Verified parsers using the refinement calculus and algebraic effects

Tim Baanen and Wouter Swierstra

Vrije Universiteit Amsterdam, Utrecht University
{t.baanen@vu.nl,w.s.swierstra@uu.nl}

There are various ways to write a parser in functional languages, for example using parser combinations. How do we ensure these parsers are correct? Previous work has shown that predicate transformers are useful for verification of programs using algebraic effects. This paper will show how predicate transformers and algebraic effects allow for formal verification of parsers.

Recap: algebraic effects and predicate transformers

Algebraic effects were introduced to allow for incorporating side effects in functional languages. For example, the effect *ENondet* allows for nondeterministic programs:

```
record Effect : Set where
  constructor eff
  field
    C : Set
    R : C → Set
data CNondet : Set where
  Fail : CNondet
  Choice : CNondet
RNondet : CNondet → Set
RNondet Fail = ⊥
RNondet Choice = Bool
ENondet = eff CNondet RNondet
```

We represent effectful programs using the *Free* datatype.

```
data Free (e : Effect) (a : Set) : Set where
  Pure : a → Free e a
  Step : (c : Effect.C e) → (Effect.R e c → Free e a) → Free e a
```

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

```
_>>=_ : ∀ {a b e} → Free e a → (a → Free e b) → Free e b
Pure x >>= f = f x
Step c k >>= f = Step c (λ x → k x >>= f)
```

The easiest way to use effects is with smart constructors:

```
fail : ∀ { a } → Free ENondet a
fail { a } = Step Fail magic
choice : ∀ { a } → Free ENondet a → Free ENondet a → Free ENondet a
choice S₁ S₂ = Step Choice λ b → if b then S₁ else S₂
```

To give specifications of programs that incorporate effects, we can use predicate transformers.

```
wpFree : { C : Set } { R : C → Set } → ((c : C) → (R c → Set) → Set) →
  { a : Set } → Free (eff C R) a → (a → Set) → Set
wpFree alg (Pure x) P = P x
wpFree alg (Step c k) P = alg c \x → wpFree alg (k x) P
```

Interestingly, these predicate transformers are exactly the catamorphisms from *Free* to *Set*.

```
ptAll : (c : CNondet) → (RNondet c → Set) → Set
ptAll Fail P = ⊤
ptAll Choice P = P True ∧ P False


wpNondetAll : { a : Set } → Free ENondet a →
  (a → Set) → Set
wpNondetAll S P = wpFree ptAll S P
```

We use pre- and postconditions to give a specification for a program. If the precondition holds on the input, the program needs to ensure the postcondition holds on the output.

```
module Spec where
  record Spec (a : Set) : Set where
    constructor [_,_]
    field
      pre : Set
      post : a → Set
  wpSpec : { a : Set } → Spec a → (a → Set) → Set
  wpSpec [ pre , post ] P = pre ∧ (∀ o → post o → P o)
```

The refinement relation expresses when one program is "better" than another. We need to take into account the semantics we want to impose on the program, so we define it in terms of the predicate transformer associated with the program.

```
_⊑_ : { a : Set } (pt₁ pt₂ : (a → Set) → Set) → Set
pt₁ ⊑ pt₂ = ∀ P → pt₁ P → pt₂ P
```

2 Almost parsing regular languages

To see how we can use the *Free* monad for writing and verifying a parser, and more specifically how we use the *ENondet* effect for writing and the *wpNondetAll* semantics for verifying a parser, we will look at parsing a given regular language. Our approach is first to define the specification of a parser, then inspect this specification to write the first implementation and prove (partial) correctness of this implementation. We will later improve this implementation by refining it.

Theorem 1 ([AU77]) *The class of* regular languages *is the smallest class such that:*

- *the empty language is regular,*
- *the language containing only the empty string is regular,*
- *for each character* `x`*, the language containing only the string* `"x"` *is regular,*
- *the union and concatenation of regular languages are regular, and*
- *the repetition of a regular language is regular.*

A regular language can defined using a regular expression, which we will represent as an element of the *Regex* datatype. An element of this type represents the syntax of a regular language, and we will generally identify a regular expression with the language it denotes.

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

Here, *Empty* is an expression for empty language (which matches no strings at all), while *Epsilon* is an expression for the language of the empty string (which matches exactly one string: `""`).

What should a parser for regular expressions output? Perhaps it could return a *Bool* indicating whether a given string matches the regular expression, or we could annotate the regular expression with capture groups, and say that the output of the parser maps each capture group to the substring that the capture group matches. In this case, we define the return type to be a parse tree mirroring the structure of the 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 Agda, we can represent the semantics of the *Regex* type by giving a relation between a *Regex* and a *String* on the one hand (the input of the matcher), and a parse tree on the other hand (the output of the parser). Note that the *Tree* type itself is not sufficient to represent the semantics, since it does not say which strings result in any given parse tree. If the *Regex* and *String* do not match, there should be no output, otherwise the output consists of all relevant parse trees. We give the relation using the following inductive definition:

```
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*, which we interpret as that 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*.

Since the definition of *Match* allows for multiple ways that a given *Regex* and *String* may match, such as in the trivial case where the *Regex* is of the form $r \mid r$, and it also has cases where there is no way to match a *Regex* and a *String*, such as where the *Regex* is *Empty*, we can immediately predict some parts of the implementation. Whenever we encounter an expression of the form $l \mid r$, we make a nondeterministic *Choice* between either l or r. Similarly, whenever we encounter the *Empty* expression, we immediately *fail*. In the previous analysis steps, we have already assumed that we implement the parser by structural recursion on the *Regex*, so let us consider other cases.

The implementation for concatenation is not as immediately obvious. One way that we can deal with it is to change the type of the parser. Instead write a parser that returns the unmatched portion of the string, and when we have to match a regular expression of the form $l \cdot r$ with a string *xs*, we match l with *xs* giving a left over string *ys*, then match r with *ys*. We can also do without changing the return values of the parser, by nondeterministically splitting the string *xs* into *ys* ++ *zs*. That is what we do in a helper function *allSplits*, which nondeterministically chooses such *ys* and *zs* and returns them as a pair.

```
allSplits : (xs : List a) → Free ENondet (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)})
```

Armed with this helper function, we can write the first part of a nondeterministic regular expression matcher, that does a case distinction on the expression and then checks that the string has the correct format.

```
match : (r : Regex) (xs : String) → Free ENondet (Tree r)
match Empty xs = fail
match Epsilon Nil = Pure tt
match Epsilon xs@ (_ :: _) = 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) xs@ (_ :: _ :: _) = fail
match (l · r) xs = do
  (ys , zs) ← allSplits xs
  y ← match l ys
  z ← match r zs
  Pure (y , z)
match (l | r) xs = choice (Inl <$> match l xs) (Inr <$> match r xs)
```

Unfortunately, we get stuck in the case of $\_\star$. We could do a similar construction to $l \cdot r$, where we split the string into two parts and match the first part with r and the second part with $r\,\star$, but this runs afoul of Agda's termination checker. Since there is no easy way to handle this case for now, we just *fail* when we encounter a regex $r\,\star$.

```
match (r ⋆) xs = fail
```

Still, we can prove that this matcher works, as long as the regex does not contain $\_\star$. In other words, we can prove that the *match* function refines a specification where the precondition states that the regex contains no Kleene star, and the postcondition states that the matching is correct, with respect to the type *Match*.

```
hasNo∗ : Regex → Set
hasNo∗ Empty = ⊤
hasNo∗ Epsilon = ⊤
hasNo∗ (Singleton x) = ⊤
hasNo∗ (l · r) = hasNo∗ l ∧ hasNo∗ r
hasNo∗ (l | r) = hasNo∗ l ∧ hasNo∗ r
hasNo∗ (r ⋆) = ⊥
pre : (r : Regex) (xs : String) → Set
pre r xs = hasNo∗ r
post : (r : Regex) (xs : String) → Tree r → Set
post = Match
```

If we now want to give a correctness proof with respect to these pre- and postconditions, we run into an issue in cases where the definition makes use of the $\_ \gg\!= \_$ operator. The *wpFree*-based semantics completely unfolds the left hand side, before it can talk about the right hand side. Whenever our matcher makes use of recursion on the left hand side of a $\_ \gg\!= \_$ (as we do in *allSplits* and

in the cases of $l \cdot r$ and $l \mid r$), we cannot make progress in our proof without reducing this left hand side to a recursion-less expression. We need a lemma relating the semantics of program composition to the semantics of individual programs, which is also known as the law of consequence for traditional predicate transformer semantics.cite?

```
consequence : ∀ {a b es P} pt (mx : Free es a) (f : a → Free es b) →
  wpFree pt mx (λ x → wpFree pt (f x) P) == wpFree pt (mx >>= f) P
consequence pt (Pure x) f = refl
consequence pt (Step c k) f = cong (pt c) (extensionality λ x → consequence pt (k x) f)
wpToBind : ∀ {a b es pt P} (mx : Free es a) (f : a → Free es b) →
  wpFree pt mx (λ x → wpFree pt (f x) P) → wpFree pt (mx >>= f) P
wpToBind {pt = pt} mx f H = subst id (consequence pt mx f) H
wpFromBind : ∀ {a b es pt P} (mx : Free es a) (f : a → Free es b) →
  wpFree pt (mx >>= f) P → wpFree pt mx (λ x → wpFree pt (f x) P)
wpFromBind {pt = pt} mx f H = subst id (sym (consequence pt mx f)) H
```

The correctness proof 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.

```
allSplitsSound : ∀ (xs : String) →
  wpSpec [ ⊤ , (λ {(ys , zs) → xs == ys ++ zs}) ] ⊑ wpNondetAll (allSplits xs)
allSplitsSound Nil       P (fst , snd) = snd _ refl
allSplitsSound (x :: xs) P (fst , snd) = snd _ refl ,
  wpToBind (allSplits xs) _ (allSplitsSound xs _ (tt , (λ {(ys , zs) H → snd _ (cong (x :: _) H)})))
```

Then, using *wpToBind*, we incorporate this correctness proof in the correctness proof of *match*. Apart from having to introduce *wpToBind*, the proof essentially follows automatically from the definitions.

```
matchSound : ∀ r xs → wpSpec [ pre r xs , post r xs ] ⊑ wpNondetAll (match r xs)
matchSound Empty xs P (preH , postH) = tt
matchSound Epsilon Nil P (preH , postH) = postH _ Epsilon
matchSound Epsilon (x :: xs) P (preH , postH) = tt
matchSound (Singleton x) Nil P (preH , postH) = tt
matchSound (Singleton x) (c :: Nil) P (preH , postH) with x ≟ c
matchSound (Singleton x) (c :: Nil) P (preH , postH) | yes refl = postH _ Singleton
matchSound (Singleton x) (c :: Nil) P (preH , postH) | no ¬p = tt
matchSound (Singleton x) (_ :: _ :: _) P (preH , postH) = tt
matchSound (l · r) xs P ((preL , preR) , postH) = wpToBind (allSplits xs) _ (allSplitsSound xs _ (tt ,
  λ {(ys , zs) splitH → wpToBind (match l ys) _ (matchSound l ys _ (preL ,
  λ y lH → wpToBind (match r zs) _ ((matchSound r zs _ (preR ,
  λ z rH → postH (y , z) (subst (λ xs → Match _ xs _) (sym splitH) (Concat lH rH)))))))}))
matchSound (l | r) xs P ((preL , preR) , postH) =
  wpToBind (match l xs) _ (matchSound l xs _ (preL , λ _ lH → postH _ (OrLeft lH))) ,
```

```
    wpToBind (match r xs) _ (matchSound r xs _ (preR , λ _ rH → postH _ (OrRight rH)))
  matchSound (r ⋆) xs P ((() , postH)
```

3 Combining nondeterminism and general recursion

The matcher we have defined in the previous section is unfinished, since it is not able to handle regular expressions that incorporate the Kleene star. The fundamental issue is that the Kleene star allows for arbitrarily many distinct matchings in certain cases. For example, matching *Epsilon* $\star$ with the string *Nil* will allow for repeating the *Epsilon* arbitrarily often, since *Epsilon* $\cdot$ (*Epsilon* $\star$) is equivalent to both *Epsilon* and *Epsilon* $\star$. Thus, we cannot fix *match* by improving Agda's termination checker.

What we will do instead is to deal with the recursion as an effect. A recursively defined (dependent) function of type $(i : I) \to O\ i$ can instead be given as an element of the type $(i : I) \to$ *Free* (*ERec I O*) (*O i*), where *ERec I O* is the effect of *general recursion* [**McBride-totally-free**]:

```
ERec : (I : Set) (O : I → Set) → Effect
ERec I O = eff I O
```

We are not yet done now that we have defined the missing effect, since replacing the effect *ENondet* with *ERec* (*Pair Regex String*) (*List String*) does not allow for nondeterminism anymore, so while the Kleene star might work, the other parts of *match* do not work anymore. We need a way to combine the two effects.

We can combine two effects in a straightforward way: given *eff* C_1 R_1 and *eff* C_2 R_2, we can define a new effect by taking the disjoint union of the commands and responses, resulting in *eff* (*Either* C_1 C_2) [R_1 , R_2], where [R_1 , R_2] is the unique map given by applying R_1 to values in C_1 and R_2 to C_2. If we want to support more effects, we can repeat this process of disjoint unions, but this quickly becomes somewhat cumbersome. For example, the disjount union construction is associative, but we would need to supply a proof of this whenever the associations of our types do not match.

Instead of building a new effect type, we modify the *Free* monad to take a list of effects instead of a single effect. The *Pure* constructor remains as it is, while the *Step* constructor takes an index into the list of effects and the command and continuation for the effect with this index.

```
data Free (es : List Effect) (a : Set) : Set where
  Pure : a → Free es a
  Step : {e : Effect} (i : e ∈ es) (c : Effect.C e) (k : Effect.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 a canonical association order of these unions. Since the disjoint union is also commutative, it would be cleaner to have the collection of effects

be unordered as well, but there does not seem to be a data type built into Agda that allows for unordered collections.

To make use of the new definition of *Free*, we need to translate the previous constructions. We can define the monadic bind $\_ \gg\!= \_$ in the same way as in the previous definition of *Free*. We also to make a small modification to the smart constructors for nondeterminism, since they now need to keep track of their position within a list of effects. Most of the bookkeeping can be offloaded to Agda's instance argument solver.

```
fail : ∀ {a es} ⦃ iND : ENondet ∈ es ⦄ → Free es a
fail ⦃ iND ⦄ = Step iND Fail magic
choice : ∀ {a es} ⦃ iND : ENondet ∈ es ⦄ → Free es a → Free es a → Free es a
choice ⦃ iND ⦄ S1 S2 = Step iND Choice λ b → if b then S1 else S2
call : ∀ {I O es} → ⦃ iRec : ERec I O ∈ es ⦄ → (i : I) → Free es (O i)
call ⦃ iRec ⦄ i = Step iRec i Pure
```

For convenience of notation, we introduce the $\_ \overset{es}{\nrightarrow} \_$ notation for general recursion, i.e. Kleisli arrows into *Free* (*ERec* _ _ :: *es*).

```
_ ↛ _ : (C : Set) (es : List Effect) (R : C → Set) → Set
C ↛[ es ] R = (c : C) → Free (eff C R :: es) (R c)
```

```
instance
  inHead : ∀ {a} {x : a} {xs : List a} → x ∈ (x :: xs)
  inHead = ∈Head
  inTail : ∀ {a} {x x' : a} {xs : List a} → ⦃ i : x ∈ xs ⦄ → x ∈ (x' :: xs)
  inTail ⦃ i ⦄ = ∈Tail i
```

Since we want the effects to behave compositionally, the semantics of the combination of effects should be similarly found in a list of predicate transformers. The type *List* ((*c* : *C*) → (*R c* → *Set*) → *Set*) is not sufficient, since we need to ensure the types match up. Using a dependent type we can define a list of predicate transformers for a list of effects: We introduce a type *PT* hwich includes a monotonicity requirement.

```
record PT (e : Effect) : Set where
  constructor mkPT
  field
    pt : (c : Effect.C e) → (Effect.R e c → Set) → Set
    mono : ∀ c → (P P' : Effect.R e c → Set) → P ⊆ P' → pt c P → pt c P'
data PTs : List Effect → Set where
  Nil : PTs Nil
  _::_ : ∀ {e es} → PT e → PTs es → PTs (e :: es)
```

Given a such a list of predicate transformers, defining the semantics of an effectful program is a straightforward generalization of *wpFree*. The *Pure* case

is identical, and in the *Step* case we find the predicate transformer at the corresponding index to the effect index $i : e \in es$ using the *lookupPT* helper function.

```
lookupPT : ∀ {C R es} (pts : PTs es) (i : eff C R ∈ es) → (c : C) → (R c → Set) → Set
lookupPT (pt :: pts) ∈Head = PT.pt pt
lookupPT (pt :: pts) (∈Tail i) = lookupPT pts i
```

This results in the following definition of *wpFree* for combinations of effects.

```
wpFree : ∀ {a es} (pts : PTs es) →
  Free es a → (a → Set) → Set
wpFree pts (Pure x) P = P x
wpFree pts (Step i c k) P = lookupPT pts i c (λ x → wpFree pts (k x) P)
```

In the new definition of *match*, we want to combine the effects of nondeterminism and general recursion. To verify this definition, we need to give its semantics, for which we need to give the list of predicate transformers to *wpFree*. For nondeterminism we alread have the predicate transformer *ptAll*. However, it is not as easy to give a predicate transformer for general recursion, since the intended semantics of a recursive call depend on the function that is being called, i.e. the function that is being defined.

However, if we have a specification of a function of type $(i : I) \to O\ i$, for example in terms of a relation of type $(i : I) \to O\ i \to Set$, we can define a predicate transformer:

```
ptRec : ∀ {I : Set} {O : I → Set} → ((i : I) → O i → Set) → PT (ERec I O)
PT.pt (ptRec R) i P = ∀ o → R i o → P o
PT.mono (ptRec R) c P P′ imp asm o h = imp _ (asm _ h)
```

For example, the *Match* relation serves as a specification for the *match* function. If we use *ptRec R* as a predicate transformer to check that a recursive function satisfies the relation R, then we are proving *partial correctness*, since we assume each recursive call terminates according to the relation R.

4 Recursively parsing every regular expression

Now we are able to handle the Kleene star:

```
allSplits : {{ iND : ENondet ∈ es }} (xs : List a) → Free es (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 : {{ iND : ENondet ∈ es }} → Regex  String ↦[es] λ {(r , xs) → Tree r}
match (Empty , xs) = fail
```

```
match (Epsilon , Nil) = Pure tt
match (Epsilon , xs@ (_ :: _)) = 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) = do
  (ys , zs) ← allSplits xs
  y ← call (l , ys)
  z ← call (r , zs)
  Pure (y , z)
match ((l | r) , xs) = choice
  (Inl <$> call (l , xs))
  (Inr <$> call (r , xs))
match ((r ⋆) , Nil) = Pure Nil
match ((r ⋆) , xs@ (_ :: _)) = do
  (y , ys) ← call ((r · (r ⋆)) , xs)
  Pure (y :: ys)
```

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 makes use of *match*.

```
matchSpec : (r, xs : Pair Regex String) → Tree (Pair.fst r, xs) → Set
matchSpec (r , xs) ms = Match r xs ms

wpMatch : Free (eff (Pair Regex String) (λ {(r , xs) → Tree r}) :: ENondet :: Nil) a →
  (a → Set) → Set
wpMatch = wpFree (ptRec matchSpec :: ptAll :: Nil)
```

In a few places, we use a recursive *call* instead of actual recursion. One advantage to this choice is that in proving correctness, we can use the specification of *match* directly, without having to use the following rule of *consequence* in between. Unfortunately, we still need *consequence* to deal with the call to *allSplits*.

```
consequence : ∀ {a b es P} pts (mx : Free es a) (f : a → Free es b) →
  wpFree pts mx (λ x → wpFree pts (f x) P) == wpFree pts (mx >>= f) P
consequence pts (Pure x) f = refl
consequence pts (Step i c k) f = cong (lookupPT pts i c) (extensionality λ x → consequence pts (k x) f)

wpToBind : ∀ {a b es pts P} (mx : Free es a) (f : a → Free es b) →
  wpFree pts mx (λ x → wpFree pts (f x) P) → wpFree pts (mx >>= f) P
wpToBind {pts = pts} mx f H = subst id (consequence pts mx f) H

wpFromBind : ∀ {a b es pts P} (mx : Free es a) (f : a → Free es b) →
  wpFree pts (mx >>= f) P → wpFree pts mx (λ x → wpFree pts (f x) P)
wpFromBind {pts = pts} mx f H = subst id (sym (consequence pts mx f)) H
```

We can reuse exactly the same proof to show *allSplits* is correct, since we use the same semantics for the effects in *allSplits*. On the other hand, the correctness

proof for *match* needs a bit of tweaking to deal with the difference in the recursive steps.

```
matchSound : ∀ r, xs → wpSpec [ ⊤ , matchSpec r, xs ] ⊑ wpMatch (match r, xs)
matchSound (Empty , xs) P (preH , postH) = tt
matchSound (Epsilon , Nil)        P (preH , postH) = postH _ Epsilon
matchSound (Epsilon , (_ :: _)) P (preH , postH) = tt
matchSound (Singleton c , Nil)             P (preH , postH) = tt
matchSound (Singleton c , (x :: Nil))      P (preH , postH) with c ≟ x
matchSound (Singleton c , (.c :: Nil))     P (preH , postH) | yes refl = postH _ Singleton
matchSound (Singleton c , (x :: Nil))      P (preH , postH) | no ¬p = tt
matchSound (Singleton c , (_ :: _ :: _)) P (preH , postH) = tt
matchSound ((l · r) , xs) P (preH , postH) = wpToBind (allSplits xs) _
  (allSplitsSound xs _ (_ , (λ {(ys , zs) splitH y lH z rH → postH (y , z)
  (coerce (cong (λ xs → Match _ xs _) (sym splitH)) (Concat lH rH))})))
matchSound ((l | r) , xs) P (preH , postH) =
  (λ o H → postH _ (OrLeft H)) ,
  (λ o H → postH _ (OrRight H))
```

Now we are able to prove correctness of *match* on a Kleene star.

```
matchSound ((r ⋆) , Nil)        P (preH , postH) = postH _ StarNil
matchSound ((r ⋆) , (x :: xs)) P (preH , postH) = λ o H → postH _ (StarConcat H)
```

At this point, we have defined a parser for regular languages and formally proved that its output is always correct. However, *match* 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 results in the first nondeterministic alternative being an infinitely deep recursion.

The next step is then to write a parser that always terminates and show that *match* is refined by it. Our approach is to do recursion on the input string instead of on the regular expression.

5 Termination, using derivatives

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 make use of the Brzozowski derivative.

Theorem 2 ([Brz64]) *The* Brzozowski derivative *of a formal language L with respect to a character x consists of all strings xs such that $x :: xs \in L$.*

Importantly, 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 $x :: xs$ 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 let x be the head of the string and xs' the tail and go in recursion on matching $d\ r\ /d\ x$ with xs'.

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.

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

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

$$\begin{aligned}
&d_/d_\ :\ Regex\ \to\ Char\ \to\ Regex\\
&d\ Empty\ /d\ c\ =\ Empty\\
&d\ Epsilon\ /d\ c\ =\ Empty\\
&d\ Singleton\ x\ /d\ c\ \mathbf{with}\ c\ \stackrel{?}{=}\ x\\
&...\ |\ yes\ p\ =\ Epsilon\\
&...\ |\ no\ \neg p\ =\ Empty\\
&d\ l\cdot r\ /d\ c\ \mathbf{with}\ \varepsilon?\ l\\
&...\ |\ yes\ p\ =\ ((d\ l\ /d\ c)\cdot r)\ |\ (d\ r\ /d\ c)\\
&...\ |\ no\ \neg p\ =\ (d\ l\ /d\ c)\cdot r\\
&d\ l\ |\ r\ /d\ c\ =\ (d\ l\ /d\ c)\ |\ (d\ r\ /d\ c)\\
&d\ r\ \star\ /d\ c\ =\ (d\ r\ /d\ c)\cdot(r\ \star)
\end{aligned}$$

In order 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. We do this with the function *integralTree*:

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

We can also define it with exactly the same case distinction as we used to define $d\_/d\_$.

The code for the parser, *dmatch*, itself is very short. As we sketched, for an empty string we check that the expression matches the empty string, while for a non-empty string we use the derivative to perform a recursive call.

$$\begin{aligned}
&dmatch\ :\ \{\!\{\ iND\ :\ ENondet\ \in\ es\ \}\!\}\ \to\ Regex\ \ String\ \overset{es}{\rightharpoonup}\ \lambda\{(r\ ,\ xs)\ \to\ \mathsf{Tree}\ r\}\\
&dmatch\ (r\ ,\ Nil)\ \mathbf{with}\ \varepsilon?\ r\\
&...\ |\ yes\ (ms\ ,\ _)\ =\ Pure\ ms\\
&...\ |\ no\ \neg p\ \ \ \ \ \ \ \ =\ fail\\
&dmatch\ (r\ ,\ (x\ ::\ xs))\ =\ integralTree\ r\ <\$>\ call\ ((d\ r\ /d\ x)\ ,\ xs)
\end{aligned}$$

Since *dmatch* always consumes a character before going in recursion, we can easily prove that each recursive call only leads to finitely many other calls. This means that for each input value we can unfold the recursive step in the definition a bounded number of times and get a computation with no recursion. Intuitively, this means that *dmatch* terminates on all input. If we want to make

this more formal, we need to consider what it means to have no recursion in the computation. A definition for the *while* loop using general recursion looks as follows:

$$\begin{aligned}
&\mathit{while} : \forall \{\, es\ a \,\} \to \{\!\{\, \mathit{iRec} : \mathit{ERec}\ a\ (K\ a) \in es \,\}\!\} \to (a \to \mathit{Bool}) \to (a \to a) \to (a \to \mathit{Free}\ es\ a) \\
&\mathit{while}\ \mathit{cond}\ \mathit{body}\ i = \mathbf{if}\ \mathit{cond}\ i\ \mathbf{then}\ \mathit{Pure}\ i\ \mathbf{else}\ (\mathit{call}\ (\mathit{body}\ i))
\end{aligned}$$

We would like to say that some *while* loops terminate, yet the definition of *while* always contains a *call* in it. Thus, the requirement should not be that there are no more calls left, but that these calls are irrelevant. A next intuitive idea could be the following: if we make the recursive computation into a *Partial* computation by *fail*ing instead of *call*ing, the *Partial* computation still works the same, i.e. it refines, the recursive computation. However, this mixes the concepts of correctness and termination. We want to see that the computation terminates with some output, without caring about which output this is. Thus, we should only have a trivial postcondition $\lambda\ \_ \to \top$. We formalize this idea in the $\mathit{terminates-in}$ predicate.

$$\begin{aligned}
&\mathit{terminates-in} : (\mathit{pts} : \mathit{PTs}\ es)\ (f : C \overset{es}{\nrightarrow} R)\ (S : \mathit{Free}\ (\mathit{eff}\ C\ R :: es)\ a) \to \mathbb{N} \to \mathit{Set} \\
&\mathit{terminates-in}\ \mathit{pts}\ f\ (\mathit{Pure}\ x)\ n = \top \\
&\mathit{terminates-in}\ \mathit{pts}\ f\ (\mathit{Step} \in \mathsf{Head}\ c\ k)\ \mathit{Zero} = \bot \\
&\mathit{terminates-in}\ \mathit{pts}\ f\ (\mathit{Step} \in \mathsf{Head}\ c\ k)\ (\mathit{Succ}\ n) = \mathit{terminates-in}\ \mathit{pts}\ f\ (f\ c \gg\!= k)\ n \\
&\mathit{terminates-in}\ \mathit{pts}\ f\ (\mathit{Step}\ (\in \mathsf{Tail}\ i)\ c\ k)\ n = \mathit{lookupPT}\ \mathit{pts}\ i\ c\ (\lambda\ x \to \mathit{terminates-in}\ \mathit{pts}\ f\ (k\ x)\ n)
\end{aligned}$$

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. The proof goes by induction on this string. Unfolding the recursive *call* gives $(\mathit{dmatch}\ (d\ r\ /d\ x\ ,\ xs) \gg\!= (\mathit{Pure} \circ \mathit{integralTree})$, which we can rewrite in the lemma $\mathit{terminates-fmap}$ using the associativity monad law.

$$\begin{aligned}
&\mathit{dmatchTerminates} : (r : \mathit{Regex})\ (xs : \mathit{String}) \to \\
&\quad \mathit{terminates-in}\ (\mathit{ptAll} :: \mathit{Nil})\ (\mathit{dmatch})\ (\mathit{dmatch}\ (r\ ,\ xs))\ (\mathit{length}\ xs) \\
&\mathit{dmatchTerminates}\ r\ \mathit{Nil}\ \mathbf{with}\ \varepsilon?\ r \\
&\mathit{dmatchTerminates}\ r\ \mathit{Nil}\ |\ \mathit{yes}\ p = \mathit{tt} \\
&\mathit{dmatchTerminates}\ r\ \mathit{Nil}\ |\ \mathit{no}\ \neg p = \mathit{tt} \\
&\mathit{dmatchTerminates}\ r\ (x :: xs) = \mathit{terminates-fmap}\ (\mathit{length}\ xs) \\
&\quad (\mathit{dmatch}\ ((d\ r\ /d\ x)\ ,\ xs)) \\
&\quad (\mathit{dmatchTerminates}\ (d\ r\ /d\ x)\ xs)
\end{aligned}$$

To show partial correctness of *dmatch*, we can use the transitivity of the refinement relation. If we apply transitivity, it suffices to show that *dmatch* is a refinement of *match*. Our 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 $x :: xs$.

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

Since the definition of $d\_/d\_$ uses the *integralTree* function, we also prove the correctness of *integralTree*.

$$integralTreeCorrect\ :\ \forall\ r\ x\ xs\ y\ \rightarrow\ Match\ (d\ r\ /d\ x)\ xs\ y\ \rightarrow\ Match\ r\ (x\ ::\ xs)\ (integralTree\ r\ y)$$

All three proofs mirror the definitions of these functions, being structured as a case distinction on the regular expression.

Before we can prove the correctness of *dmatch* in terms of *match*, it turns out that we also need to describe *match* itself better. To show *match* is refined by *dmatch*, we need to prove that the output of *dmatch* is a subset of that of *match*. Since *match* makes use of *allSplits*, we first prove that *allSplits* returns all possible splittings of a string.

$$\begin{aligned}&allSplitsComplete\ :\ (xs\ ys\ zs\ :\ String)\ (P\ :\ String\ \ String\ \rightarrow\ Set)\ \rightarrow\\&\quad wpMatch\ (allSplits\ xs)\ P\ \rightarrow\ (xs\ ==\ ys\ +\!\!+\ zs)\ \rightarrow\ P\ (ys\ ,\ zs)\end{aligned}$$

The proof mirrors *allSplits*, performing induction on *xs*. Note that *allSplitsSound* and *allSplitsComplete* together show that *allSplits xs* is equivalent to its specification $[\ \top\ ,\ \lambda\ \{(ys\ ,\ zs)\ \rightarrow\ xs\ ==\ ys\ +\ zs\}\ ]$, in the sense of the $\_\equiv\_$ relation.

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

$$dmatchSound\ :\ \forall\ r\ xs\ \rightarrow\ wpMatch\ (match\ (r\ ,\ xs))\ \sqsubseteq\ wpMatch\ (dmatch\ (r\ ,\ xs))$$

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. Therefore, we omit the proof.

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: since *dmatch* never makes a nondeterministic choice, it will not return all possible parse trees as specified by *Match*, only the first tree that it encounters. Still, we can express the property that *dmatch* finds a parse tree if it exists. In other words, we will show that if there is a valid parse tree, *dmatch* returns any parse tree (and this is a valid tree by *dmatchSound*). To express that *dmatch* returns something, we use a trivially true postcondition, and replace the demonic choice of the *ptAll* semantics with the angelic choice of *ptAny*:

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

The proof is short, since *dmatch* can only *fail* when it encounters an empty string and a regex that does not match the empty string, contradicting the assumption immediately:

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

Here we have demonstrated 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.

Note the correspondences of this section with 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 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 *EParser* 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.

6 Effects as unifying theory of parsers

In the previous section, we have developed a formally verified parser for regular languages. The class of regular languages is small, and does not include most programming languages. If we want to write a parser for a larger class of languages, we fist need a practical representation. In classical logic, the most general concept of a formal language is no more than a set of strings, or a predicate over strings, represented by the type $String \to Set$. Constructively, such predicates (even when decidable) are not very useful: the language $\{xs \mid xs$ is a valid proof of the Riemann Hypothesis$\}$ has no defined cardinality. To make them more amenable to reasoning, we impose structure on languages, for example by giving their definition as a regular expression. When we have a more structured grammar, we can write a parser for these grammars, and prove its partial correctness and termination, just as we did for regular expressions and *dmatch*.

One structure we can impose on languages is that we can always perform local operations, in the style of the Brzozowski derivative. This means we can decide whether a language l matches the empty string (as $\varepsilon?$ does for regular languages), and for each character x, we can compute the derivative $d\ l\ /d\ x$, which contains exactly those xs such that $x\ ::\ xs$ is in l. Packaging up these two operations into a record type gives the *coinductive trie* representation of a formal language, as described by Abel [Abe16]. We augment the definition by including a list of the parser's output values for the empty string, instead of a Boolean stating whether the language contains the empty string. An empty list corresponds to the original *False*, while a non-empty list corresponds to *True*.

```
record Trie (i : Size) (a : Set) : Set where
  coinductive
  field
    ε? : List a
    d_/d_ : Char → Trie j a
```

The definition of the *Trie* type is complicated by making it coinductive and using sized types. We need *Trie* to be coinductive since it appears in a negative position in the $d\_/d\_$ operator, or viewed in another way, since the *Trie* type needs to be nested arbitrarily deeply to describe arbitrarily long strings. The sized types help Agda to check that certain definitions terminate. Despite being needed to ensure the *Trie* type is useful, the two complications do not play an important role in the remainder of the development.

Example 1. Let us look at two simple examples of definitions using the *Trie* type. The first definition, *emptyTrie*, represents the empty language. It does not contain the empty string, so $\varepsilon?$ *emptyTrie* is the empty list. It also does not contain any string of the form $x\ ::\ xs$, so the derivatives of the empty trie are all the empty trie again.

```
emptyTrie : Trie i a
ε? emptyTrie = Nil
d emptyTrie /d x = emptyTrie
```

This is also an example of why we need the coinductive structure of the *Trie* type, since the definition $d\ emptyTrie\ /d\ x\ =\ emptyTrie$ is not productive for an inductive type.

The second example of a construction in the *Trie* type is the union operator, which is straightforward to write out.

```
_∪_ : Trie i a → Trie i a → Trie i a
ε? (t ∪ t′) = ε? t ++ ε? t′
d (t ∪ t′) /d x = (d t /d x) ∪ (d t′ /d x)
```

We can also take a very computational approach to languages, representing them by a parser. This parser takes a string and returns a list of successful matches, similar to the $\varepsilon?$ operator of the coinductive *Trie*.

```
Parser : Set → Set
Parser a = String → List a
```

6.1 Context-free grammars with *Productions*

Using a *Trie* or a *Parser* to define a language requires a lot of low-level work, since we first need to implement operations such as the union of a language or concatenation. The *Regex* representation of regular languages has such operations built-in, allowing us to have intuition on the level of grammar rather than operations. A class of languages that is more expressive than the regular languages, while remaining tractable in parsing is that of the *context-free language*. The expressiveness of context-free languages is enough to cover most programming languages used in practice [AU77]. We will represent context-free languages in Agda by giving a grammar in the style of Brink, Holdermans, and Löh [BHL10], in a similar way as we represent a regular language using an element of the *Regex* type. Following their development, we parametrize our definitions over a collection of nonterminal symbols.

```
record GrammarSymbols : Set where
  field
    Nonterminal : Set
    ⟦_⟧ : Nonterminal → Set
    _≟_ : Decidable {A = Nonterminal} _==_
```

The elements of the type *Char* are the *terminal* symbols, for example characters or tokens. The elements of the type *Nonterminal* are the *nonterminal* symbols, representing the language constructs. As for *Char*, we also need to be able to decide the equality of nonterminals. The (disjoint) union of *Char* and *Nonterminal* gives all the symbols that we can use in defining the grammar.

```
Symbol = Either Char Nonterminal
Symbols = List Symbol
```

For each nonterminal A, our goal is to parse a string into a value of type $[\![A]\!]$, based on a set of production rules. A production rule $A \to xs$ gives a way to expand the nonterminal A into a list of symbols xs, such that successfully matching each symbol of xs with parts of a string gives a match of the string with A. Since matching a nonterminal symbol B with a (part of a) string results in a value of type $[\![B]\!]$, a production rule for A is associated with a *semantic function* that takes all values arising from submatches and returns a value of type $[\![A]\!]$, as expressed by the following type:

```
⟦_∣_⟧ : Symbols → Nonterminal → Set
⟦ Nil         ∥ A ⟧ = ⟦ A ⟧
⟦ Inl x  :: xs ∥ A ⟧ = ⟦ xs ∥ A ⟧
⟦ Inr B  :: xs ∥ A ⟧ = ⟦ B ⟧ → ⟦ xs ∥ A ⟧
```

Now we can define the type of production rules. A rule of the form $A \to BcD$ is represented as $prod\ A\ (Inr\ B\ ::\ Inl\ c\ ::\ Inr\ D\ ::\ Nil)\ f$ for some f.

```
record Production : Set where
  constructor prod
  field
    lhs : Nonterminal
    rhs : Symbols
    sem : ⟦ rhs ‖ lhs ⟧
```

We use the abbreviation *Productions* to represent a list of productions, and a grammar will consist of the list of all relevant productions.

7 Parsing as effect

While we can follow the traditional development of parsers from nondeterministic state, algebraic effects allow us to introduce new effects, which saves us bookkeeping issues. For a description of parsing based on algebraic effects, we introduce a new effect *EParser*, and use a state consisting of a *String*. The *EParser* effect has one command *Parse*, which either returns the current character in the state (advancing it to the next character) or fails if all characters have been consumed. In our current development, we do not need more commands such as an *EOF* command which succeeds only if all characters have been consumed, so we do not incorporate them. However, in the semantics we will define that parsing was successful if the input string has been completely consumed.

```
data CParser : Set where
  Parse : CParser
RParser : CParser → Set
RParser Parse = Char
EParser = eff CParser RParser
parse : {{ iP : EParser ∈ es }} → Free es Char
parse {{ iP }} = Step iP Parse Pure
```

Note that *EParse* is not sufficient alone to implement even simple parsers such as *dmatch*: we need to be able to choose between parsing the next character or returning a value for the empty string. This is why we usually combine *EParser* with the nondeterminism effect *ENondet*. However, nondeterminism and parsing is not always enough: we also need general recursion to deal with productions of the form $E \to E$.

The denotational semantics of a parser in the *Free* monad are given by handling the effects. We give two functions, one returning a *Parser* and one returning a *Trie*.

```
toParser : Free (ENondet :: EParser :: Nil) a → Parser a
toTrie : Free (ENondet :: EParser :: Nil) a → Trie a
```

```
toParser (Pure x) Nil = x :: Nil
toParser (Pure x) (_ :: _) = Nil
toParser (Step ∈Head Fail k) xs = Nil
toParser (Step ∈Head Choice k) xs = toParser (k True) xs ++ toParser (k False) xs
toParser (Step (∈Tail ∈Head) Parse k) Nil = Nil
toParser (Step (∈Tail ∈Head) Parse k) (x :: xs) = toParser (k x) xs

toTrie (Pure x) = record {ε? = x :: Nil; d_/d_ = λ _ → emptyTrie}
toTrie (Step ∈Head Fail k) = emptyTrie
toTrie (Step ∈Head Choice k) = toTrie (k True) ∪ toTrie (k False)
toTrie (Step (∈Tail ∈Head) Parse k) = record {ε? = Nil; d_/d_ = λ x → toTrie (k x)}
```

If we prefer to look at the semantics of parsing as a proposition instead of a function, we can use predicate transformers. Since the *Parse* effect uses a state consisting of the string to be parsed, the predicates depend on this state. We modify the definition of *wp* so each *Effect* can access its own state.

```
record PT^S (s : Set) (e : Effect) : Set where
  constructor mkPTS
  field
    pt : (c : Effect.C e) → (Effect.R e c → s → Set) → s → Set
    mono : ∀ c P Q → (∀ x t → P x t → Q x t) → ∀ t → pt c P t → pt c Q t

data PTs^S (s : Set) : List Effect → Set where
  Nil : PTs^S s Nil
  _::_ : ∀ {e es} → PT^S s e → PTs^S s es → PTs^S s (e :: es)

lookupPTS : ∀ {s C R es} (pts : PTs^S s es) (i : eff C R ∈ es) → (c : C) → (R c → s → Set) →
lookupPTS (pt :: pts) ∈Head c P t = PTS.pt pt c P t
lookupPTS (pt :: pts) (∈Tail i) c P t = lookupPTS pts i c P t

lookupMono : ∀ {s C R es} (pts : PTs^S s es) (i : eff C R ∈ es) → ∀ c P P′ → (∀ x t → P x t →
lookupMono (pt :: pts) ∈Head = PTS.mono pt
lookupMono (pt :: pts) (∈Tail i) = lookupMono pts i

wp : ∀ {s es a} → (pts : PTs^S s es) → Free es a → (a → s → Set) → s → Set
wp pts (Pure x) P = P x
wp pts (Step i c k) P = lookupPTS pts i c (λ x → wp pts (k x) P)
```

To give the predicate transformer semantics of the *EParser* effect, we need to choose the meaning of failure, for the case where the next character is needed and all characters have already been consumed. Since we want all results returned by the parser to be correct, we use demonic choice and the *ptAll* predicate transformer as the semantics for *ENondet*. Using *ptAll*'s semantics for the *Fail* command gives the following semantics for the *EParser* effect.

```
ptParse : PT^S String EParser
PTS.pt ptParse Parse P Nil = ⊤
PTS.pt ptParse Parse P (x :: xs) = P x xs
```

Example 2. With the predicate transformer semantics of *EParse*, we can define the language accepted by a parser in the *Free* monad as a predicate over strings: a string *xs* is in the language of a parser S if the postcondition "all characters have been consumed" is satisfied.

```
empty? : List a → Set
empty? Nil = ⊤
empty? (_ :: _) = ⊥
_∈[_] : String → Free (ENondet :: EParser :: Nil) a → Set
xs ∈[ S ] = wp (ptAll :: ptParse :: Nil) S (λ _ → empty?) xs
```

8 From abstract grammars to abstract parsers

We want to show that the effects *EParser* and *ENondet* are sufficient to parse any context-free grammar, using a generally recursive function. To show this claim, we implement a function *fromProductions* that constructs a parser for any context-free grammar given as a list of *Production*s, then formally verify the correctness of *fromProductions*. Our implementation mirrors the definition of the *generateParser* function by Brink, Holdermans, and Löh, differing in the naming and in the system that the parser is written in: our implementation uses the *Free* monad and algebraic effects, while Brink, Holdermans, and Löh use a monad *Parser* that is based on parser combinators.

We start by defining two auxiliary types, used as abbreviations in our code.

```
FreeParser = Free (eff Nonterminal ⟦_⟧ :: ENondet :: EParser :: Nil)
record ProductionRHS (A : Nonterminal) : Set where
  constructor prodrhs
  field
    rhs : Symbols
    sem : ⟦ rhs ‖ A ⟧
```

The core algorithm for parsing a context-free grammar consists of the following functions, calling each other in mutual recursion:

```
fromProductions : (A : Nonterminal) → FreeParser ⟦ A ⟧
filterLHS       : (A : Nonterminal) → Productions → List (ProductionRHS A)
fromProduction  : ProductionRHS A → FreeParser ⟦ A ⟧
buildParser     : (xs : Symbols) → FreeParser (⟦ xs ‖ A ⟧ → ⟦ A ⟧)
exact           : a → Char → FreeParser a
```

The main function is *fromProductions*: given a nonterminal, it selects the productions with this nonterminal on the left hand side using *filterLHS*, and makes a nondeterministic choice between the productions.

```
filterLHS A Nil = Nil
filterLHS A (prod lhs rhs sem :: ps) with A ≟ lhs
```

```
... | yes refl  =  prodrhs rhs sem  ::  filterLHS A ps
... | no _     =  filterLHS A ps
fromProductions A  =  foldr (choice) (fail) (map fromProduction (filterLHS A prods))
```

The function *fromProduction* takes a single production and tries to parse the input string using this production. It then uses the semantic function of the production to give the resulting value.

```
fromProduction (prodrhs rhs sem)  =  buildParser rhs  >>=  λ f  →  Pure (f sem)
```

The function *buildParser* iterates over the *Symbols*, calling *exact* for each literal character symbol, and making a recursive *call* to *fromProductions* for each nonterminal symbol.

```
buildParser Nil  =  Pure id
buildParser (Inl x   ::  xs)  =  exact tt x  >>=  λ _  →  buildParser xs
buildParser (Inr B  ::  xs)  =  call B  >>=  (λ x  →  buildParser xs  >>=  λ o  →  Pure λ f  →  o (f x))
```

Finally, *exact* uses the *parse* command to check that the next character in the string is as expected, and *fail*s if this is not the case.

```
exact x t  =
  parse  >>=  λ t′  →
  if t  ≟  t′ then Pure x else fail
```

9 Partial correctness of the parser

Partial correctness of the parser is relatively simple to show, as soon as we have a specification. Since we want to prove that *fromProductions* correctly parses any given context free grammar given as an element of *Productions*, the specification consists of a relation between many sets: the production rules, an input string, a nonterminal, the output of the parser, and the remaining unparsed string. Due to the many arguments, the notation is unfortunately somewhat unwieldy. To make it a bit easier to read, we define two relations in mutual recursion, one for all productions of a nonterminal, and for matching a string with a single production rule.

```
data _⊢_∈⟦_⟧⇒_,_ prods where
  Produce :  prod lhs rhs sem  ∈  prods  →
             prods  ⊢  xs  ∼  rhs  ⇒  f , ys  →
             prods  ⊢  xs  ∈⟦ lhs ⟧⇒  f sem , ys
data _⊢_∼_⇒_,_ prods where
  Done :     prods  ⊢  xs  ∼  Nil  ⇒  id , xs
  Next :     prods  ⊢  xs  ∼  ps  ⇒  o , ys  →
             prods  ⊢  (x  ::  xs)  ∼  (Inl x  ::  ps)  ⇒  o , ys
  Call :     prods  ⊢  xs  ∈⟦ A ⟧⇒  o , ys  →
```

$$
\begin{array}{l}
\mathit{prods} \vdash \mathit{ys} \sim \mathit{ps} \Rightarrow f\ ,\ \mathit{zs} \to \\
\mathit{prods} \vdash \mathit{xs} \sim (\mathit{Inr}\ A :: \mathit{ps}) \Rightarrow (\lambda\ g \to f\ (g\ o))\ ,\ \mathit{zs}
\end{array}
$$

With these relations, we can define the specification *parserSpec* to be equal to $\_ \vdash \_ \in [\![\_]\!] \Rightarrow \_,\_$ (up to reordering some arguments), and show that *fromProductions* refines this specification. For the semantics of general recursion, we also make use of the specification, while for the semantics of nondeterminism, we use the *ptAll* semantics to ensure all output is correct. This gives the partial correctness term as defined below

$$
\begin{array}{l}
\mathit{pts}\ \mathit{prods} = \mathit{ptRec}\ (\mathit{parserSpec}\ \mathit{prods}) :: \mathit{ptAll} :: \mathit{ptParse} :: \mathit{Nil} \\
\mathit{wpFromProd}\ \mathit{prods} = \mathit{wp}\ (\mathit{pts}\ \mathit{prods}) \\
\mathit{partialCorrectness} : (\mathit{prods} : \mathit{Productions})\ (A : \mathit{Nonterminal}) \to \\
\quad \mathit{wpSpec}\ [\ \top\ ,\ (\mathit{parserSpec}\ \mathit{prods}\ A)\ ] \sqsubseteq \mathit{wpFromProd}\ \mathit{prods}\ (\mathit{fromProductions}\ \mathit{prods}\ A)
\end{array}
$$

Let us fix the production rules *prods*. How do we prove the partial correctness? Since the structure of *fromProductions* is of a nondeterministic choice between productions to be parsed, and we want to show that all alternatives for a choice result in success, we will first give a lemma expressing the correctness of each alternative. Correctness in this case is expressed by the semantics of a single production rule, i.e. the $\_\vdash\_\sim\_\Rightarrow\_,\_$ relation. Thus, we want to prove a lemma with a type as follows:

$$
\begin{array}{l}
\mathit{parseStep} : \forall\ A\ \mathit{xs}\ P\ \mathit{str} \to \\
\quad ((o : [\![\ \mathit{xs} \parallel A\ ]\!] \to [\![\ A\ ]\!])\ (\mathit{str}' : \mathit{String}) \to \mathit{prods} \vdash \mathit{str} \sim \mathit{xs} \Rightarrow o\ ,\ \mathit{str}' \to P\ o\ \mathit{str}') \to \\
\quad \mathit{wpFromProd}\ \mathit{prods}\ (\mathit{buildParser}\ \mathit{prods}\ \mathit{xs})\ P\ \mathit{str}
\end{array}
$$

The lemma can be proved by reproducing the case distinctions used to define *buildParser*; there is no complication apart from having to use the *wpToBind* lemma to deal with the $\_ \gg\!= \_$ operator in a few places.

$$
\begin{array}{l}
\mathit{parseStep}\ A\ \mathit{Nil}\ P\ t\ H = H\ \mathit{id}\ t\ \mathit{Done} \\
\mathit{parseStep}\ A\ (\mathit{Inl}\ x :: \mathit{xs})\ P\ \mathit{Nil}\ H = \mathit{tt} \\
\mathit{parseStep}\ A\ (\mathit{Inl}\ x :: \mathit{xs})\ P\ (x' :: t)\ H\ \mathbf{with}\ x \stackrel{?}{=} x' \\
\ldots \mid \mathit{yes}\ \mathit{refl} = \mathit{parseStep}\ A\ \mathit{xs}\ P\ t\ \lambda\ o\ t'\ H' \to H\ o\ t'\ (\mathit{Next}\ H') \\
\ldots \mid \mathit{no}\ \neg p = \mathit{tt} \\
\mathit{parseStep}\ A\ (\mathit{Inr}\ B :: \mathit{xs})\ P\ t\ H\ o\ t'\ \mathit{Ho} = \\
\quad \mathit{wpToBind}\ (\mathit{buildParser}\ \mathit{prods}\ \mathit{xs})\ \_\ \_ \\
\qquad (\mathit{parseStep}\ A\ \mathit{xs}\ \_\ t'\ \lambda\ o'\ \mathit{str}'\ \mathit{Ho}' \to H\ \_\ \_\ (\mathit{Call}\ \mathit{Ho}\ \mathit{Ho}'))
\end{array}
$$

To combine the *parseStep* for each of the productions in the nondeterministic choice, it is tempting to define another lemma *filterStep* by induction on the list of productions. But we must be careful that the productions that are used in the *parseStep* are the full list *prods*, not the sublist *prods′* used in the induction step. Additionally, we must also make sure that *prods′* is indeed a sublist, since using an incorrect production rule in the *parseStep* will result in an invalid result. Thus, we parametrise *filterStep* by a list *prods′* and a proof that it is a sublist of

prods. Again, the proof uses the same distinction as *fromProductions* does, and uses the *wpToBind* lemma to deal with the _≫=_ operator.

```
filterStep : ∀ prods′ → (p ∈ prods′ → p ∈ prods) →
  ∀ A → wpSpec [ ⊤ , parserSpec prods A ] ⊑ wpFromProd prods
    (foldr (choice) (fail) (map (fromProduction prods) (filterLHS prods A prods′)))
filterStep Nil ⊆ A P xs H = tt
filterStep (prod lhs rhs sem :: prods′) ⊆ A P xs H with A ≟ lhs
filterStep (prod .A rhs sem :: prods′) ⊆ A P xs (_ , H) | yes refl
  = wpToBind (buildParser prods rhs) _ _
  (parseStep A rhs _ xs λ o t′ H′ → H _ _ (Produce (⊆ ∈Head) H′))
  , filterStep prods′ (⊆ ∘ ∈Tail) A P xs (_ , H)
... | no ¬p = filterStep prods′ (⊆ ∘ ∈Tail) A P xs H
```

With these lemmas, *partialCorrectness* just consists of applying *filterStep* to the subset of *prods* consisting of *prods* itself.

10 Termination of the parser

To show termination we need a somewhat more subtle argument: since we are able to call the same nonterminal repeatedly, termination cannot be shown simply by inspecting the definitions. Consider the grammar given by $E \to aE; E \to b$, where we see that the string that matches E in the recursive case is shorter than the original string, but the definition itself is of unbounded length. Fortunately for us, predicate transformer semantics allow us to give this more subtle definition of termination, in the form of the *Termination* type in Definition **??**. By taking into account the current state, i.e. the string to be parsed, in the variant, we can show that a decreasing string length leads to termination.

But not all grammars feature this decreasing string length in the recursive case, with the most pathological case being those of the form $E \to E$. The issues do not only occur in edge cases: the grammar $E \to E + E; E \to 1$ representing very simple expressions will already result in non-termination for *fromProductions* as it will go in recursion on the first non-terminal without advancing the input string. Since the position in the string and current nonterminal together fully determine the state of *fromParsers*, it will not terminate. We need to ensure that the grammars passed to the parser do not allow for such loops.

Intuitively, the condition on the grammars should be that they are not *left-recursive*, since in that case, the parser should always advance its position in the string before it encounters the same nonterminal. This means that the number of recursive calls to *fromProductions* is bounded by the length of the string times the number of different nonterminals occurring in the production rules. The type we will use to describe the predicate "there is no left recursion" is constructively somewhat stronger: we define a left-recursion chain from A to B to be a sequence of nonterminals $A, \ldots, A_i, A_{i+1}, \ldots, B$, such that for each adjacent pair A_i, A_{i+1} in the chain, there is a production of the form $A_{i+1} \to B_1 B_2 \ldots B_n A_i \ldots$, where $B_1 \ldots B_n$ are all nonterminals. In other words, we can advance the parser to A

starting in B without consuming a character. Disallowing (unbounded) left recursion is not a limitation for our parsers: Brink, Holdermans, and Löh [BHL10] have shown that the *left-corner transform* can transform left-recursive grammars into an equivalent grammar without left recursion. Moreover, they have implemented this transform, including formal verification, in Agda. In this work, we assume that the left-corner transform has already been applied if needed, so that there is an upper bound on the length of left-recursive chains in the grammar.

We formalize one link of this left-recursive chain in the type *LeftRec*, while a list of such links forms the *LeftRecChain* data type.

```
record LeftRec (prods : Productions) (A B : Nonterminal) : Set where
  field
    rec : prod A (map Inr xs ++ (Inr B :: ys)) sem ∈ prods
```

(We leave *xs*, *ys* and *sem* as implicit fields of *LeftRec*, since they are fixed by the type of *rec*.)

```
data LeftRecChain (prods : Productions) : Nonterminal → Nonterminal → Set where
  Nil : LeftRecChain prods A A
  _::_ : LeftRec prods B A → LeftRecChain prods A C → LeftRecChain prods B C
```

Now we say that a set of productions has no left recursion if all such chains have an upper bound on their length.

```
chainLength : LeftRecChain prods A B → ℕ
chainLength Nil = 0
chainLength (c :: cs) = Succ (chainLength cs)

leftRecBound : Productions → ℕ → Set
leftRecBound prods n = (cs : LeftRecChain prods A B) → chainLength cs < n
```

If we have this bound on left recursion, we are able to prove termination, since each call to *fromProductions* will be made either after we have consumed an extra character, or it is a left-recursive step, of which there is an upper bound on the sequence. Thus, the relation *RecOrder* will work as a recursive variant for *fromProductions*:

```
data RecOrder (prods : Productions) : (x y : Nonterminal String) → Set where
  Adv : length str < length str' → RecOrder prods (A , str) (B , str')
  Rec : length str ≤ length str' → LeftRec prods A B → RecOrder prods (A , str) (B , str')
```

Goede plaats hiervoor? The petrol-driven semantics are based on a syntactic argument: we know a computation terminates because expanding the call tree will eventually result in no more *call*s. Termination can also be defined based on a well-foundedness argument, such as the size-change principle of Agda's termination checker. Thus, we want to say that a recursive definition is well-founded if all recursive calls are made to a smaller argument, according to a well-founded relation.

Theorem 3 ([Acz77]) *In intuitionistic type theory, we say that a relation* $\_\prec\_$ *on a type* a *is well-founded if all elements* $x : a$ *are* accessible*, which is defined by (well-founded) recursion to be the case if all elements in the downset of* x *are accessible.*

$$\begin{array}{l}\textbf{data}\ \mathit{Acc}\ (\_\prec\_ : a \to a \to \mathit{Set}) : a \to \mathit{Set}\ \textbf{where}\\ \quad \mathit{acc} : (\forall\ y \to y \prec x \to \mathit{Acc}\ \_\prec\_\ y) \to \mathit{Acc}\ \_\prec\_\ x\end{array}$$

To see that this is equivalent to the definition of well-foundedness in set theory, recall that a relation $\_\prec\_$ on a set a is well-founded if and only if there is a monotone function from a to a well-founded order. Since all inductive data types are well-founded, and the termination checker ensures that the argument to *acc* is a monotone function, there is a function from $x : a$ to $\mathit{Acc}\ \_\prec\_\ x$ if and only if $\_\prec\_$ is a well-founded relation in the set-theoretic sense.

The condition that all calls are made to a smaller argument is related to the notion of a loop *variant* in imperative languages. While an invariant is a predicate that is true at the start and end of each looping step, the variant is a relation that holds between successive looping steps.

Theorem 4 *Given a recursive definition* $f : \cdot \dot{\pitchfork} \cdot\ I\ es\ O$*, a relation* $\_\prec\_$ *on* C *is a recursive* variant *if for each argument* c*, and each recursive call made to* c' *in the evaluation of* $f\ c$*, we have* $c' \prec c$*. Formally:*

$$\begin{array}{l}\mathit{variant}' : (\mathit{pts} : \mathsf{PTs}^S\ s\ (\mathit{eff}\ C\ R :: es))\ (f : C \overset{es}{\pitchfork} R)\ (\_\prec\_ : (C\ \ s) \to (C\ \ s) \to \mathit{Set})\\ \quad (c : C)\ (t : s)\ (S : \mathit{Free}\ (\mathit{eff}\ C\ R :: es)\ a) \to s \to \mathit{Set}\\ \mathit{variant}'\ \mathit{pts}\ f\ \_\prec\_\ c\ t\ (\mathit{Pure}\ x)\ t' = \top\\ \mathit{variant}'\ \mathit{pts}\ f\ \_\prec\_\ c\ t\ (\mathit{Step} \in \mathsf{Head}\ c'\ k)\ t'\\ \quad = ((c' , t') \prec (c , t))\ \ \mathit{lookupPTS}\ \mathit{pts} \in \mathsf{Head}\ c'\ (\lambda\ x \to \mathit{variant}'\ \mathit{pts}\ f\ \_\prec\_\ c\ t\ (k\ x))\ t'\\ \mathit{variant}'\ \mathit{pts}\ f\ \_\prec\_\ c\ t\ (\mathit{Step}\ (\in \mathsf{Tail}\ i)\ c'\ k)\ t'\\ \quad = \mathit{lookupPTS}\ \mathit{pts}\ (\in \mathsf{Tail}\ i)\ c'\ (\lambda\ x \to \mathit{variant}'\ \mathit{pts}\ f\ \_\prec\_\ c\ t\ (k\ x))\ t'\end{array}$$

$$\begin{array}{l}\mathit{variant} : (\mathit{pts} : \mathsf{PTs}^S\ s\ (\mathit{eff}\ C\ R :: es))\ (f : C \overset{es}{\pitchfork} R) \to ((C\ \ s) \to (C\ \ s) \to \mathit{Set}) \to \mathit{Set}\\ \mathit{variant}\,\mathit{pts}\ f\ \_\prec\_ = \forall\ c\ t \to \mathit{variant}'\ \mathit{pts}\ f\ \_\prec\_\ c\ t\ (f\ c)\ t\end{array}$$

Note that *variant* depends on the semantics *pt* we give to f. We cannot derive the semantics in *variant* from the structure of f, since we do not yet know whether f terminates. Using *variant*, we can define another termination condition:

Theorem 5 *A recursive definition* f *is* well-founded *if it has a variant that is well-founded.*

$$\begin{array}{l}\textbf{record}\ \mathit{Termination}\ (\mathit{pts} : \mathsf{PTs}^S\ s\ (\mathit{eff}\ C\ R :: es))\ (f : C \overset{es}{\pitchfork} R) : \mathit{Set}\ \textbf{where}\\ \quad \textbf{field}\\ \quad\quad \_\prec\_ : (C\ \ s) \to (C\ \ s) \to \mathit{Set}\\ \quad\quad w - f : \forall\ c\ t \to \mathit{Acc}\ \_\prec\_\ (c , t)\\ \quad\quad \mathit{var} : \mathit{variant}\ \mathit{pts}\ f\ \_\prec\_\end{array}$$

A generally recursive function that terminates in the petrol-driven semantics is also well-founded, since a variant is given by the well-order _<_ on the amount of fuel consumed by each call. The converse also holds: if we have a descending chain of calls *cs* after calling f with argument c, we can use induction on the type $Acc\ \_\prec\_\ c$ to bound the length of *cs*. This bound gives the amount of fuel consumed by evaluating a call to f on c.

With the definition of *RecOrder*, we can complete the correctness proof of *fromProductions*, by giving an element of the corresponding *Termination* type. We assume that the length of recursion is bounded by $bound : \mathbb{N}$.

```
fromProductionsTerminates : (prods : Productions) (bound : ℕ) → leftRecBound prods bound →
  Termination (pts prods) (fromProductions prods)
Termination.___ (fromProductionsTerminates prods bound H) = RecOrder prods
```

To show that the relation *RecOrder* is well-founded, we need to show that there is no infinite descending chain starting from some nonterminal A and string *str*. The proof is based on iteration on two natural numbers n and k, which form an upper bound on the number of allowed left-recursive calls in sequence and unconsumed characters in the string respectively. Note that the number *bound* is an upper bound for n and the length of the input string is an upper bound for k. Since each nonterminal in the production will decrease n and each terminal will decrease k, we eventually reach the base case *0* for either. If n is zero, we have made more than *bound* left-recursive calls, contradicting the assumption that we have bounded left recursion. If k is zero, we have consumed more than *length str* characters of *str*, also a contradiction.

```
Termination.w-f (fromProductionsTerminates prods bound H) A str
  = acc (go A str (length str) ≤-refl bound Nil ≤-refl)
  where
  go : ∀ A str →
    (k : ℕ) → length str ≤ k →
    (n : ℕ) (cs : LeftRecChain prods A B) → bound ≤ chainLength cs + n →
    ∀ y → RecOrder prods y (A , str) → Acc (RecOrder prods) y
  go A Nil Zero ltK n cs H′ (A′ , str′) (Adv ())
  go A (_ :: _) Zero () n cs H′ (A′ , str′) (Adv lt)
  go A (_ :: _) (Succ k) (s≤s ltK) n cs H′ (A′ , str′) (Adv (s≤s lt))
    = acc (go A′ str′ k (≤-trans lt ltK) bound Nil ≤-refl)
  go A str k ltK Zero cs H′ (A′ , str′) (Rec lt cs′)
    = magic (<→≱ (H cs) (≤-trans H′ (≤-reflexive (+-zero _))))
  go A str k ltK (Succ n) cs H′ (A′ , str′) (Rec lt c)
    = acc (go A′ str′ k (≤-trans lt ltK) n (c :: cs) (≤-trans H′ (≤-reflexive (+ - suc _ _))))
```

To show that *RecOrder* is a variant for *fromProductions*, we cannot follow the definitions of *fromProductions* as closely as we did for the partial correctness proof. We need a complicated case distinction to keep track of the left-recursive chain we have followed in the proof. For this reason, we split the *parseStep* apart

into two lemmas *parseStepAdv* and *parseStepRec*, both showing that *buildParser* maintains the variant. We also use a *filterStep* that calls the correct *parseStep* for each production in the nondeterministic choice.

```
parseStepAdv : ∀ A xs str str' → length str' < length str →
  variant' (pts prods) (fromProductions prods) (RecOrder prods) A str (buildParser xs) str'
parseStepRec : ∀ A xs str str' → length str' ≤ length str →
  ∀ ys → prod A (map Inr ys ++ xs) sem ∈ prods →
  variant' (pts prods) (fromProductions prods) (RecOrder prods) A str (buildParser xs) str'
filterStep : ∀ prods' → (x ∈ prods' → x ∈ prods) →
  ∀ A str str' → length str' ≤ length str →
  variant' (pts prods) (fromProductions prods) (RecOrder prods) A str
    (foldr (choice) (fail) (map fromProduction (filterLHS A prods')))
  str'
```

In the *parseStepAdv*, we deal with the situation that the parser has already consumed at least one character since it was called. This means we can repeatedly use the *Adv* constructor of *RecOrder* to show the variant holds.

```
parseStepAdv A Nil str str' lt = tt
parseStepAdv A (Inl x :: xs) str Nil lt = tt
parseStepAdv A (Inl x :: xs) str (c :: str') lt with x ≟ c
parseStepAdv A (Inl x :: xs) (_ :: _ :: str) (.x :: str') (s≤s (s≤s lt)) | yes refl
  = parseStepAdv A xs _ _ (s≤s (≤-step lt))
... | no ¬p = tt
parseStepAdv A (Inr B :: xs) str str' lt
  = Adv lt
  , λ o str'' H → variant−fmap (pts prods) (fromProductions prods) (buildParser xs)
    (parseStepAdv A xs str str'' (≤-trans (s≤s (consumeString str' str'' B o H)) lt))
```

Here, the lemma *variant − fmap* states that the variant holds for a program of the form $S \gg\!= (Pure \circ f)$ if it does for S, since the *Pure* part does not make any recursive calls; the lemma *consumeString str' str'' B* states that the string *str''* is shorter than *str'* if *str''* is the left-over string after matching *str''* with nonterminal B.

In the *parseStepRec*, we deal with the situation that the parser has only encountered nonterminals in the current production. This means that we can use the *Rec* constructor of *RecOrder* to show the variant holds until we consume a character, after which we call *parseStepAdv* to finish the proof.

```
parseStepRec A Nil str str' lt ys i = tt
parseStepRec A (Inl x :: xs) str Nil lt ys i = tt
parseStepRec A (Inl x :: xs) str (c :: str') lt ys i with x ≟ c
parseStepRec A (Inl x :: xs) (_ :: str) (.x :: str') (s≤s lt) ys i | yes refl
  = parseStepAdv A xs _ _ (s≤s lt)
... | no ¬p = tt
parseStepRec A (Inr B :: xs) str str' lt ys i
```

```
  = Rec lt (record {rec = i})
  , λ o str'' H → variant − fmap (pts prods) (fromProductions prods) (buildParser xs)
    (parseStepRec A xs str str'' (≤-trans (consumeString str' str'' B o H) lt)
    (ys ++ (B :: Nil)) (nextNonterminal i))
```

Apart from the previous lemmas, we make use of *nextNonterminal* i, which states that the current production starts with the nonterminals $ys \mathbin{+\!\!+} (B :: Nil)$.

The lemma *filterStep* shows that the variant holds on all subsets of the production rules, analogously to the *filterStep* of the partial correctness proof. It calls *parseStepRec* since the parser only starts consuming characters after it selects a production rule.

```
filterStep Nil A str str' lt ⊆ = tt
filterStep (prod lhs rhs sem :: prods') ⊆ A str str' lt with A ≟ lhs
... | yes refl
  = variant − fmap (pts prods) (fromProductions prods) (buildParser rhs)
    (parseStepRec A rhs str str' lt Nil (⊆ ∘ ∈Head))
    , filterStep prods' (⊆ ∘ ∈Tail) A str str' lt
... | no ¬p = filterStep prods' (⊆ ∘ ∈Tail) A str str' lt
```

As for partial correctness, the main proposition consists of applying *filterStep* to the subset of *prods* consisting of *prods* itself.

Having divided the proof into the three lemmas, the remainder is straightforward. The proofs of the lemmas use induction on the production rule for *parseStepAdv* and *parseStepRec*, and induction on the list of rules for *filterStep*, and call each other as indicated.