, the compiler was unable to warn us and guide us towards this correct solution. Additionally, the blatant partiality of some of these definitions means that these combinators and these types are wholly unsuitable in a total setting. We could, of course use an escape hatch and implement our parsers in Haskell but that would both be unsafe and mean we would not be able to run them at typechecking time which we may want to do if we embed checked examples in our software's documentation, or use constant values (e.g. filepaths).

4 Indexed Sets and Course-of-Values Recursion

Our implementation of Total Parser Combinators is in Agda, a total dependently typed programming language and it will rely heavily on indexed sets. But the indices will not be playing any interesting role apart from enforcing totality. As a consequence, we introduce combinators to build indexed sets without having to mention the index explicitly. This ought to make the types more readable by focusing on the important components and hiding away the artefacts of the encoding.

The first kind of combinators corresponds to operations on sets which are lifted to indexed sets by silently propagating the index. We only show the ones we will use in this paper: the pointwise arrow and product types and the constant function.

```
_⟶_ : (I → Set) → (I → Set) → (I → Set)
(A ⟶ B) n = A n → B n
```

```
_⊗_ : (I → Set) → (I → Set) → (I → Set)
(A ⊗ B) n = A n × B n
```

```
κ : Set → (I → Set)
κ A n = A
```

Remark: Mixfix Operators In Agda underscores correspond to positions in which arguments are to be inserted. It may be a bit surprising to see infix notations for functions taking three arguments but they are only meant to be partially applied.

The second kind of combinator turns an indexed set into a set by universally quantifying over the index.

```
[_] : (I → Set) → Set
[ A ] = ∀ {n} → A n
```

Remark: Implicit Arguments We use curly braces so that the index we use is an *implicit* argument we will never have to write: Agda will fill it in for us by unification.

We can already see the benefits of using these aliases. For instance the fairly compact expression $[\ (\kappa\ P \otimes Q) \longrightarrow R\ ]$ corresponds to the more verbose type $\forall\ \{n\} \to (P \times Q\ n) \to R\ n$.

Last but not least, the □ type constructor takes a ℕ-indexed set and produces the set of valid recursive calls for a function defined by course-of-values recursion: □ $A\ n$ associates to each m strictly smaller than n a value of type $A\ m$. This construct, analogous to the later modality showing up in Guarded Type Theory [13], empowers the user to give precise types in a total language to programs commonly written in partial ones (see e.g. the definition of *fix* below).

```
record □_ (A : ℕ → Set) (n : ℕ) : Set where
  constructor mkBox
  field call : ∀ {m} → .(m < n) → A m
```

Remark: Record Wrapper Instead of defining □ as a function like the other combinators, we wrap the corresponding function space in a record type. This prevents normalisation from unfolding the combinator too eagerly and makes types more readable during interactive development.

Remark: Irrelevance The argument stating that m is strictly smaller than n is preceded by a dot. In Agda, it means that this value is irrelevant and can be erased by the compiler. In Coq, we would define the relation `_<_` in `Prop` to achieve the same.

This □ type combinator has some useful value combinators. The first thing we can notice is the fact that **□ is a functor**; that is to say that given a natural transformation from A to B, we can define a natural transformation from □ A to □ B.

```
map : [ A ⟶ B ] → [ □ A ⟶ □ B ]
call (map f A) m<n = f (call A m<n)
```

Remark: Copatterns The definition of map uses the □ field named call on the *left hand side*. This is a copattern [1], meaning that we explain how the definition is *observed* (via call) rather than *constructed* (via mkBox).

Because _<_ is defined in terms of _⩽_, ⩽-refl which is the proof that _⩽_ is reflexive is also a proof that any n is strictly smaller than 1 + n. We can use this fact to write the following extract function:

```
extract : [ □ A ] → [ A ]
extract a = call a ⩽-refl
```

Remark: Counit The careful reader will have noticed that this is not quite the *extract* we would expect from a comonad: for a counit, we would need a natural transformation between □ A and A i.e. a function of type [□ A ⟶ A]. We will not be able to define such a function: □ A 0 is isomorphic to the unit type so we would have to generate an A 0 out of thin air. The types A for which □ has a counit are interesting in their own right: they are inhabited at every single index as demonstrated by fix later on.

Even though we cannot have a counit, we are still able to define a comultiplication thanks to the fact that _<_ is transitive.

```
duplicate : [ □ A ⟶ □ □ A ]
call (call (duplicate A) m<n) p<m = call A (<-trans p<m m<n)
```

Remark: Identifiers in Agda Any space-free string which is not a reserved keyword is a valid identifier. As a consequence we can pick suggestive names such as $m{<}n$ for a proof that $m < n$ (notice the extra spaces around the infix operator (<)).

Exploring further the structure of the functor □, we can observe that just like it is not quite a comonad, it is not quite an applicative functor. Indeed we can only define *pure*, a natural transformation of type [A ⟶ □ A], for the types A that are downwards closed. Providing the user with *app* is however possible:

```
app : [ □ (A ⟶ B) ⟶ (□ A ⟶ □ B) ]
call (app F A) m<n = call F m<n (call A m<n)
```

Finally, we can reach what will serve as the backbone of our parser definitions: a safe, total fixpoint combinator. It differs from the traditional Y combinator in that all the recursive calls have to be guarded.

```
fix : ∀ A → [ □ A ⟶ A ] → [ A ]
```

If we were to unfold all the type-level combinators and record wrappers, the type of fix would correspond exactly to strong induction for the natural numbers. Hence its implementation also follows the one of strong induction: it is a combination of a call to extract and an auxiliary definition fix□ of type [□ A ⟶ A] → [□ A].

Remark: Generalisation A similar □ type constructor can be defined for any induction principle relying on an accessibility predicate. Which means that a library's types can be cleaned up by using these combinators in any situation where one had to give up structural induction for a more powerful alternative.

5 Parsing, Totally

As already highlighted in Section 3, *some* and *many* can yield diverging computations if the parser they are given as an argument succeeds on the empty string. To avoid any such issue, we adopt a radical solution: for a parser's run to be considered successful, it must have consumed some of its input.

This can be made formal with the Success record type: a Success of type A and size n is a value of type A together with the leftovers of the input string of size strictly smaller than n.

```
record Success (A : Set) (n : ℕ) : Set where
 constructor _^_,_
 field value     : A
       {size}    : ℕ
       .small    : size < n
       leftovers : Vec Char size
```

Remark: Implicit Field Just like the arguments to a function can be implicit, so can a record's fields. The user can then leave them out when building a value and they will be filled in by unification.

Coming back to Fritz Ruehr's rhyme, we can define our own Parser type: a parser for things up to size n is a function from strings of length m less than n to lists of Successes of size m.

```
record Parser (A : Set) (n : ℕ) : Set where
 constructor mkParser
 field runParser : ∀ {m} → .(m ⩽ n) → Vec Char m →
                   List (Success A m)
```

Now that we have a precise definition of Parsers, we can start building our library of combinators. Our first example anyChar can be defined by copattern-matching and then case analysis on the input string: if it is empty then the list of Successes is also empty, otherwise it contains exactly one element which corresponds to the head of the input string and its tail as leftovers.

```
anyChar : [ Parser Char ]
runParser anyChar _ s with s
... | []       = []
... | c :: cs = (c ^ ⩽-refl , cs) :: []
```

Unsurprisingly guardM is still a valid higher-order combinator: filtering out results which do not agree with a predicate is absolutely compatible with the consumption constraint we have drawn. To implement guardM we can once more reuse existing library functions such as gfilter which turns a List A into a List B provided a predicate $A \to$ Maybe B.

```
guardM : (A → Maybe B) → [ Parser A ⟶ Parser B ]
runParser (guardM p A) m⩽n s =
 gfilter (sequence ∘ Success.map p) (runParser A m⩽n s)
```

Demonstrating that Parser is a functor goes along the same lines: using List's and Success's maps. Similarly, we can prove that it is an Alternative: failing corresponds to returning the empty list no matter what whilst disjunction is implemented using concatenation.

```
_<$>_ : (A → B) → [ Parser A ⟶ Parser B ]
```

```
fail : [ Parser A ]                  _<|>_ : [ Parser A ⟶ Parser A ⟶ Parser A ]
```

So far the types we have ascribed to our combinators are, if we ignore the $\mathbb{N}$ indices, exactly the same as the ones one would find in any other parsec library. In none of the previous combinators do we run a second parser on the leftovers of a first one. All we do is either manipulate or combine the results of one or more parsers run in parallel, potentially discarding some of these results on the way.

However when we run a parser *after* some of the input has already been consumed, we could safely perform a *guarded* call. This being made explicit would be useful when using fix to define a parser for a recursive grammar. Luckily Parser is, by definition, a downwards-closed type. This means that we may use very precise types marking all the guarded positions with □; if the user doesn't need that extra power she can very easily bypass the □ annotations by using box:

```
box : [ Parser A ⟶ □ Parser A ]
```

The most basic example we can give of such an annotation is probably the definition of a conjunction combinator _<&>_ taking two parsers, running them sequentially and returning a pair of their results. The second parser is given the type □ Parser B instead of Parser B which we would expect to find in other parsec libraries.

```
_<&>_ : [ Parser A ⟶ □ Parser B ⟶ Parser (A × B) ]
```

We can immediately use all of these newly-defined combinators to give a safe, total definition of some which takes a parser for A and returns a parser for non-empty lists of As. It is defined as a fixpoint and proceeds as follows: it either combines a head and a non-empty tail using _∷⁺_ or returns a singleton list.

```
some : [ Parser A ] → [ Parser (List⁺ A) ]
some p = fix _ $ λ rec → uncurry _∷⁺_ <$> (p <&> rec)
                         <|> (_∷ []) <$> p
```

Remark: Inefficiency Unfortunately this definition is inefficient. Indeed, in the base case some p is going to run the parser p twice: once in the first branch before realising that *rec* fails and once again in the second branch. Compare this definition to the Haskell version (after inlining *many*) where p is run once and then its result is either combined with a list obtained by recursion or returned as a singleton:

```
some :: Parser a → Parser [a]
some p = (:) <$> p <*> (some p <|> pure [])
```

This inefficiency can be fixed by using a notion of conjunction _<&?>_ which allows the second parser to fail whilst still producing a successful run. The parser some p can then be defined as the conjunction of p and a potentially failing recursive call. The function cons (whose definition we omit here) either uses $\_::^{+}\_$ to put the head and the non-empty tail together or returns a singleton list if the recursive call failed.

```
_<&?>_ : [ Parser A ⟶ □ Parser B ⟶ Parser (A × Maybe B) ]
A <&?> B = A &?≫= const B


some : [ Parser A ] → [ Parser (List⁺ A) ]
some p = fix _ $ λ rec → cons <$> (p <&?> rec)
```

Remark: Non-Compositional The higher-order parser some expects a *fully* defined parser as an argument. This makes it impossible to use it as one of the building blocks of a larger, recursive parser. Ideally we would rather have a combinator of type [Parser A ⟶ Parser (List^{+} A)]. This will be addressed in the next subsection.

As the previous code snippet shows, _<&?>_ is defined in terms of a more fundamental notion _&?≫=_ which is a combinator analogous to a monad's *bind*. On top of running two parsers sequentially (with the second one being chosen based on the result obtained by running the first), it allows the second one to fail and returns both results.

```
_&?≫=_ : [ Parser A ⟶ (κ A ⟶ □ Parser B) ⟶ Parser (A × Maybe B) ]
```

This crucial definition makes it possible to port a lot of the Haskell definitions where one would use a parser which does not use any of its input. And this is possible without incurring any additional cost as the optimised version of some showed.

6 Left Chains

The pattern used in the solution presented in Section 3 can be abstracted with the notion of an (heterogeneous) left chain which takes a parser for a seed, one for a constructor, and one for and argument. The crucial thing is to make sure *not* to use the parser one is currently defining as the seed.

```
hchainl :: Parser a → Parser (a → b → a) → Parser b → Parser a
hchainl seed con arg = seed ≫= rest where
  rest :: a → Parser a
  rest a = do {f ← con; b ← arg; rest (f a b)} <|> return a
```

We naturally want to include a safe variant of this combinator in our library. However this definition relies on the ability to simply use *return* in case it's not possible to parse an additional constructor and argument and that is something we simply don't have access to. This forces us to find the essence of *rest*, the auxiliary definition used in *hchainl*: its first argument is not just a value, it is a Success upon which it builds until it can't anymore and simply returns.

We define schainl as the fixpoint of either rest (whose definition we omit here) or the Success that was passed as an input (the function $\_::^{r}$ appends a single element at the end of a list).

```
schainl : [ Success A ⟶ □ Parser (A → A) ⟶ List ∘ Success A ]
schainl = fix _ $ λ rec sA op → rest rec sA op ∷ʳ sA where

 rest : [ □ (Success A ⟶ □ Parser (A → A) ⟶ List ∘ Success A)
        ⟶ Success A ⟶ □ Parser (A → A) ⟶ List ∘ Success A ]
```

The type of rest may look intimidating but it has this shape because we build a list of Successes at the *same* index as the input Success and Parser which will make this combinator compositional (as opposed to *some* defined in Section 5). Ultimately all of this complexity only shows up in the implementation of our library: the end user can happily ignore these details.

From this definition we can derive iterate which takes a parser for a seed and a parser for a function and kickstarts a call to schainl on the result of the parser for the seed.

```
iterate : [ Parser A ⟶ □ Parser (A → A) ⟶ Parser A ]
```

Finally, hchainl can be implemented using iterate, the applicative structure of Parser and some of the properties of □.

```
hchainl : [ Parser A ⟶ □ Parser (A → B → A) ⟶ □ Parser B ⟶ Parser A ]
```

As we have mentioned when defining schainl, the combinator hchainl we have just implemented does not expect fully-defined parsers as arguments. As a consequence it can be used inside a fixpoint construction. Both the parser for the constructor and the one for its B argument are guarded whilst the one for the A seed is not. This means that trying to define a left-recursive grammar by immediately using a recursive substructure on the left is now a type error. But it still possible to have some on the right or after having consumed at least one character (typically an opening parenthesis, cf. the `Expr` example in Section 3).

7 Fully Worked-Out Example

From hchainl, one can derive chainl1 which is not heterogeneous and uses the same parser for the seed and the constructors' arguments. This combinator together with the idea of fixity mentioned in Section 3 is typically used to implement left-recursive grammars. Looking up the documentation of the `parsec` library on hackage [8] we can find a fine example: an extension of our early arithmetic language.

```
expr   = term `chainl1` addop
term   = factor `chainl1` mulop
factor = parens expr <|> integer

mulop = do { symbol "*"; return (*) }
        <|> do { symbol "/"; return (div) }

addop = do { symbol "+"; return (+) }
        <|> do { symbol "-"; return (-) }
```

One important thing to note here is that in the end we not only get a parser for the `expressions` but also each one of the intermediate categories `term` and `factor`. Luckily, our library lets us take fixpoints of any sized types we may fancy. As such, we can define a sized record of parsers for each one of the syntactic categories:

```
record Language (n : ℕ) : Set where
 field expr   : Parser Expr n
       term   : Parser Term n
       factor : Parser Factor n
```

Here, unlike the Haskell example, we decide to be painfully explicit about the syntactic categories we are considering: we mutually define three inductive types representing *left-associated* arithmetic expressions.

```
data Expr : Set where            data Term : Set where              data Factor : Set where
 Emb : Term → Expr                Emb : Factor → Term                 Emb : Expr → Factor
 Add : Expr → Term → Expr         Mul : Term → Factor → Term          Lit : ℕ → Factor
 Sub : Expr → Term → Expr         Div : Term → Factor → Term
```

The definition of the parser itself is then basically the same as the Haskell one. Contrary to a somewhat popular belief, working in a dependently-typed language does not force us to add any type annotation except for the top-level one.

```
language : [ Language ]
language = fix Language $ λ rec →
 let addop = Add <$ char '+' <|> Sub <$ char '-'
     mulop = Mul <$ char '*' <|> Div <$ char '/'
     factor = Emb <$> parens (map expr rec) <|> Lit <$> decimal
     term   = hchainl (Emb <$> factor) (box mulop) (box factor)
     expr   = hchainl (Emb <$> term)   (box addop) (box term)
 in record { expr = expr ; term = term ; factor = factor }
```

We can notice four minor changes with respect to the Haskell version. Firstly, the intermediate parsers need to be declared before being used which effectively reverses the order in which they are spelt out. Secondly, the recursive calls are now explicit: in the definition of `factor`, expr is mapped under the □ to project the recursive call to the Expr Parser out of Language. Thirdly, we use hchainl instead of chainl1 because breaking the grammar into three distinct categories leads us to parsing *h*eterogeneous left chains. Fourthly, we have to insert calls to box to lift Parsers into boxed ones whenever the added guarantee that the call will be guarded is of no use to us.

8 More Power: Switching to other Representations

We have effectively managed to take Haskell's successful approach to defining a Domain Specific Language of parser combinators and impose type constraints which make it safe to use in a total setting. All of which we have done whilst keeping the concision and expressivity of the original libraries. A natural next question would be the speed and efficiency of such a library.

Although we have been using a concrete type for Parser throughout this article, our library actually implements a more general one. It uses Agda's instance arguments throughout thus letting the user pick the representation they like best.

Firstly, there is nothing special about vectors of characters as an input type: any sized input off of which one can peel characters one at a time would do. Users may instead use Haskell's `Text` packaged together with an irrelevant proof that the given text has the right length and a binding for *uncons*. This should lead to a more efficient memory representation of the text being analysed.

Similarly, there is no reason to force the user to get back a *list* of successes: any alternative monad will do. This means in particular that a user may for instance instrument a parser with a logging ability to be able to return good error messages, have a (re)configurable grammar using a `Reader` transformer or use a `Maybe` type if they want to make explicit the fact that their grammar is unambiguous.

9 Related Work

This work is based on Hutton and Meijer's influential functional pearl [6] which builds on Walder's insight that exception handling and backtracking can be realised using a list of successes [14]. Similar Domain Specific Languages have been implemented in various functional languages such as Scala [11] or, perhaps more interestingly for us, Rust [4] where the added type-level information about ownership can help implement a guaranteed zero-copy parser.

9.1 Total Parser Combinators

When it comes to total programming languages, Danielsson's library [5] is to our knowledge the only attempt so far at defining a library of total parser combinators in a dependently-typed host language. He reifies recursive grammars as values of a mixed inductive-coinductive type and tracks at the type level whether a sub-grammar accepts the empty word and, as a consequence, whether one can meaningfully take its fixpoint.

The reified approach allows him to define a grammar's semantics in terms of bags of words and prove sound a variant of Brzozowski derivatives [3] as well as study the equational theory of parsers. The current implementation, based on the Brzozowski derivatives, is however of complexity at least exponential in the size of the input.

Our approach, although not as successful as Danielsson's when it comes to certification, is however more lightweight. Using only strong induction on the natural numbers, it is compatible with languages a lot less powerful than Agda. Indeed there is no need for good support for mixed induction and coinduction in the host language. And although we do rely on enforcing invariants at the type-level, one could mimic these in languages with even weaker type systems by defining an *abstract* □ and only providing the user with our set of combinators which is guaranteed to be safe.

9.2 Certified Parsing

Ambitious projects such as CompCert [9] providing the user with an ever more certified toolchain tend to bring to light the lack of proven-correct options for very practical concerns such as parsing. Jourdan, Pottier and Leroy's work [7] fills that gap by certifying the output of an untrusted parser generator for LR(1) grammars. This approach serves a different purpose than ours: parser combinators libraries are great for rapid prototyping and small, re-configurable parsers for non-critical applications.

Bernardy and Jansson have implemented in Agda a fully-certified generalisation of Valiant's algorithm [2] by deriving it from its specification. This algorithm gives the best asymptotic bounds on context-free grammar, that is the `Applicative` subset tackled by parser combinators.

10 Conclusion and Future Work

Starting from the definition of "parsers for things as functions from strings to lists of strings and things" common in Haskell, we have been able to (re)define versatile combinators. However the type system was completely unable to root out some badly-behaved programs, namely the ones taking the fixpoint of a grammar accepting the empty word or non well-founded left recursive grammars. Wanting to use a total programming language, this led us to a radical solution: rejecting all the parsers accepting the empty word. Luckily, it was still possible to recover a notion of "potentially failing" sub-parses via a *bind*-like combinator as well as defining combinators for left chains. Finally we saw that this yielded a perfectly safe and only barely more verbose set of total parser combinators.

In the process of describing our library we have introduced a set of type-level combinators for manipulating indexed types and defining values by strong induction. If we want to provide our users with the tools to modularly prove some of the properties of their grammars, we need to come up with proof combinators corresponding to the value ones. As far as we know this is still an open problem.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[2] Jean-Philippe Bernardy and Patrik Jansson. Certified context-free parsing: A formalisation of valiant's algorithm in agda. *arXiv preprint arXiv:1601.07724*, 2016.

[3] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[4] Geoffroy Couprie. Nom, a byte oriented, streaming, zero copy, parser combinators library in rust. In *Security and Privacy Workshops (SPW), 2015 IEEE*, pages 142–148. IEEE, 2015.

[5] Nils Anders Danielsson. Total parser combinators. In *ACM Sigplan Notices*, volume 45, pages 285–296. ACM, 2010.

[6] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of functional programming*, 8(4):437–444, 1998.

[7] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating lr (1) parsers. In *ESOP*, volume 7211, pages 397–416. Springer, 2012.

[8] Daan Leijen, Paolo Martini, and Antoine Latter. Parsec documentation. https://hackage.haskell.org/package/parsec-3.1.11/docs/Text-Parsec.html, 2017. Retrieved on 2017-10-30.

[9] Xavier Leroy et al. The compcert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[10] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

[11] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical report, 2008.

[12] Ulf Norell. Dependently typed programming in Agda. In *AFP Summer School*, pages 230–266. Springer, 2009.

[13] Andrea Vezzosi. Guarded recursive types in type theory. Licentiate thesis, Chalmers University of Technology, 2015.

[14] Philip Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.