Combining predicate transformer semantics for effects: a case study in parsing regular languages

Anne Baanen
Vrije Universiteit Amsterdam

Wouter Swierstra
Utrecht University

This paper describes how to verify a parser for regular expressions in a functional programming language using predicate transformer semantics for a variety of effects. Where our previous work in this area focused on the semantics for a single effect, parsing requires a combination of effects: non-determinism, general recursion and mutable state. Reasoning about such combinations of effects is notoriously difficult, yet our approach using predicate transformers enables the careful separation of program syntax, correctness proofs and termination proofs.

1 Introduction

There is a significant body of work on parsing using combinators in functional programming languages [33, 12, 8, 30, 17, 15, 9, 21], among many others. Yet how can we ensure that these parsers are correct? There is notably less work that attempts to answer this question [5, 7].

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 [10, 13].

In this paper, we explore a novel approach, drawing inspiration from recent work on algebraic effects [3, 35, 20]. We demonstrate how to reason about all parsers uniformly using predicate transformers [32]. 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, meaning we can introduce new effects (***Rec*** in Section 4), semantics (***hParser*** in Section 5) and specifications (***terminates-in*** in Section 6) when they are needed, without having to rework the previous definitions. In particular, our careful treatment of general recursion lets us separate partial correctness from the proof of termination cleanly. Most existing proofs require combinators to guarantee that the string being parsed decreases, conflating these two issues.

In particular, the sections of this paper make the following contributions:

- After quickly revisiting our previous work on predicate transformer semantics for effects (Section 2), we show how 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.
- Although the resulting parser is not guaranteed to terminate, we can define another implementation using Brzozowski derivatives (Section 5), introducing an additional effect and its semantics in the process.

Submitted to:
MSFP 2020

- Finally, we show that the derivative-based implementation terminates and refines the original parser (Section 6).

The goal of our work is not so much the verification of a parser for regular languages, which has been done before [11, 16]. Instead, we aim to illustrate the steps of incrementally developing and verifying a parser using predicate transformers and algebraic effects. This work is in the spirit of a Theoretical Pearl [11]: we begin by defining a *match* function that does not terminate. The remainder of the paper then shows how to fix this function, without having to redo the complete proof of correctness.

All the programs and proofs in this paper are written in the dependently typed language Agda [23]. The full source code, including lemmas we have chosen to omit for sake of readability, is available online.[1] Apart from postulating function extensionality, we remain entirely within Agda's default theory.

2 Recap: algebraic effects and predicate transformers

Algebraic effects separate the syntax and semantics of effectful operations. In this paper, we will model them by taking the free monad over a given signature [14], describing certain operations. Signatures are represented by the type *Sig*, as follows:

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

This is Agda syntax for defining a type *Sig* with constructor $mkSig : (C : Set) \to (C \to Set) \to Sig$ and two fields, $C : Sig \to Set$ and $R : (e : Sig) \to C\ e \to Set$. 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 [1]. The following signature describes two commands: the non-deterministic choice between two values, *Choice*; and a failure operator, *Fail*. The response type *RNondet* is defined by pattern matching on the command. If the command is *Choice*, the response is a *Bool*; the *Fail* command gives no response, indicated by the empty type $\bot$.

```
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:

[1] https://github.com/Vierkantor/refinement-parsers

$$
\begin{array}{l}
\textbf{data}\ \mathit{Free}\ (e : \mathit{Sig})\ (a : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\
\quad \mathit{Pure} : a \rightarrow \mathit{Free}\ e\ a \\
\quad \mathit{Op} : (c : C\ e) \rightarrow (R\ e\ c \rightarrow \mathit{Free}\ e\ a) \rightarrow \mathit{Free}\ e\ a
\end{array}
$$

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

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

To facilitate programming with effects, we define the following smart constructors, sometimes referred to as generic effects in the literature [25]:

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

The empty parentheses () in the definition of *fail* are Agda syntax for an argument in an uninhabited type, hence no body for the lambda is provided.

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 $P : a \rightarrow \mathit{Set}$ to a predicate of the entire computation of type $\mathit{Free}\ (\mathit{mkSig}\ C\ R)\ a \rightarrow \mathit{Set}$.

$$
\begin{array}{l}
[\![\_]\!] : (\mathit{alg} : (c : C) \rightarrow (R\ c \rightarrow \mathit{Set}) \rightarrow \mathit{Set}) \rightarrow \mathit{Free}\ (\mathit{mkSig}\ C\ R)\ a \rightarrow (a \rightarrow \mathit{Set}) \rightarrow \mathit{Set} \\
[\![\mathit{Pure}\ x]\!]_{\mathit{alg}}\ P = P\ x \\
[\![\mathit{Op}\ c\ k]\!]_{\mathit{alg}}\ P = \mathit{alg}\ c\ (\lambda\ x \rightarrow [\![k\ x]\!]_{\mathit{alg}}\ P)
\end{array}
$$

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 $\mathit{alg} : (c : C) \rightarrow (R\ c \rightarrow \mathit{Set}) \rightarrow \mathit{Set}$ then becomes $C \rightarrow (R \rightarrow \mathit{Set}) \rightarrow \mathit{Set}$, which is isomorphic to $(R \rightarrow \mathit{Set}) \rightarrow (C \rightarrow \mathit{Set})$. Thus, *alg* has the form of a predicate transformer from postconditions of type $R \rightarrow \mathit{Set}$ into preconditions of type $C \rightarrow \mathit{Set}$.

Two considerations cause us to define the types as $\mathit{alg} : (c : C) \rightarrow (R\ c \rightarrow \mathit{Set}) \rightarrow \mathit{Set}$ and $[\![\_]\!]_{\mathit{alg}} : \mathit{Free}\ (\mathit{mkSig}\ C\ R)\ a \rightarrow (a \rightarrow \mathit{Set}) \rightarrow \mathit{Set}$. By passing the command $c : C$ as first argument to *alg*, we allow R to depend on c. Moreover, $[\![\_]\!]_{\mathit{alg}}$ 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 $(a \rightarrow \mathit{Set}) \rightarrow \mathit{Set}$.

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:

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

A different semantics may instead require that P holds on any of the return values:

$$
\begin{array}{l}
\mathit{ptAny} : (c : \mathit{CNondet}) \rightarrow (\mathit{RNondet}\ c \rightarrow \mathit{Set}) \rightarrow \mathit{Set} \\
\mathit{ptAny}\ \mathit{Fail}\ P \quad\ \ = \bot \\
\mathit{ptAny}\ \mathit{Choice}\ P = P\ \mathit{True} \vee P\ \mathit{False}
\end{array}
$$

Predicate transformers provide a single semantic domain to relate programs and specifications [22]. Throughout this paper, we will consider specifications consisting of a pre- and postcondition:

$$
\begin{array}{l}
\textbf{record}\ \mathit{Spec}\ (a : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\
\quad \textbf{constructor}\ [\_,\_] \\
\quad \textbf{field} \\
\quad\quad \mathit{pre} : \mathit{Set} \\
\quad\quad \mathit{post} : a \to \mathit{Set}
\end{array}
$$

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

$$
\begin{array}{l}
[\![\_,\_]\!]_{\mathrm{spec}} : \mathit{Spec}\ a \to (a \to \mathit{Set}) \to \mathit{Set} \\
[\![\mathit{pre},\mathit{post}]\!]_{\mathrm{spec}}\ P = \mathit{pre} \wedge (\forall\ o \to \mathit{post}\ o \to P\ o)
\end{array}
$$

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:

$$
\begin{array}{l}
\_\sqsubseteq\_ : (\mathit{pt}_1\ \mathit{pt}_2 : (a \to \mathit{Set}) \to \mathit{Set}) \to \mathit{Set} \\
\mathit{pt}_1 \sqsubseteq \mathit{pt}_2 = \forall\ P \to \mathit{pt}_1\ P \to \mathit{pt}_2\ P
\end{array}
$$

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 will define and verify 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. An element of this type represents the syntax of a regular expression.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Regex} : \mathit{Set}\ \textbf{where}} \\
\quad \mathit{Empty} & : & \mathit{Regex} \\
\quad \mathit{Epsilon} & : & \mathit{Regex} \\
\quad \mathit{Singleton} & : & \mathit{Char} \to \mathit{Regex} \\
\quad \_|\_ & : & \mathit{Regex} \to \mathit{Regex} \to \mathit{Regex} \\
\quad \_\cdot\_ & : & \mathit{Regex} \to \mathit{Regex} \to \mathit{Regex} \\
\quad \_\star & : & \mathit{Regex} \to \mathit{Regex}
\end{array}
$$

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 ($\_\cdot\_$) and linear repetition denoted by the Kleene star ($\_\star$).

The input to the regular expression matcher will be a *String* together with a *Regex* denoting the language to match the string against. What should our 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:

$$
\begin{array}{ll}
\mathit{Tree} : \mathit{Regex} \rightarrow \mathit{Set} & \\
\mathit{Tree}\ \mathit{Empty} & = \bot \\
\mathit{Tree}\ \mathit{Epsilon} & = \top \\
\mathit{Tree}\ (\mathit{Singleton}\ \_) & = \mathit{Char} \\
\mathit{Tree}\ (l \mid r) & = \mathit{Either}\ (\mathit{Tree}\ l)\ (\mathit{Tree}\ r) \\
\mathit{Tree}\ (l \cdot r) & = \mathit{Pair}\ (\mathit{Tree}\ l)\ (\mathit{Tree}\ r) \\
\mathit{Tree}\ (r \star) & = \mathit{List}\ (\mathit{Tree}\ r)
\end{array}
$$

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

$$
\mathit{match} : (r : \mathit{Regex})\ (\mathit{xs} : \mathit{String}) \rightarrow \mathit{Free}\ \mathit{Nondet}\ (\mathit{Tree}\ r)
$$

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:

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Match} : (r : \mathit{Regex}) \rightarrow \mathit{String} \rightarrow \mathit{Tree}\ r \rightarrow \mathit{Set}\ \textbf{where} & \\
\quad \mathit{Epsilon} & : \mathit{Match}\ \mathit{Epsilon}\ \mathit{Nil}\ \mathit{tt} \\
\quad \mathit{Singleton} & : \mathit{Match}\ (\mathit{Singleton}\ x)\ (x :: \mathit{Nil})\ x \\
\quad \mathit{OrLeft} & : \mathit{Match}\ l\ \mathit{xs}\ x \rightarrow \mathit{Match}\ (l \mid r)\ \mathit{xs}\ (\mathit{Inl}\ x) \\
\quad \mathit{OrRight} & : \mathit{Match}\ r\ \mathit{xs}\ x \rightarrow \mathit{Match}\ (l \mid r)\ \mathit{xs}\ (\mathit{Inr}\ x) \\
\quad \mathit{Concat} & : \mathit{Match}\ l\ \mathit{ys}\ y \rightarrow \mathit{Match}\ r\ \mathit{zs}\ z \rightarrow \mathit{Match}\ (l \cdot r)\ (\mathit{ys} +\!\!+ \mathit{zs})\ (y , z) \\
\quad \mathit{StarNil} & : \mathit{Match}\ (r \star)\ \mathit{Nil}\ \mathit{Nil} \\
\quad \mathit{StarConcat} & : \mathit{Match}\ (r \cdot (r \star))\ \mathit{xs}\ (y , \mathit{ys}) \rightarrow \mathit{Match}\ (r \star)\ \mathit{xs}\ (y :: \mathit{ys})
\end{array}
$$

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 *ys* and *zs* – although there may be many possible ways to decompose a string into two substrings. Finally, the two constructors for the Kleene star, $r \star$, 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:

$$
\begin{array}{llll}
\mathit{match} : (r : \mathit{Regex})\ (\mathit{xs} : \mathit{String}) & \rightarrow \mathit{Free\ Nondet}\ (\mathit{Tree}\ r) \\
\mathit{match\ Empty} & \mathit{xs} & = \mathit{fail} \\
\mathit{match\ Epsilon} & \mathit{Nil} & = \mathit{Pure\ tt} \\
\mathit{match\ Epsilon} & (\_ :: \_) & = \mathit{fail} \\
\mathit{match}\ (\mathit{Singleton}\ c) & \mathit{Nil} & = \mathit{fail} \\
\mathit{match}\ (\mathit{Singleton}\ c) & (x :: \mathit{Nil}) & \textbf{with}\ c \stackrel{?}{=} x \\
\mathit{match}\ (\mathit{Singleton}\ c) & (c :: \mathit{Nil}) & \mid \mathit{yes\ refl} = \mathit{Pure}\ c \\
\mathit{match}\ (\mathit{Singleton}\ c) & (x :: \mathit{Nil}) & \mid \mathit{no}\ \neg p \ \ = \mathit{fail} \\
\mathit{match}\ (\mathit{Singleton}\ c) & (\_ :: \_ :: \_) & = \mathit{fail} \\
\mathit{match}\ (l \mid r) & \mathit{xs} & = \mathit{choice}\ (\mathit{Inl}\ \langle\$\rangle\ \mathit{match}\ l\ \mathit{xs})\ (\mathit{Inr}\ \langle\$\rangle\ \mathit{match}\ r\ \mathit{xs}) \\
\mathit{match}\ (l \cdot r) & \mathit{xs} & = \mathrm{do}\ (\mathit{ys}, \mathit{zs}) \leftarrow \mathit{allSplits}\ \mathit{xs} \\
 & & \quad\ \ y \leftarrow \mathit{match}\ l\ \mathit{ys} \\
 & & \quad\ \ z \leftarrow \mathit{match}\ r\ \mathit{zs} \\
 & & \quad\ \ \mathit{Pure}\ (y, z) \\
\mathit{match}\ (r\,\star)\ \mathit{xs} & & = \mathit{fail}
\end{array}
$$

Figure 1: The definition of the *match* function

$$
\begin{array}{l}
\mathit{allSplits} : (\mathit{xs} : \mathit{List}\ a) \rightarrow \mathit{Free\ Nondet}\ (\mathit{List}\ a \times \mathit{List}\ a) \\
\mathit{allSplits\ Nil} = \mathit{Pure}\ (\mathit{Nil}, \mathit{Nil}) \\
\mathit{allSplits}\ (x :: \mathit{xs}) = \mathit{choice} \\
\quad (\mathit{Pure}\ (\mathit{Nil}, (x :: \mathit{xs}))) \\
\quad (\mathit{allSplits}\ \mathit{xs} \gg\!= \lambda\ \{ (\mathit{ys}, \mathit{zs}) \rightarrow \mathit{Pure}\ ((x :: \mathit{ys}), \mathit{zs}) \})
\end{array}
$$

Finally, we cannot yet implement the case for the Kleene star. We could attempt to mimic the case for concatenation, attempting to match $r \cdot (r\,\star)$. This definition, however, is rejected by Agda as it is not structurally recursive. For now we choose to simply fail on all such regular expressions. In Section 4 we will fix this issue, after introducing the auxiliary definitions.

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

$$\mathit{hasNo}\star : \mathit{Regex} \rightarrow \mathit{Set}$$

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

$$
\begin{array}{l}
\mathit{pre} : (r : \mathit{Regex})\ (\mathit{xs} : \mathit{String}) \rightarrow \mathit{Set} \\
\mathit{pre}\ r\ \mathit{xs} = \mathit{hasNo}\star\ r \\
\mathit{post} : (r : \mathit{Regex})\ (\mathit{xs} : \mathit{String}) \rightarrow \mathit{Tree}\ r \rightarrow \mathit{Set} \\
\mathit{post} = \mathit{Match}
\end{array}
$$

The main correctness result can now be formulated as follows:

$$\mathit{matchSound} : \forall\ r\ \mathit{xs} \rightarrow [\![(\mathit{pre}\ r\ \mathit{xs}), (\mathit{post}\ r\ \mathit{xs})]\!]_{\mathrm{spec}} \sqsubseteq [\![\mathit{match}\ r\ \mathit{xs}]\!]_{\mathit{ptAll}}$$

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. The proof goes by induction on the regular expression *r*. Although we have omitted the proof, we will sketch the key lemmas and definitions that are necessary to complete it.

In most of the cases for *r*, the definition of *match r* is uncomplicated and the proof is similarly simple. As soon as we need to reason about programs composed using the monadic bind operator, we quickly run into issues. In particular, when verifying the case for $l \cdot r$, we would like to use our induction hypotheses on two recursive calls. To do so, 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:

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

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

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

Not only does *match* $(l \cdot r)$ result in two recursive calls, it also makes a call to a helper function *allSplits*. Thus, we also need to formulate and prove the correctness of that function, as follows:

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

Using *wpToBind*, we can incorporate the correctness proof of *allSplits* in the correctness proof of *match*. 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* $\star$ 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 [20]:

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

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 recurring directly, the 'effects' that this signature supports 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. A handler for the *Rec* effect, under the intended semantics, thus behaves like a fixed-point combinator, introducing recursion to an otherwise recursion-free language by substituting the function body in each recursive call.

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 [35, 31].

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 unchanged, while the *Op* constructor additionally takes an index into the list to specify the effect that is invoked.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Free}\ (\mathit{es} : \mathit{List}\ \mathit{Sig})\ (a : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\
&\quad \mathit{Pure} : a \to \mathit{Free}\ \mathit{es}\ a \\
&\quad \mathit{Op} : (i : e \in \mathit{es})\ (c : C\ e)\ (k : R\ e\ c \to \mathit{Free}\ \mathit{es}\ a) \to \mathit{Free}\ \mathit{es}\ a
\end{aligned}
$$

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 [6]. 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:

$$
\begin{aligned}
&\mathit{fail} : \{\!\{\, \mathit{iND} : \mathit{Nondet} \in \mathit{es} \,\}\!\} \to \mathit{Free}\ \mathit{es}\ a \\
&\mathit{fail}\ \{\!\{\, \mathit{iND} \,\}\!\} = \mathit{Op}\ \mathit{iND}\ \mathit{Fail}\ (\lambda\ ()) \\
&\mathit{choice} : \{\!\{\, \mathit{iND} : \mathit{Nondet} \in \mathit{es} \,\}\!\} \to \mathit{Free}\ \mathit{es}\ a \to \mathit{Free}\ \mathit{es}\ a \to \mathit{Free}\ \mathit{es}\ a \\
&\mathit{choice}\ \{\!\{\, \mathit{iND} \,\}\!\}\ S_1\ S_2 = \mathit{Op}\ \mathit{iND}\ \mathit{Choice}\ (\lambda\ b \to \mathrm{if}\ b\ \mathrm{then}\ S_1\ \mathrm{else}\ S_2) \\
&\mathit{call} : \{\!\{\, \mathit{iRec} : \mathit{Rec}\ I\ O \in \mathit{es} \,\}\!\} \to (i : I) \to \mathit{Free}\ \mathit{es}\ (O\ i) \\
&\mathit{call}\ \{\!\{\, \mathit{iRec} \,\}\!\}\ i = \mathit{Op}\ \mathit{iRec}\ i\ \mathit{Pure}
\end{aligned}
$$

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 the type of generally recursive functions with effects in *es*, i.e. Kleisli arrows into $\mathit{Free}\ (\mathit{Rec}\ \_\ \_ :: \mathit{es})$.

$$
\begin{aligned}
&\_ \overset{-}{\looparrowright} \_ : (I : \mathit{Set})\ (\mathit{es} : \mathit{List}\ \mathit{Sig})\ (O : I \to \mathit{Set}) \to \mathit{Set} \\
&I \overset{es}{\looparrowright} O = (i : I) \to \mathit{Free}\ (\mathit{Rec}\ I\ O :: \mathit{es})\ (O\ i)
\end{aligned}
$$

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.

$$
\begin{array}{l}
\textbf{record}\ \mathit{PT}\ (e : \mathit{Sig}) : \mathit{Set}\ \textbf{where} \\
\quad \textbf{constructor}\ \mathit{mkPT} \\
\quad \textbf{field} \\
\qquad \mathit{pt}\quad\ \ : (c : C\ e) \to (R\ e\ c \to \mathit{Set}) \to \mathit{Set} \\
\qquad \mathit{mono} : \forall\ c\ P\ P' \to (\forall\ x \to P\ x \to P'\ x) \to \mathit{pt}\ c\ P \to \mathit{pt}\ c\ P' \\
\textbf{data}\ \mathit{PTs} : \mathit{List}\ \mathit{Sig} \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{Nil}\quad\ : \mathit{PTs}\ \mathit{Nil} \\
\quad \_{::}\_ : \mathit{PT}\ e \to \mathit{PTs}\ es \to \mathit{PTs}\ (e :: es)
\end{array}
$$

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 of Section 6, rely on the monotonicity of the underlying predicate transformers; for each semantics we present, the proof of monotonicity is immediate.

Given 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.

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

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

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

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:

$$
\begin{array}{l}
\mathit{ptRec} : ((i : I) \to O\ i \to \mathit{Set}) \to \mathit{PT}\ (\mathit{Rec}\ I\ O) \\
\mathit{PT}.\mathit{pt}\ (\mathit{ptRec}\ R)\ i\ P = \forall\ o \to R\ i\ o \to P\ o
\end{array}
$$

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

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:

$$\mathit{allSplits} : \{\{\, \mathit{iND} : \mathit{Nondet} \in \mathit{es} \,\}\} \to \mathit{List}\ a \to \mathit{Free}\ \mathit{es}\ (\mathit{List}\ a \times \mathit{List}\ a)$$
$$\mathit{match} : \{\{\, \mathit{iND} : \mathit{Nondet} \in \mathit{es} \,\}\} \to (x : \mathit{Regex} \times \mathit{String}) \overset{es}{\leadsto} \mathit{Tree}\ (\mathit{Pair.fst}\ x)$$

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:

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

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*.

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

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.

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

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.

5 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, changing the regular expression to be matched based on the characters we have seen.

The Brzozowski derivative of a formal language L with respect to a character x consists of all strings xs such that $x :: \mathit{xs} \in L$ [4]. 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 $x :: \mathit{xs}$ 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 [4], we use a helper function $\varepsilon?$ that decides whether an expression matches the empty string.

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

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

$$\begin{array}{llll}
d_/d_ : \mathit{Regex} \to \mathit{Char} \to \mathit{Regex} & & & \\
d\ \mathit{Empty} & /d\ c = \mathit{Empty} & & \\
d\ \mathit{Epsilon} & /d\ c = \mathit{Empty} & & \\
d\ \mathit{Singleton}\ x & /d\ c\ \textbf{with}\ c \stackrel{?}{=} x & & \\
\ldots & \quad \mid \mathit{yes}\ p & = \mathit{Epsilon} & \\
\ldots & \quad \mid \mathit{no}\ \neg p & = \mathit{Empty} & \\
d\ l \cdot r & /d\ c\ \textbf{with}\ \varepsilon?\ l & & \\
\ldots & \quad \mid \mathit{yes}\ p & = ((d\ l\ /d\ c) \cdot r) \mid (d\ r\ /d\ c) & \\
\ldots & \quad \mid \mathit{no}\ \neg p & = (d\ l\ /d\ c) \cdot r & \\
d\ l \mid r & /d\ c = (d\ l\ /d\ c) \mid (d\ r\ /d\ c) & & \\
d\ r \star & /d\ c = (d\ r\ /d\ c) \cdot (r \star) & &
\end{array}$$

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 differentiation, we name it *integralTree*:

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

Its definition closely follows the pattern matching performed in the definition of $d\_/d\_$.

The description of a derivative-based matcher is stateful: we perform a step by removing a character from the input string. To match the description, we introduce new effect *Parser* which provides a parser-specific interface to this state. The *Parser* effect has one command *Symbol* that returns a *Maybe Char*. Calling *Symbol* will return *just* the head of the unparsed remainder (advancing the string by one character) or *nothing* if the string has been totally consumed.

$$\begin{array}{l}
\textbf{data}\ \mathit{CParser} : \mathit{Set}\ \textbf{where} \\
\quad \mathit{Symbol} : \mathit{CParser} \\
\mathit{RParser} : \mathit{CParser} \to \mathit{Set} \\
\mathit{RParser}\ \mathit{Symbol} = \mathit{Maybe}\ \mathit{Char} \\
\mathit{Parser} = \mathit{mkSig}\ \mathit{CParser}\ \mathit{RParser} \\
\mathit{symbol} : \{\{\, \mathit{iP} : \mathit{Parser} \in \mathit{es} \,\}\} \to \mathit{Free}\ \mathit{es}\ (\mathit{Maybe}\ \mathit{Char}) \\
\mathit{symbol}\ \{\{\, \mathit{iP} \,\}\} = \mathit{Op}\ \mathit{iP}\ \mathit{Symbol}\ \mathit{Pure}
\end{array}$$

The code for the new parser, *dmatch*, is now only a few lines long. When the input contains at least one character, we use the derivative to match the first character and recurse; when the input string is empty, we check that the expression matches the empty string.

$$
\begin{aligned}
&\mathit{dmatch} : \{\{\, \mathit{iP} : \mathit{Parser} \in \mathit{es} \,\}\}\ \{\{\, \mathit{iND} : \mathit{Nondet} \in \mathit{es} \,\}\} \to \mathit{Regex} \overset{\mathit{es}}{\looparrowright} \mathit{Tree} \\
&\mathit{dmatch}\ r = \mathit{symbol} \gg\!\!= \mathit{maybe} \\
&\quad (\lambda\, x \to \mathit{integralTree}\ r\ \langle\$\rangle\ \mathit{call}\ (d\ r\ /d\ x)) \\
&\quad (\text{if}\ p \leftarrow \varepsilon ?\ r\ \text{then}\ \mathit{Pure}\ (\mathit{Sigma.fst}\ p)\ \text{else}\ \mathit{fail})
\end{aligned}
$$

Here, $\mathit{maybe}\, f\, y$ takes a *Maybe* value and applies f to the value in *just*, or returns y if it is *nothing*. Although the parser is easily seen to terminate in the intended semantics (since a character is removed from the input string between each recursive call), a semantics where the call to *symbol* always returns *just* a character causes *dmatch* to diverge. The termination of *dmatch* is not a syntactical property, as reflected by the use of the recursive *call* in its definition, and the custom arrow used in the type of functions defined using general recursion.

Adding the new effect *Parser* to our repertoire thus 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' arguments and return values (here of type $r : \mathit{Regex}$ and $\mathit{Tree}\ r$ 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:

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

In this article, we want to demonstrate the modularity of predicate transformer semantics, allowing us to introduce new notions without having to rework existing constructions. To illustrate how the semantics mesh well with other forms of semantics, we do not use *ptParser* as semantics for *Parser* in the remainder. We give denotational semantics, in the form of an effect handler for *Parser* [26, 35]:

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

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

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

Note that dmatch' has exactly the type of the previously defined *match*, conveniently allowing us to re-use the $[\![\_]\!]_{\text{match}}$ semantics. In this way, the handler *hParser* "hides" the implementation detail that the *Parser* effect was used.

6 Proving total correctness

We finish the development process by proving that *dmatch* is correct. The first step in this proof is that *dmatch* always terminates. To express the termination of a recursive computation, we define the following predicate, *terminates-in*:

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

Given a program S that calls the recursive function $f : I \overset{es}{\looparrowright} O$, we check whether the computation requires no more than a fixed number of steps to terminate.

Since *dmatch* always consumes a character before recurring, 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 $\textit{integralTree}\ r\ \langle\$\rangle\ \textit{dmatch}'\ (d\ r\ /d\ x\ ,\textit{xs})$, which we rewrite using the associativity monad law in a lemma called *terminates-fmap*.

$$
\begin{aligned}
&\textit{dmatchTerminates} : (r : \textit{Regex})\ (\textit{xs} : \textit{String}) \to\\
&\quad \textit{terminates-in}\ (\textit{ptAll} :: \textit{Nil})\ (\textit{dmatch}')\ (\textit{dmatch}'\ (r\ ,\textit{xs}))\ (\textit{length}\ \textit{xs})\\
&\textit{dmatchTerminates}\ r\ \textit{Nil}\ \textbf{with}\ \varepsilon ?\ r\\
&\textit{dmatchTerminates}\ r\ \textit{Nil}\ \mid \textit{yes}\ p\ \ = \textit{tt}\\
&\textit{dmatchTerminates}\ r\ \textit{Nil}\ \mid \textit{no}\ \neg p = \textit{tt}\\
&\textit{dmatchTerminates}\ r\ (x :: \textit{xs}) = \textit{terminates-fmap}\ (\textit{length}\ \textit{xs})\ (\textit{dmatch}'\ ((d\ r\ /d\ x)\ ,\textit{xs}))\\
&\quad (\textit{dmatchTerminates}\ (d\ r\ /d\ x)\ \textit{xs})\\
&\quad \textbf{where}\\
&\quad \textit{terminates-fmap} : \{f : I \overset{es}{\looparrowright} O\}\ \{g : a \to b\}\ (n : \mathbb{N})\ (S : \textit{Free}\ (\textit{Rec}\ I\ O :: \textit{es})\ a) \to\\
&\qquad \textit{terminates-in}\ \textit{pts}\ f\ S\ n \to \textit{terminates-in}\ \textit{pts}\ f\ (g\ \langle\$\rangle\ S)\ n
\end{aligned}
$$

Apart from termination, correctness consists of soundness and completeness: the parse trees returned by *dmatch* should satisfy the specification given by the original *Match* relation, and for any string that matches the regular expression, *dmatch* should return a parse tree. In the *ptAll* semantics, a nondeterministic program S is refined by T if and only if the output values of T are a subset of the output values of S; conversely S is refined by T in the *ptAny* semantics if and only if the output values of S are a subset of the output values of T. These properties allow us to express program correctness in terms of refinement.

We can show soundness of *dmatch* by proving it refines *match*. Transitivity of the refinement relation then allows us to 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 $\textit{xs}$ such that r matches $x :: \textit{xs}$.

$$
\textit{derivativeCorrect} : \forall\ r \to \textit{Match}\ (d\ r\ /d\ x)\ \textit{xs}\ y \to \textit{Match}\ r\ (x :: \textit{xs})\ (\textit{integralTree}\ r\ y)
$$

The proof is straightforward by induction on the derivation of type $\textit{Match}\ (d\ r\ /d\ x)\ \textit{xs}\ y$.

Using the preceding lemmas, we can prove the partial correctness of *dmatch*.

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

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 this case distinction, then giving a simple argument for each case.

Although we successfully proved *dmatch* is sound with respect to the *Match* relation, it is not complete: 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 any result at all, 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:

$$
\begin{aligned}
&\mathit{dmatchComplete} : \forall\, r\; xs\; y \;\rightarrow\; \mathit{Match}\; r\; xs\; y \;\rightarrow \\
&\quad [\![\mathit{dmatch}'\;(r\;,xs)]\!]_{\mathit{ptRec}\;\mathit{matchSpec}\;::\;\mathit{ptAny}\;::\;\mathit{Nil}}\;(\lambda\;\_\;\rightarrow\;\top)
\end{aligned}
$$

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 $\mathit{Match}\; r\; xs\; y$:

$$
\begin{aligned}
&\mathit{dmatchComplete}\; r\; \mathit{Nil}\; y\; H\; \textbf{with}\; \varepsilon?\; r \\
&...\;\mid\; \mathit{yes}\; p \;=\; \mathit{tt} \\
&...\;\mid\; \mathit{no}\; \neg p \;=\; \neg p\;(\_\;, H) \\
&\mathit{dmatchComplete}\; r\; (x\;::\;xs)\; y\; H\; y'\; H' \;=\; \mathit{tt}
\end{aligned}
$$

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.

7 Discussion

Related work

The refinement calculus has traditionally been used to verify imperative programs [22]. In this paper, however, we show how many of the ideas from the refinement calculus can also be used in the verification of functional programs [32]. The Dijkstra monad, introduced in the language F⋆, also uses a predicate transformer semantics for verifying effectful programs by collecting the proof obligations for verification [29, 2, 19]. This paper demonstrates how similar verification efforts can be undertaken directly in an interactive theorem prover such as Agda. The separation of syntax and semantics in our approach allows for verification to be performed in several steps, such as we did for *dmatchTerminates*, *dmatchSound* and *dmatchComplete*, adding new effects as we need them.

Our running example of the regular expression parser is inspired by the development of a regular expression parser by Harper [11]. More recently, Korkut, Trifunovski, and Licata [16] adapted the Functional Pearl to Agda. A direct translation of Harper's definitions is not possible: they are rejected by Agda's termination checker because they are not structurally recursive. Korkut, Trifunovski, and Licata show how the defunctionalization of Harper's matcher, written in continuation-passing style, is accepted by Agda's termination checker.

Formally verified parsers for a more general class of languages have been developed before: Danielsson [5], Firsov [7], and Ridge [27], among others, have previously shown how to verify parsers developed in a functional language. In these developments, semantics are defined specialized to the domain of parsing, while our semantics arise from combining a generic set of effect semantics. Furthermore, we allow our parsers to be written using general recursion directly, whereas most existing approaches 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. The modularity of our setup allows us to separate partial and total correctness cleanly.

There are various ways to represent a combination of effects such as used in parsers. A traditional approach is to use monad transformers to add each effect in turn, producing a complicated monad that incorporates all required operations [18]. More recently, graded monads were introduced as a way to indicate more precisely the effects used in a specific computation [24, 34]. With some slight changes to the types of *Pure* and _≫=_, the *Free* monad can be viewed as graded over the free monoid *List Sig* generated by the type of effect signatures. As this monad containing the computation is freely generated, it does not require us to assign any semantics to the effects ahead of time.

Open issues

This paper builds upon our previous results [32] by demonstrating their use in non-trivial development. In the process, we show how to combine predicate transformer semantics and reason about programs using a combination of effects.

Our approach relies on using coproducts to combine effect syntax. 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—but we would still like to explore how to handle effects one-by-one, allowing for greater flexibility when assigning semantics to effectful programs [28, 35].

Conclusions

In conclusion, we have illustrated the approach to developing verified software in a proof assistant using a predicate transformer semantics for effects for a non-trivial example. We believe this approach enables us to add new effects in a modular fashion, while still being able to re-use any existing proofs. Along the way, we demonstrated how to combine different effects and define different semantics for these effects, without impacting existing definitions. As a result, the verification effort—while conceptually more challenging at times—remains fairly modular.

Acknowledgements T. Baanen has received funding from the NWO under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. “Categories of Containers”. In: Proceedings of Foundations of Software Science and Computation Structures. 2003. DOI: 10.1007/3-540-36576-1_2.

[2] Danel Ahman et al. “Dijkstra Monads for Free”. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017. Paris, France: Association for Computing Machinery, 2017, pp. 515–529. ISBN: 9781450346603. DOI: 10.1145/3009837.3009878.

[3] 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: 10.1016/j.jlamp.2014.02.001.

[4] 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.

[5] Nils Anders Danielsson. “Total Parser Combinators”. In: SIGPLAN Not. 45.9 (Sept. 2010), pp. 285–296. ISSN: 0362-1340. DOI: 10.1145/1932681.1863585.

[6] Dominique Devriese and Frank Piessens. “On the Bright Side of Type Classes: Instance Arguments in Agda”. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. ICFP ’11. Tokyo, Japan: Association for Computing Machinery, 2011, pp. 143–155. ISBN: 9781450308656. DOI: 10.1145/2034773.2034796.

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

[8] Jeroen Fokker. “Functional Parsers”. In: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text. Berlin, Heidelberg: Springer, 1995, pp. 1–23. ISBN: 3540594515. DOI: 10.1007/3-540-59451-5_1.

[9] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. “Parser Combinators for Ambiguous Left-Recursive Grammars”. In: Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages. PADL’08. San Francisco, CA, USA: Springer-Verlag, 2008, pp. 167–181. ISBN: 3540774416. DOI: 10.1007/978-3-540-77442-6_12.

[10] Jeremy Gibbons and Ralf Hinze. “Just Do It: Simple Monadic Equational Reasoning”. In: SIGPLAN Not. 46.9 (Sept. 2011), pp. 2–14. ISSN: 0362-1340. DOI: 10.1145/2034574.2034777.

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

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

[13] Graham Hutton and Diana Fulger. “Reasoning about effects: Seeing the wood through the trees”. In: Proceedings of the Ninth Symposium on Trends in Functional Programming. 2008. DOI: 10.1.1.110.5912.

[14] 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.

[15] Pieter W. M. Koopman and Marinus J. Plasmeijer. “Efficient Combinator Parsers”. In: Selected Papers from the 10th International Workshop on 10th International Workshop. IFL ’98. Berlin, Heidelberg: Springer-Verlag, 1998, pp. 120–136. ISBN: 3540662294. DOI: 10.1007/3-540-48515-5_8.

[16] Joomy Korkut, Maksim Trifunovski, and Daniel R. Licata. “Intrinsic Verification of a Regular Expression Matcher”. preprint available at http://dlicata.web.wesleyan.edu/pubs/ktl16regexp/ktl16regexp.pdf. Jan. 2016.

[17] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Tech. rep. UU-CS-2001-27. 2001.

[18] Sheng Liang, Paul Hudak, and Mark Jones. “Monad Transformers and Modular Interpreters”. In: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL ’95. San Francisco, California, USA: ACM, 1995, pp. 333–343. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199528.

[19] Kenji Maillard et al. “Dijkstra Monads for All”. In: Proc. ACM Program. Lang. 3.ICFP (July 2019). DOI: 10.1145/3341708.

[20] 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. DOI: 10.1007/978-3-319-19797-5_13.

[21] Matthew Might, David Darais, and Daniel Spiewak. “Parsing with Derivatives: A Functional Pearl”. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. ICFP ’11. Tokyo, Japan: Association for Computing Machinery, 2011, pp. 189–195. ISBN: 9781450308656. DOI: 10.1145/2034773.2034801.

[22] Caroll Morgan. Programming from Specifications. 2nd ed. Prentice Hall, 1998.

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

[24] Dominic Orchard and Tomas Petricek. “Embedding Effect Systems in Haskell”. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. Haskell ’14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 13–24. ISBN: 9781450330411. DOI: 10.1145/2633357.2633368.

[25] 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.

[26] Gordon Plotkin and Matija Pretnar. “Handlers of Algebraic Effects”. In: Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. ESOP ’09. York, UK: Springer-Verlag, 2009, pp. 80–94. ISBN: 9783642005893. DOI: 10.1007/978-3-642-00590-9_7.

[27] Tom Ridge. “Simple, Functional, Sound and Complete Parsing for All Context-Free Grammars”. In: Certified Programs and Proofs. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 103–118. ISBN: 978-3-642-25379-9. DOI: 10.1007/978-3-642-25379-9_10.

[28] Tom Schrijvers et al. “Monad Transformers and Modular Algebraic Effects: What Binds Them Together”. In: Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell. Haskell 2019. Berlin, Germany: Association for Computing Machinery, 2019, pp. 98–113. ISBN: 9781450368131. DOI: `10.1145/3331545.3342595`.

[29] Nikhil Swamy et al. “Verifying Higher-order Programs with the Dijkstra Monad”. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI ’13. Seattle, Washington, USA: ACM, 2013, pp. 387–398. ISBN: 978-1-4503-2014-6. DOI: `10.1145/2491956.2491978`.

[30] S. Doaitse Swierstra and Luc Duponcheel. “Deterministic, Error-Correcting Combinator Parsers”. In: Advanced Functional Programming. Springer-Verlag, 1996, pp. 184–207. DOI: `10.1007/3-540-61628-4_7`.

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

[32] 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`.

[33] 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. DOI: `10.1007/3-540-15975-4_33`.

[34] Philip Wadler and Peter Thiemann. “The Marriage of Effects and Monads”. In: ACM Trans. Comput. Logic 4.1 (Jan. 2003), pp. 1–32. ISSN: 1529-3785. DOI: `10.1145/601775.601776`.

[35] 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`.