# A predicate transformer semantics of parser combinators

Tim Baanen
Vrije Universiteit Amsterdam

Wouter Swierstra
Utrecht University

## 1 Introduction

There is a significant body of work on parsing using combinators in functional programming languages [others?; Hut92; SD96; Wad85],. Yet how can we ensure that these parsers are correct? There is notably less work that attempts to answer this question [Dan10; Fir16].

Reasoning about such parser combinators is not at all trivial; they use a variety of effects: state to store the string being parsed; non-determinism to handle backtracking; and general recursion to deal with recursive grammars. Proof techniques, such as equational reasoning, that are commonly used when reasoning about pure functional programs, are less suitable when verifying effectful programs.

In this paper, we explore a different approach, drawing inspiration from recent work on algebraic effects [BP15; WSH14; McB15]. We demonstrate how to reason about all parsers uniformly using predicate transformers [SB19]. We extend our previous work that uses predicate transformer semantics to reason about a single effect to handle the combinations of effects used by parsers. Our semantics is modular, allowing us to introduce concepts only when they are needed, without having to rework the previous definitions. In particular, our careful treatment of general recursion lets us separate the partial correctness of the combinators from their termination cleanly. Most existing proofs require combinators to guarantee that the string being parsed decreases, conflating termination and correctness.

In particular, this paper makes the following novel contributions:

- The non-recursive fragment of regular expressions can be correctly parsed using non-determinism (Section 3).
- By combining non-determinism with general recursion (Section 4), support for the Kleene star can be added without compromising our previous definitions (Section 5).
- Although the resulting parser is not guaranteed to terminate, we can define another implementation using Brzozowski derivatives (Section 6), introducing an extra effect to our combinations and its handler in the process.
- Finally, we show that the derivative-based implementation terminates and refines the original parser (Section 7).

The structure of our article is similar to a Functional Pearl by Harper [Har99], which also uses the parsing of regular languages as an example of principles of functional software development. Starting out with defining regular expressions as a data type and the language associated with each expression as an inductive relation, both use the relation to implement essentially the same *match* function, which does not terminate. In both, the partial correctness proof of *match* uses a specification expressed as a postcondition, based on the inductive relation representing the language of a given regular expression. Where we use nondeterminism to handle

To appear in EPTCS.

the concatenation operator, Harper uses a continuation-passing parser for control flow. Since the continuations take the unparsed remainder of the string, they correspond almost directly to the *Parser* effect of the following section. Another main difference between our implementation and Harper's is in the way the non-termination of *match* is resolved. Harper uses the derivative operator to rewrite the expression in a standard form which ensures that the *match* function terminates. We use the derivative operator to implement a different matcher *dmatch* which is easily proved to be terminating, then show that *match*, which we have already proven partially correct, is refined by *dmatch*. The final major difference is that Harper uses manual verification of the program and our work is formally computer-verified. Although our development takes more work, the correctness proofs give more certainty than the informal arguments made by Harper. In general, choosing between informal reasoning and formal verification will always be a trade-off between speed and accuracy.

All the programs and proofs in this paper are written in the dependently typed language Agda [Nor07].

## 2 Recap: algebraic effects and predicate transformers

Algebraic effects separate the syntax and semantics of effectful operations. In this paper, we will model the by taking the free monad over a given signature, describing certain operations. The type of such a signature is defined as follows:

```
record Sig : Set where
  constructor mkSig
  field
    C : Set
    R : C → Set
```

Here the type *C* contains the 'commands', or effectful operations that a given effect supports. For each command *c* : *C*, the type *R c* describes the possible responses. The structure on a signature is that of a container [AAG03]. For example, the following signature describes two operations: the non-deterministic choice between two values, *Choice*; and a failure operator, *Fail*.

```
data CNondet : Set where
  Choice : CNondet
  Fail   : CNondet
RNondet : CNondet → Set
RNondet Choice = Bool
RNondet Fail   = ⊥
Nondet = mkSig CNondet RNondet
```

We represent effectful programs that use a particular effect using the corresponding free monad:

```
data Free (e : Sig) (a : Set) : Set where
  Pure : a → Free e a
  Op : (c : C e) → (R e c → Free e a) → Free e a
```

This gives a monad, with the bind operator defined as follows:

$$\begin{aligned}
&\_{\gg\!=}\_ : \mathit{Free}\ e\ a \to (a \to \mathit{Free}\ e\ b) \to \mathit{Free}\ e\ b\\
&\mathit{Pure}\ x \gg\!= f\ = f\ x\\
&\mathit{Op}\ c\ k \gg\!= f\ = \mathit{Op}\ c\ (\lambda\ x \to\ k\ x \gg\!= f)
\end{aligned}$$

To facilitate programming with effects, we define the following smart constructors, sometimes referred to as generic effects in the literature [PP03]:

$$\begin{aligned}
&\mathit{fail} : \mathit{Free}\ \mathit{Nondet}\ a\\
&\mathit{fail} = \mathit{Op}\ \mathit{Fail}\ \lambda\ ()\\
&\mathit{choice} : \mathit{Free}\ \mathit{Nondet}\ a \to \mathit{Free}\ \mathit{Nondet}\ a \to \mathit{Free}\ \mathit{Nondet}\ a\\
&\mathit{choice}\ S_1\ S_2 = \mathit{Op}\ \mathit{Choice}\ \lambda\ b \to \text{if } b \text{ then } S_1 \text{ else } S_2
\end{aligned}$$

In this paper, we will assign semantics to effectful programs by mapping them to predicate transformers. Each semantics will be computed by a fold over the free monad, mapping some predicate $P : a \to \mathit{Set}$ to a predicate on the result of the free monad to a predicate of the entire computation of type $\mathit{Free}\ (\mathit{eff}\ C\ R)\ a \to \mathit{Set}$.

$$\begin{aligned}
&[\![\cdot]\!]_\cdot : ((c : C) \to (R\ c \to \mathit{Set}) \to \mathit{Set}) \to\\
&\quad \mathit{Free}\ (\mathit{mkSig}\ C\ R)\ a \to (a \to \mathit{Set}) \to \mathit{Set}\\
&[\![\mathit{Pure}\ x]\!]_{alg}\ P = P\ x\\
&[\![\mathit{Op}\ c\ k]\!]_{alg}\ P = \mathit{alg}\ c\ \lambda\ x \to [\![k\ x]\!]_{alg}\ P
\end{aligned}$$

The predicate transformer nature of these semantics becomes evident when we assume the type of responses $R$ does not depend on the command $c : C$. The type of $\mathit{alg} : (c : C) \to (R\ c \to \mathit{Set}) \to \mathit{Set}$ then becomes $C \to (R \to \mathit{Set}) \to \mathit{Set}$, which is isomorphic to $(R \to \mathit{Set}) \to (C \to \mathit{Set})$. Thus, $\mathit{alg}$ has the form of a predicate transformer from postconditions of type $R \to \mathit{Set}$ into preconditions of type $C \to \mathit{Set}$. Two considerations cause us to define the types $\mathit{alg} : (c : C) \to (R\ c \to \mathit{Set}) \to \mathit{Set}$, and analogously $[\![\cdot]\!]_{alg} : \mathit{Free}\ (\mathit{mkSig}\ C\ R)\ a \to (a \to \mathit{Set}) \to \mathit{Set}$. By having the command as first argument to $\mathit{alg}$, we allow $R$ to depend on $C$. Moreover, $[\![\cdot]\!]_{alg}$ computes semantics, so it should take a program $S : \mathit{Free}\ (\mathit{mkSig}\ C\ R)\ a$ as its argument and return the semantics of $S$, which is then of type $(a \to \mathit{Set}) \to \mathit{Set}$.

In the case of non-determinism, for example, we may want to require that a given predicate $P$ holds for all possible results that may be returned:

$$\begin{aligned}
&\mathit{ptAll} : (c : \mathit{CNondet}) \to (\mathit{RNondet}\ c \to \mathit{Set}) \to \mathit{Set}\\
&\mathit{ptAll}\ \mathit{Fail}\ P \quad\ = \top\\
&\mathit{ptAll}\ \mathit{Choice}\ P = P\ \mathit{True} \wedge P\ \mathit{False}
\end{aligned}$$

$$\begin{aligned}
&[\![\cdot]\!]_{\mathrm{all}} : \mathit{Free}\ \mathit{Nondet}\ a \to (a \to \mathit{Set}) \to \mathit{Set}\\
&[\![S]\!]_{\mathrm{all}} = [\![\cdot]\!]_\cdot\ \mathit{ptAll}\ S
\end{aligned}$$

Predicate transformers provide a single semantic domain to relate programs and specifications. Throughout this paper, we will consider specifications consisting of a pre- and postcondition:

```
module Spec where
  record Spec (a : Set) : Set where
```

```
    constructor [_,_]
    field
      pre : Set
      post : a → Set
```

Inspired by work on the refinement calculus, we can assign a predicate transformer semantics to specifications as follows:

$$\llbracket \cdot,\cdot \rrbracket_{\text{spec}} : \mathit{Spec}\ a \to (a \to \mathit{Set}) \to \mathit{Set}$$
$$\llbracket \mathit{pre}, \mathit{post} \rrbracket_{\text{spec}}\ P = \mathit{pre} \wedge (\forall\ o \to \mathit{post}\ o \to P\ o)$$

This computes the 'weakest precondition' necessary for a specification to imply that the desired postcondition $P$ holds. In particular, the precondition *pre* should hold and any possible result satisfying the postcondition *post* should imply the postcondition $P$.

Finally, we use the refinement relation to compare programs and specifications:

$$\_\sqsubseteq\_ : (pt_1\ pt_2 : (a \to \mathit{Set}) \to \mathit{Set}) \to \mathit{Set}$$
$$pt_1 \sqsubseteq pt_2 = \forall\ P \to pt_1\ P \to pt_2\ P$$

Together with the predicate transformer semantics we have defined above, this refinement relation can be used to relate programs to their specifications. The refinement relation is both transitive and reflexive.

## 3 Regular languages without recursion

To illustrate how to reason about non-deterministic code, we begin by defining a regular expression matcher. Initially, we will restrict ourselves to non-recursive regular expressions; we will add recursion in the next section.

We begin by defining the *Regex* datatype for regular expressions as follows: An element of this type represents the syntax of a regular

```
  data Regex : Set where
    Empty     : Regex
    Epsilon   : Regex
    Singleton : Char → Regex
    _|_       : Regex → Regex → Regex
    _·_       : Regex → Regex → Regex
    _⋆        : Regex → Regex
```

Note that the *Empty* regular expression corresponds to the empty language, while the *Epsilon* expression only matches the empty string. Furthermore, our language for regular expressions is closed under choice (_|_), concatenation (_·_) and linear repetition denoted by the Kleene star (_⋆).

What should our regular expression matcher return? A Boolean value is not particularly informative; yet we also choose not to provide an intrinsically correct definition, instead performing extrinsic verification using our predicate transformer semantics. The Tree data type below, captures a potential parse tree associated with a given regular expression:

```
Tree : Regex → Set
Tree Empty         = ⊥
Tree Epsilon       = ⊤
Tree (Singleton _) = Char
Tree (l | r)       = Either (Tree l) (Tree r)
Tree (l · r)       = Pair (Tree l) (Tree r)
Tree (r ⋆)         = List (Tree r)
```

In the remainder of this section, we will develop a regular expression matcher with the following type:

```
match : (r : Regex) (xs : String) → Free Nondet (Tree r)
```

Before we do so, however, we will complete our specification. Although the type above guarantees that we return a parse tree matching the regular expression *r*, there is no relation between the tree and the input string. To capture this relation, we define the following *Match* data type. A value of type *Match r xs t* states that the string *xs* is in the language given by the regular expression *r* as witnessed by the parse tree *t*:

```
data Match : (r : Regex) → String → Tree r → Set where
  Epsilon    : Match Epsilon Nil tt
  Singleton  : Match (Singleton x) (x ∷ Nil) x
  OrLeft     : Match l xs x → Match (l | r) xs (Inl x)
  OrRight    : Match r xs x → Match (l | r) xs (Inr x)
  Concat     : Match l ys y → Match r zs z →
               Match (l · r) (ys ++ zs) (y , z)
  StarNil    : Match (r ⋆) Nil Nil
  StarConcat : Match (r · (r ⋆)) xs (y , ys) → Match (r ⋆) xs (y ∷ ys)
```

Note that there is no constructor for *Match Empty xs ms* for any *xs* or *ms*, as there is no way to match the *Empty* language with a string *xs*. Similarly, the only constructor for *Match Epsilon xs ms* is where *xs* is the empty string *Nil*. There are two constructors that produce a *Match* for a regular expression of the form $l \mid r$, corresponding to the choice of matching either $l$ or $r$.

The cases for concatenation and iteration are more interesting. Crucially the *Concat* constructor constructs a match on the concatenation of the strings *xs* and *zs* – although there may be many possible ways to decompose a string into two substrings. Finally, the two constructors for the Kleene star, $r$  match zero (*StarNil*) or many (*StarConcat*) repetitions of $r$.

We will now turn our attention to the *match* function. The complete definition, by induction on the argument regular expression, can be found in Figure 1. Most of the cases are straightforward—the most difficult case is that for concatenation, where we non-deterministically consider all possible splittings of the input string *xs* into a pair of strings *ys* and *zs*. The *allSplits* function, defined below, computes all possible splittings:

```
allSplits : (xs : List a) → Free Nondet (List a × List a)
allSplits Nil = Pure (Nil , Nil)
allSplits (x ∷ xs) = choice
  (Pure (Nil , (x ∷ xs)))
  (allSplits xs >>= λ {(ys , zs) → Pure ((x ∷ ys) , zs)})
```

```
match : (r : Regex) (xs : String) → Free Nondet (Tree r)
match Empty          xs              = fail
match Epsilon        Nil             = Pure tt
match Epsilon        (_ :: _)        = fail
match (Singleton c)  Nil             = fail
match (Singleton c)  (x :: Nil)      with c ≟ x
match (Singleton c)  (.c :: Nil)     | yes refl = Pure c
match (Singleton c)  (x :: Nil)      | no ¬p    = fail
match (Singleton c)  (_ :: _ :: _)   = fail
match (l | r)        xs              = choice (Inl ⟨$⟩ match l xs) (Inr ⟨$⟩ match r xs)
match (l · r)        xs              = do (ys , zs) ← allSplits xs
                                          y ← match l ys
                                          z ← match r zs
                                          Pure (y , z)
match (r ⋆) xs                       = fail
```

Figure 1: The definition of the *match* function

Finally, we cannot yet handle the case for the Kleene star. We could attempt to mimic the case for concatenation, attempting to match $r \cdot (r\ )$. This definition, however, is rejected by Agda as it is not structurally recursive. For now, however, we choose to simply fail on all such regular expressions.

Still, we can prove that the *match* function behaves correctly on all regular expressions that do not contain iteration. The *hasNo∗* predicate holds of all such iteration-free regular expressions:

$$\mathit{hasNo}{*} : \mathit{Regex} \to \mathit{Set}$$

To verify our matcher is correct, we need to prove that it satisfies the specification consisting of the following pre- and postcondition:

```
pre : (r : Regex) (xs : String) → Set
pre r xs = hasNo∗ r
post : (r : Regex) (xs : String) → Tree r → Set
post = Match
```

The main correctness result can now be formulated as follows:

$$\mathit{matchSound} : \forall\ r\ xs \to [\![(\mathit{pre}\ r\ xs), (\mathit{post}\ r\ xs)]\!]_{\text{spec}} \sqsubseteq [\![\mathit{match}\ r\ xs]\!]_{\text{all}}$$

This lemma guarantees that all the parse trees computed by the *match* function satisfy the *Match* relation, provided the input regular expression does not contain iteration. Although we have omitted the proof, we will sketch the key lemmas and definitions that are necessary to complete it.

First of all, we quickly run into problems as soon as we need to reason about programs composed using the monadic bind operator. In particular, when verifying the case for $l \cdot r$, we would like to use our induction hypotheses on two recursive calls. To do, we prove the following lemma that allows us to replace the semantics of a composite program built using the monadic bind operation with the composition of the underlying predicate transformers:

$$\begin{aligned}&\mathit{consequence} : \forall\, \mathit{pt}\ (\mathit{mx} : \mathit{Free\ es\ a})\ (f : a \to \mathit{Free\ es\ b}) \to\\&\quad [\![\mathit{mx}]\!]_{\mathit{pt}}\ (\lambda\, x \to [\![f\ x]\!]_{\mathit{pt}}\ P) \equiv [\![\mathit{mx} \gg\!= f]\!]_{\mathit{pt}}\ P\end{aligned}$$

Substituting along this equality gives us the lemmas we need to deal with the $\_\gg\!=\_$ operator:

$$\begin{aligned}&\mathit{wpToBind} : (\mathit{mx} : \mathit{Free\ es\ a})\ (f : a \to \mathit{Free\ es\ b}) \to\\&\quad [\![\mathit{mx}]\!]_{\mathit{pt}}\ (\lambda\, x \to [\![f\ x]\!]_{\mathit{pt}}\ P) \to [\![\mathit{mx} \gg\!= f]\!]_{\mathit{pt}}\ P\\&\mathit{wpFromBind} : (\mathit{mx} : \mathit{Free\ es\ a})\ (f : a \to \mathit{Free\ es\ b}) \to\\&\quad [\![\mathit{mx} \gg\!= f]\!]_{\mathit{pt}}\ P \to [\![\mathit{mx}]\!]_{\mathit{pt}}\ (\lambda\, x \to [\![f\ x]\!]_{\mathit{pt}}\ P)\end{aligned}$$

The correctness proof for *match* closely matches the structure of *match* (and by extension *allSplits*). It uses the same recursion on *Regex* as in the definition of *match*. Since we make use of *allSplits* in the definition, we first give its correctness proof.

$$\begin{aligned}&\mathit{allSplitsPost} : \mathit{String} \to \mathit{String} \times \mathit{String} \to \mathit{Set}\\&\mathit{allSplitsPost}\ \mathit{xs}\ (\mathit{ys}\ , \mathit{zs}) = \mathit{xs} \equiv \mathit{ys} +\!\!+\ \mathit{zs}\\&\mathit{allSplitsSound} : \forall\, \mathit{xs} \to [\![\top, (\mathit{allSplitsPost}\ \mathit{xs})]\!]_{\mathrm{spec}} \sqsubseteq [\![\mathit{allSplits}\ \mathit{xs}]\!]_{\mathrm{all}}\end{aligned}$$

We refer to the accompanying code for the complete details of these proofs.

## 4 General recursion and non-determinism

The matcher we have defined in the previous section is incomplete, since it fails to handle regular expressions that use the Kleene star. The fundamental issue is that the Kleene star allows for arbitrarily many matches in certain cases, that in turn, leads to problems with Agda's termination checker. For example, matching *Epsilon* ⋆ with the empty string `""` may unfold the Kleene star infinitely often without ever terminating. As a result, we cannot implement *match* for the Kleene star using recursion directly.

Instead, we will deal with this (possibly unbounded) recursion by introducing a new effect. We will represent a recursively defined (dependent) function of type $(i : I) \to O\ i$ as an element of the type $(i : I) \to \mathit{Free}\ (\mathit{Rec}\ I\ O)\ (O\ i)$. Here $\mathit{Rec}\ I\ O$ is a synonym of the the signature type we saw previously [McB15]:

$$\begin{aligned}&\mathit{Rec} : (I : \mathit{Set})\ (O : I \to \mathit{Set}) \to \mathit{Sig}\\&\mathit{Rec}\ I\ O = \mathit{mkSig}\ I\ O\end{aligned}$$

Intuitively, you may want to think of values of type $(i : I) \to \mathit{Free}\ (\mathit{Rec}\ I\ O)\ (O\ i)$ as computing a (finite) call graph for every input $i : I$. Instead of recursing directly, the 'effects' that this signature support require an input $i : I$—corresponding to the argument of the recursive call; the continuation abstracts over a value of type $O\ i$, corresponding to the result of a recursive call. Note that the functions defined in this style are not recursive; instead we will need to write handlers to unfold the function definition or prove termination separately.

We cannot, however, define a *match* function of the form $\mathit{Free}\ (\mathit{Rec}\ \_\ \_)$ directly, as our previous definition also used non-determinism. To account for both non-determinism and unbounded recursion, we need a way to combine effects. Fortunately, free monads are known to be closed under coproducts; there is a substantial body of work that exploits this to (syntactically) compose separate effects [WSH14; Swi08].

Rather than restrict ourselves to the binary composition using coproducts, we modify the *Free* monad to take a list of signatures as its argument, taking the coproduct of the elements of the list as its signature functor. The *Pure* constructor remains as unchanged; while the *Op* constructor additionally takes an index into the list to specify the effect that is invoked.

```
data Free (es : List Sig) (a : Set) : Set where
  Pure : a → Free es a
  Op : (i : e ∈ es) (c : C e) (k : R e c → Free es a) → Free es a
```

By using a list of effects instead of allowing arbitrary disjoint unions, we have effectively chosen that the disjoint unions canonically associate to the right. We choose to use the same names and (almost) the same syntax for this new definition of *Free*, since all the definitions that we have seen previously can be readily adapted to work with this data type instead.

Most of this bookkeeping involved with different effects can be inferred using Agda's instance arguments. Instance arguments, marked using the double curly braces {{ }}, are automatically filled in by Agda, provided a unique value of the required type can be found. For example, we can define the generic effects that we saw previously as follows:

```
fail : {{iND : Nondet ∈ es}} → Free es a
fail {{iND}} = Op iND Fail λ ()
choice : {{iND : Nondet ∈ es}} → Free es a → Free es a → Free es a
choice {{iND}} S₁ S₂ = Op iND Choice λ b → if b then S₁ else S₂
call : {{iRec : Rec I O ∈ es}} → (i : I) → Free es (O i)
call {{iRec}} i = Op iRec i Pure
```

These now operate over any free monad with effects given by *es*, provided we can show that the list *es* contains the *NonDet* and *Rec* effects respectively. For convenience of notation, we introduce the $\_ \overset{es}{\looparrowright} \_$ notation for general recursion, i.e. Kleisli arrows into *Free* (*Rec* _ _ :: *es*).

$$
\begin{aligned}
&\_\overset{\_}{\looparrowright}\_ : (C : Set)\ (es : List\ Sig)\ (R : C \to Set) \to Set\\
&C \overset{es}{\looparrowright} R = (c : C) \to Free\ (mkSig\ C\ R :: es)\ (R\ c)
\end{aligned}
$$

With the syntax for combinations of effects defined, let us turn to semantics. Since the weakest precondition predicate transformer for a single effect is given as a fold over the effect's signature, the semantics for a combination of effects can be given by a list of such semantics.

```
record PT (e : Sig) : Set where
  constructor mkPT
  field
    pt   : (c : C e) → (R e c → Set) → Set
    mono : ∀ c P P′ → P ⊆ P′ → pt c P → pt c P′
data PTs : List Sig → Set where
  Nil    : PTs Nil
  _::_ : PT e → PTs es → PTs (e :: es)
```

The record type *PT* not only contains a predicate transformer *pt*, but also a proof that this predicate transformer is monotone. Several lemmas throughout this paper, such as the terminates-fmap lemma below, rely on the monotonicity of the underlying predicate transformers.

Given a such a list of predicate transformers, defining the semantics of an effectful program is a straightforward generalization of the previously defined semantics. The *Pure* case is identical, and in the *Op* case we can apply the predicate transformer returned by the *lookupPT* helper function.

$$\begin{aligned}
&\mathit{lookupPT} : (\mathit{pts} : \mathit{PTs}\ \mathit{es})\ (i : \mathit{mkSig}\ C\ R \in \mathit{es}) \to (c : C) \to (R\ c \to \mathit{Set}) \to \mathit{Set}\\
&\mathit{lookupPT}\ (\mathit{pt} :: \mathit{pts})\ {\in}\mathrm{Head} \quad = \mathit{PT}.\mathit{pt}\ \mathit{pt}\\
&\mathit{lookupPT}\ (\mathit{pt} :: \mathit{pts})\ ({\in}\mathrm{Tail}\ i) = \mathit{lookupPT}\ \mathit{pts}\ i
\end{aligned}$$

This results in the following definition of the semantics for combinations of effects.

$$\begin{aligned}
&[\![\cdot]\!]_{\cdot} : (\mathit{pts} : \mathit{PTs}\ \mathit{es}) \to \mathit{Free}\ \mathit{es}\ a \to (a \to \mathit{Set}) \to \mathit{Set}\\
&[\![\mathit{Pure}\ x]\!]_{\mathit{pts}}\ \ P = P\ x\\
&[\![\mathit{Op}\ i\ c\ k]\!]_{\mathit{pts}}\ P = \mathit{lookupPT}\ \mathit{pts}\ i\ c\ \lambda\ x \to [\![k\ x]\!]_{\mathit{pts}}\ P
\end{aligned}$$

The effects that we will use for our *match* function consist of a combination of nondeterminism and general recursion. Although we can reuse the *ptAll* semantics of nondeterminism, we have not yet given the semantics for recursion. However, it is not as easy to give a predicate transformer semantics for recursion in general, since the intended semantics of a recursive call depend on the function that is being defined. Instead, to give semantics to a recursive function, we assume that we have been provided with a relation of the type $(i : I) \to O\ i \to \mathit{Set}$, reminiscent of a loop invariant in an imperative program. The semantics then establishes whether or not the recursive function adheres to this invariant or not:

$$\begin{aligned}
&\mathit{ptRec} : ((i : I) \to O\ i \to \mathit{Set}) \to \mathit{PT}\ (\mathit{Rec}\ I\ O)\\
&\mathit{PT}.\mathit{pt}\qquad (\mathit{ptRec}\ R)\ i\ P \qquad\qquad\quad\ = \forall\ o \to R\ i\ o \to P\ o\\
&\mathit{PT}.\mathit{mono}\ (\mathit{ptRec}\ R)\ c\ P\ P'\ \mathit{imp}\ \mathit{asm}\ o\ h = \mathit{imp}\ \_\ (\mathit{asm}\ \_\ h)
\end{aligned}$$

As we shall see shortly, when revisiting the *match* function, the *Match* relation defined previously will fulfill the role of this 'invariant.'

## 5 Recursively parsing every regular expression

To deal with the Kleene star, we rewrite *match* as a generally recursive function using a combination of effects. Since *match* makes use of *allSplits*, we also rewrite that function to use a combination of effects. The types become:

$$\begin{aligned}
&\mathit{allSplits} : \{\!\{\ \mathit{iND} : \mathit{Nondet} \in \mathit{es}\ \}\!\} \to \mathit{List}\ a \to \mathit{Free}\ \mathit{es}\ (\mathit{List}\ a \times \mathit{List}\ a)\\
&\mathit{match} : \{\!\{\ \mathit{iND} : \mathit{Nondet} \in \mathit{es}\ \}\!\} \to \mathit{Regex} \times \mathit{String} \overset{\mathit{es}}{\looparrowright} \mathrm{Tree} \circ \mathit{Pair}.\mathit{fst}
\end{aligned}$$

Since the index argument to the smart constructor is inferred by Agda, the only change in the definition of *match* and *allSplits* will be that *match* now does have a meaningful branch for the Kleene star case:

$$\begin{array}{l}
\textit{match}\ ((r\,\star)\ ,\textit{Nil})\ =\ \textit{Pure Nil}\\
\textit{match}\ ((r\,\star)\ ,\textit{xs@}\ (\_\ ::\ \_))\ =\ \mathrm{do}\\
\quad (y\ ,\textit{ys})\ \leftarrow\ \textit{call}\ ((r\ \cdot\ (r\,\star))\ ,\textit{xs})\\
\quad \textit{Pure}\ (y\ ::\ \textit{ys})
\end{array}$$

The effects we need to use for running *match* are a combination of nondeterminism and general recursion. As discussed, we first need to give the specification for *match* before we can verify a program that performs a recursive *call* to *match*.

$$\begin{array}{l}
\textit{matchSpec}\ :\ (r,\textit{xs}\ :\ \textit{Pair Regex String})\ \to\ \mathrm{Tree}\ (\textit{Pair.fst}\ r,\textit{xs})\ \to\ \textit{Set}\\
\textit{matchSpec}\ (r\ ,\textit{xs})\ \textit{ms}\ =\ \textit{Match}\ r\ \textit{xs}\ \textit{ms}\\
[\![\cdot]\!]_{\mathrm{match}}\ :\ \textit{Free}\ (\textit{Rec}\ (\textit{Pair Regex String})\ (\mathrm{Tree}\ \circ\ \textit{Pair.fst})\ ::\ \textit{Nondet}\ ::\ \textit{Nil})\ a\ \to\\
\quad (a\ \to\ \textit{Set})\ \to\ \textit{Set}\\
[\![S]\!]_{\mathrm{match}}\ =\ [\![S]\!]_{\textit{ptRec matchSpec} :: \textit{ptAll} :: \textit{Nil}}
\end{array}$$

We can reuse exactly our proof that *allSplits* is correct, since we use the same semantics for the non-determinism used in the definition of *allSplits*. Similarly, the partial correctness proof of *match* will be the same on all cases except the Kleene star. Now we are able to prove correctness of *match* on a Kleene star.

$$\begin{array}{llll}
\textit{matchSound}\ ((r\,\star)\ ,\textit{Nil}) & P\ (\textit{preH}\ ,\textit{postH}) & = & \textit{postH}\ \_\ \textit{StarNil}\\
\textit{matchSound}\ ((r\,\star)\ ,(x\ ::\ \textit{xs})) & P\ (\textit{preH}\ ,\textit{postH})\ o\ H & = & \textit{postH}\ \_\ (\textit{StarConcat}\ H)
\end{array}$$

At this point, we have defined a matcher for regular languages and formally proven that when it succeeds in recognizing a given string, this string is indeed in the language generated by the argument regular expression. However, the *match* function does not necessarily terminate: if $r$ is a regular expression that accepts the empty string, then calling *match* on $r\,\star$ and a string *xs* will diverge. In the next section, we will write a new parser that is guaranteed to terminate and show that this parser refines the *match* function defined above.

## 6 Derivatives and handlers

Since recursion on the structure of a regular expression does not guarantee termination of the parser, we can instead perform recursion on the string to be parsed. To do this, we will use the Brzozowski derivative [Brz64] of regular languages:

Definition 1 The Brzozowski derivative of a formal language $L$ with respect to a character $x$ consists of all strings *xs* such that $x\ ::\ \textit{xs}\ \in\ L$.

Crucially, if $L$ is regular, so are all its derivatives. Thus, let $r$ be a regular expression, and $d\,r\,/d\,x$ an expression for the derivative with respect to $x$, then $r$ matches a string $x\ ::\ \textit{xs}$ if and only if $d\,r\,/d\,x$ matches *xs*. This suggests the following implementation of matching an expression $r$ with a string *xs*: if *xs* is empty, check whether $r$ matches the empty string; otherwise remove the head $x$ of the string and try to match $d\,r\,/d\,x$.

The first step in implementing a parser using the Brzozowski derivative is to compute the derivative for a given regular expression. Following Brzozowski [Brz64], we use a helper function $\varepsilon$? that decides whether an expression matches the empty string.

$$\varepsilon? : (r : \mathit{Regex}) \to \mathit{Dec}\ (\textstyle\sum\ (\mathrm{Tree}\ r)\ (\mathit{Match}\ r\ \mathit{Nil}))$$

The definition of $\varepsilon?$ is given by structural recursion on the regular expression, just as the derivative operator is:

```
d_/d_ : Regex → Char → Regex
d Empty        /d c = Empty
d Epsilon      /d c = Empty
d Singleton x /d c with c ≟ x
...                  | yes p  = Epsilon
...                  | no ¬p  = Empty
d l · r        /d c with ε? l
...                  | yes p  = ((d l /d c) · r) | (d r /d c)
...                  | no ¬p  = (d l /d c) · r
d l | r        /d c = (d l /d c) | (d r /d c)
d r ⋆          /d c = (d r /d c) · (r ⋆)
```

To use the derivative of $r$ to compute a parse tree for $r$, we need to be able to convert a tree for $d\ r\ /d\ x$ to a tree for $r$. As this function 'inverts' the result of derivation, we name it *integralTree*:

$$\mathit{integralTree} : (r : \mathit{Regex}) \to \mathrm{Tree}\ (d\ r\ /d\ x) \to \mathrm{Tree}\ r$$

Its definition closely follows the pattern matching performed in the definition of *d_/d_*.

The description of a derivative-based matcher is stateful: a step is done by removing a character from the input string. This state can be encapsulated in a new effect *Parser*. The *Parser* effect has one command *Symbol*. Calling *Symbol* will return the head of the unparsed remainder (advancing the string by one character) or *nothing* if the string has been totally consumed.

```
data CParser : Set where
  Symbol : CParser
RParser : CParser → Set
RParser Symbol = Maybe Char
Parser = mkSig CParser RParser
symbol : {{ iP : Parser ∈ es }} → Free es (Maybe Char)
symbol {{ iP }} = Op iP Symbol Pure
```

The code for the parser, *dmatch*, is now only a few lines long. When the input string is empty, we check that the expression matches the empty string; for a non-empty string we use the derivative to match the first character and recurse:

```
dmatch : {{ iP : Parser ∈ es }} {{ iND : Nondet ∈ es }} → Regex ↬^es Tree
dmatch r = symbol >>= maybe
  (λ x → integralTree r ⟨$⟩ call {{ ∈Head }} (d r /d x))
  (if ε? r then (λ p → Pure (Sigma.fst p)) else fail)
```

Here, *maybe f y* takes a *Maybe* value and applies $f$ to the value in *just*, or returns $y$ if it is *nothing*.

Adding the new effect *Parser* to our repertoire also requires specifying its semantics. We gave the effects *Nondet* and *Rec* predicate transformer semantics in the form of a *PT* record. After introducing the *Parser* effect, the pre- and postcondition become more complicated: not only do they reference the 'pure' values passed to and returned (here of type $r : Regex$ and $Tree\ r$ respectively), there is also the current state, containing a *String*, to keep track of. With these augmented predicates, the predicate transformer semantics for the *Parser* effect can be given as:

$$
\begin{aligned}
&ptParser : (c : CParser) \to (RParser\ c \to String \to Set) \to String \to Set\\
&ptParser\ Symbol\ P\ Nil \qquad\quad = P\ nothing\ Nil\\
&ptParser\ Symbol\ P\ (x :: xs) = P\ (just\ x)\ xs
\end{aligned}
$$

In this article, we want to demonstrate the modularity of predicate transformer semantics. Thus, we do not use *ptParser* as semantics for *Parser* in the remainder, so we can illustrate how the semantics mesh well with other forms of semantics. We give denotational semantics, in the form of an effect handler for *Parser*:

$$
\begin{aligned}
&hParser : \{\{ iND : Nondet \in es \}\} \to (c : CParser) \to String \to Free\ es\ (RParser\ c \times String)\\
&hParser\ Symbol\ Nil \qquad\ = Pure\ (nothing\ , Nil)\\
&hParser\ Symbol\ (x :: xs) = Pure\ (just\ x \quad , xs)
\end{aligned}
$$

The function *handleRec* folds a given handler over a recursive definition, allowing us to handle the *Parser* effect in *dmatch*.

$$
\begin{aligned}
&handleRec : ((c : C) \to s \to Free\ es\ (R\ c \times s)) \to a \overset{mkSig\ C\ R :: es}{\rightarrowtail} b \to a \times s \overset{es}{\rightarrowtail} b \circ Pair.fst\\
&dmatch' : \{\{ iND : Nondet \in es \}\} \to Regex \times String \overset{es}{\rightarrowtail} \mathrm{Tree} \circ Pair.fst\\
&dmatch' = handleRec\ hParser\ (dmatch)
\end{aligned}
$$

Note that *dmatch′* has exactly the type of the previously defined *match*, allowing us to more easily compare the two matchers.

## 7 Proving total correctness

Since *dmatch* always consumes a character before recursing, the number of recursive calls is bounded by the length of the input string As a result, we 'handle' the recursive effect by unfolding the definition a bounded number of times. In the remainder of this section, we will make this argument precise and relate the *dmatch* function above to the *match* function defined previously.

To ensure the termination of a recursive computation, we define the following predicate, terminates-in. Given any recursive computation $f : C \overset{es}{\rightarrowtail} R$, we check whether the computation requires no more than a fixed number of steps to terminate:

$$
\begin{aligned}
&\text{terminates-in} : (pts : PTs\ es)\\
&\quad (f : C \overset{es}{\rightarrowtail} R)\ (S : Free\ (mkSig\ C\ R :: es)\ a) \to \mathbb{N} \to Set\\
&\text{terminates-in}\ pts\ f\ (Pure\ x) \qquad\qquad n \qquad\quad = \top\\
&\text{terminates-in}\ pts\ f\ (Op \in\mathrm{Head}\ c\ k)\ Zero \qquad = \bot\\
&\text{terminates-in}\ pts\ f\ (Op \in\mathrm{Head}\ c\ k)\ (Succ\ n) = \text{terminates-in}\ pts\ f\ (f\ c \gg\!= k)\ n\\
&\text{terminates-in}\ pts\ f\ (Op\ (\in\mathrm{Tail}\ i)\ c\ k)\ n \qquad =\\
&\quad lookupPT\ pts\ i\ c\ (\lambda\ x \to \text{terminates-in}\ pts\ f\ (k\ x)\ n)
\end{aligned}
$$

Since *dmatch* always consumes a character before going in recursion, we can bound the number of recursive calls with the length of the input string. We formalize this argument in the lemma *dmatchTerminates*. Note that *dmatch′* is defined using the *hParser* handler, showing that we can mix denotational and predicate transformer semantics. The proof goes by induction on this string. Unfolding the recursive *call* gives *integralTree r* ⟨\$⟩ *dmatch′* (*d r* /*d x* , *xs*), which we rewrite using the associativity monad law in a lemma called terminates-fmap.

```
dmatchTerminates : (r : Regex) (xs : String) →
  terminates-in (ptAll :: Nil) (dmatch′) (dmatch′ (r , xs)) (length xs)
dmatchTerminates r Nil with ε? r
dmatchTerminates r Nil | yes p  = tt
dmatchTerminates r Nil | no ¬p = tt
dmatchTerminates r (x :: xs) = terminates-fmap (length xs)
  (dmatch′ ((d r /d x) , xs))
  (dmatchTerminates (d r /d x) xs)
```

To show partial correctness of *dmatch*, we will show that *dmatch* is a refinement of *match*. By the transitivity of the refinement relation, we can conclude that it also satisfies the specification given by our original *Match* relation. The first step is to show that the derivative operator is correct, i.e. *d r* /*d x* matches those strings *xs* such that *r* matches *x* :: *xs*.

```
derivativeCorrect : ∀ r → Match (d r /d x) xs y → Match r (x :: xs) (integralTree r y)
```

The proof is straightforward by induction on the derivation of type *Match* (*d r* /*d x*) *xs y*.

Using the preceding lemmas, we can prove the partial correctness of *dmatch* by showing it refines *match*:

$$dmatchSound : \forall\ r\ xs \rightarrow [\![ match\ (r , xs) ]\!]_{\mathrm{match}} \sqsubseteq [\![ dmatch'\ (r , xs) ]\!]_{\mathrm{match}}$$

Since we need to perform the case distinctions of *match* and of *dmatch*, the proof is longer than that of *matchSoundness*. Despite the length, most of it consists of performing the case distinction, then giving a simple argument for each case.

With the proof of *dmatchSound* finished, we can conclude that *dmatch* always returns a correct parse tree, i.e. that *dmatch* is sound. However, *dmatch* is not complete with respect to the *Match* relation: the function *dmatch* never makes a non-deterministic choice. It will not return all possible parse trees that satisfy the *Match* relation, only the first tree that it encounters. We can, however, prove that *dmatch* will find a parse tree if it exists. To express that *dmatch* returns a result, we use a trivially true postcondition; by furthermore replacing the demonic choice of the *ptAll* semantics with the angelic choice of *ptAny*, we require that *dmatch* must return a result:

$$dmatchComplete : \forall\ r\ xs\ y \rightarrow Match\ r\ xs\ y \rightarrow$$
$$[\![ dmatch'\ (r , xs) ]\!]_{ptRec\ matchSpec\ ::\ ptAny\ ::\ Nil}\ (\lambda\ \_ \rightarrow \top)$$

The proof is short, since *dmatch* can only *fail* when it encounters an empty string and a regular expression that does not match the empty string, which contradicts the assumption *Match r xs y*:

```
dmatchComplete r Nil y H with ε? r
... | yes p = tt
```

```
    ... | no ¬p = ¬p (_ , H)
    dmatchComplete r (x :: xs) y H y′ H′ = tt
```

In the proofs of *dmatchSound* and *dmatchComplete*, we demonstrate the power of predicate transformer semantics for effects: by separating syntax and semantics, we can easily describe different aspects (soundness and completeness) of the one definition of *dmatch*. Since the soundness and completeness result we have proved imply partial correctness, and partial correctness and termination imply total correctness, we can conclude that *dmatch* is a totally correct parser for regular languages.

## 8 Discussion

### 8.1 Related work

In this paper, we have described a representation of parsers and shown how to perform verification of parsers in this representation. We will discuss how our work relates to other parser verifications. Our sections on regular expressions have a similar structure to a Functional Pearl by Harper [Har99]. The main difference is that our work is based on formal verification using Agda, while Harper uses manual and informal reasoning. The sections on context-free grammars could be compared to work by Danielsson [Dan10] and Firsov [Fir16]. Here the difference, apart from a different parsing algorithm, can be found in how (non)termination is dealt with. We opt for a strong separation of syntax and semantics, using the *Rec* effect to give the syntax of programs regardless of termination, later proving the semantic property of termination. In contrast, Danielsson; Firsov deal with termination syntactically, either by incorporating delay and force operators in the grammar, or explicitly passing around a proof of termination in the definition of the parser.

A different representation of languages used in verification is the coinductive trie [Abe16]. The approach of Abel is in the opposite direction to ours: in order to verify constructions on automata, the language they accept is mapped to a trie, then this trie is compared to the trie that we get by applying the corresponding constructions on tries. Similarly, Abel, Adelsberger, and Setzer [AAS17] use a coinductive type to represent effectful programs with arbitrarily large input. These two coinductive constructions carry proofs of productivity, in the form of sized types, in their definitions, again mixing syntax and semantics.

### 8.2 Open issues

In the process of this verification, we have solved some open issues in the area of predicate transformer semantics and leave others open. Swierstra and Baanen [SB19] mention two avenues of further work that our work makes advances on: the semantics for combinations of effects and the verification of non-trivial programs using algebraic effects. Still, we chose to verify parsers with applying predicate transformers to them in the back of our mind, so the goal of verifying a practical program remains a step further.

We have described how coproducts allow for combinations of effect syntax and semantics, and how an individual handler interacts with these semantics. The interaction between different effects means applying handlers in a different order can result in different semantics. We assign predicate transformer semantics to a combination of effects all at once, specifying their interaction explicitly. Can we assign semantics to effects such that they interact in a similar way as handlers do?

Another issue that remains is dealing with other representations of the free monad. The *Free* datatype could be replaced with more efficient versions to run practical computations [KSS13; KI15]. We expect that predicate transformer semantics, although arising from a fold on the *Free* monad, will generalize without problems to these more advanced representations.

### 8.3 Conclusions

In conclusion, the two distinguishing features of our work are formality and modularity. We could introduce the combination of effects, petrol-driven termination, semantics for state and variant-based termination without impacting existing definitions. We strictly separate the syntax and semantics of the programs, and partial correctness from termination. This results in verification proofs that do not need to carry around many goals, allowing most of them to consist of unfolding the definition and filling in the obvious terms.

We should also note that the engineering effort expected by Swierstra and Baanen has not been needed for our paper. The optimist can conclude that the elegance of our framework caused it to prevent the feared level of complication; the pessimist can conclude that the real hard work will be required as soon as we encounter a real-world application.

## References

[AAG03] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. "Categories of Containers". In: In Proceedings of Foundations of Software Science and Computation Structures. 2003.

[AAS17] Andreas Abel, Stephan Adelsberger, and Anton Setzer. "Interactive programming in Agda – Objects and graphical user interfaces". In: Journal of Functional Programming 27 (Feb. 2017). DOI: `10.1017/S0956796816000319`.

[Abe16] Andreas Abel. "Equational Reasoning about Formal Languages in Coalgebraic Style". preprint available at `http://www.cse.chalmers.se/~abela/jlamp17.pdf`. Dec. 2016.

[BP15] Andrej Bauer and Matija Pretnar. "Programming with algebraic effects and handlers". In: Journal of Logical and Algebraic Methods in Programming 84.1 (2015). Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011, pp. 108–123. ISSN: 2352-2208. DOI: `https://doi.org/10.1016/j.jlamp.2014.02.001`.

[Brz64] Janusz A. Brzozowski. "Derivatives of Regular Expressions". In: J. ACM 11.4 (Oct. 1964), pp. 481–494. ISSN: 0004-5411. DOI: `10.1145/321239.321249`. URL: `http://doi.acm.org/10.1145/321239.321249`.

[Dan10] Nils Anders Danielsson. "Total Parser Combinators". In: SIGPLAN Not. 45.9 (Sept. 2010), pp. 285–296. ISSN: 0362-1340. DOI: `10.1145/1932681.1863585`. URL: `http://doi.acm.org/10.1145/1932681.1863585`.

[Fir16] Denis Firsov. "Certification of Context-Free Grammar Algorithms". PhD thesis. Institute of Cybernetics at Tallinn University of Technology, 2016.

[Har99] Robert Harper. “Proof-directed debugging”. In: Journal of Functional Programming 9.4 (1999), pp. 463–469. DOI: `10.1017/S0956796899003378`.

[Hut92] Graham Hutton. “Higher-order functions for parsing”. In: Journal of Functional Programming 2.3 (1992), pp. 323–343. DOI: `10.1017/S0956796800000411`.

[KI15] Oleg Kiselyov and Hiromi Ishii. “Freer Monads, More Extensible Effects”. In: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell. Haskell ’15. Vancouver, BC, Canada: ACM, 2015, pp. 94–105. ISBN: 978-1-4503-3808-0. DOI: `10.1145/2804302.2804319`. URL: `http://doi.acm.org/10.1145/2804302.2804319`.

[KSS13] Oleg Kiselyov, Amr Sabry, and Cameron Swords. “Extensible Effects: An Alternative to Monad Transformers”. In: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell. Haskell ’13. Boston, Massachusetts, USA: ACM, 2013, pp. 59–70. ISBN: 978-1-4503-2383-3. DOI: `10.1145/2503778.2503791`. URL: `http://doi.acm.org/10.1145/2503778.2503791`.

[McB15] Conor McBride. “Turing-Completeness Totally Free”. In: Mathematics of Program Construction. Ed. by Ralf Hinze and Janis Voigtländer. Cham: Springer International Publishing, 2015, pp. 257–275. ISBN: 978-3-319-19797-5.

[Nor07] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. Chalmers University of Technology, 2007.

[PP03] Gordon Plotkin and John Power. “Algebraic Operations and Generic Effects”. In: Applied Categorical Structures 11.1 (Feb. 2003), pp. 69–94. ISSN: 1572-9095. DOI: `10.1023/A:1023064908962`. URL: `https://doi.org/10.1023/A:1023064908962`.

[SB19] Wouter Swierstra and Tim Baanen. “A predicate transformer semantics for effects (Functional Pearl)”. In: Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming. ICFP ’19. 2019. DOI: `10.1145/3341707`.

[SD96] S. Doaitse Swierstra and Luc Duponcheel. “Deterministic, Error-Correcting Combinator Parsers”. In: Advanced Functional Programming. Springer-Verlag, 1996, pp. 184–207.

[Swi08] Wouter Swierstra. “Data types à la carte”. In: Journal of Functional Programming 18.4 (2008), pp. 423–436. DOI: `10.1017/S0956796808006758`.

[Wad85] Philip Wadler. “How to Replace Failure by a List of Successes”. In: Proc. Of a Conference on Functional Programming Languages and Computer Architecture. Nancy, France: Springer-Verlag New York, Inc., 1985, pp. 113–128. ISBN: 3-387-15975-4. URL: `http://dl.acm.org/citation.cfm?id=5280.5288`.

[WSH14] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. “Effect Handlers in Scope”. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. Haskell ’14. Gothenburg, Sweden: ACM, 2014, pp. 1–12. ISBN: 978-1-4503-3041-1. DOI: `10.1145/2633357.2633358`.