Combining predicate transformer semantics for effects

A case study in parsing regular languages

Anne Baanen
Wouter Swierstra

Vrije Universiteit Amsterdam
Utrecht University

Algebraic effects

Algebraic effects separate the syntax and semantics of effects.

- The syntax describes the sequencing of the primitive operations
- The semantics assigns meaning to these operations

In this work, we use a free monad to model effectful programs in Agda:

```
data Free (C : Set) (R : C -> Set) : Set -> Set where
    Pure : a -> Free C R a
    Op : (c : C) -> (k : R c -> Free C R a) -> Free C R a
```

Example: Nondeterminism

Nondet has two primitive operations:

- Choice chooses between two values
- Fail goes to a failure state and stops execution

```
data CNondet where
        Choice : CNondet
        Fail : CNondet

RNondet : CNondet -> Set
RNondet Choice = Bool
RNondet Fail = ⊥

Nondet = Free CNondet RNondet
```

Semantics for algebraic effects

Handlers give semantics for the `Free` monad naturally as a fold:

```
handleList : Nondet a -> List a
handleList (Pure x) = [x]
handleList (Op Choice k) = k True ++ k False
handleList (Op Fail k) = []
```

Semantics for algebraic effects

Handlers give semantics for the Free monad naturally as a fold:

```
handleList : Nondet a -> List a
handleList (Pure x) = [x]
handleList (Op Choice k) = k True ++ k False
handleList (Op Fail k) = []
```

The generic fold that computes a predicate of type Set:

```
[[_]] : Free C R a -> ((c : C) -> (R c -> Set) -> Set)
      -> (a -> Set) -> Set
[[ Pure x ]] alg P = P x
[[ Op c k ]] alg P = alg c (λ x -> [[ k x ]] alg P)
```

Predicate transformer semantics

A predicate transformer for commands C and responses R is a function from postconditions of type `R -> Set` to preconditions of type `C -> Set`.
If R depends on C, this becomes:

```
pt C R = (c : C) -> (R c -> Set) -> Set
```

The type of the algebra passed to `[[_]]` is exactly `pt C R`. We have assigned *predicate transformer semantics* to algebraic effects.

Predicate transformer semantics for `Nondet`

For nondeterminism, there are two canonical choices of predicate transformer semantics.

`ptAll` requires that all potential results satisfy the postcondition:

```
ptAll Fail k = ⊤
ptAll Choice k = k True ∧ k False
```

`ptAny` requires that there is at least one outcome that satisfies the postcondition:

```
ptAny Fail k = ⊥
ptAny Choice k = k True ∨ k False
```

Parsing regular expressions

To illustrate these semantics, we wrote a parser. The input is a regular expression and a `String`, and the output a parse tree.

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

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)
```

Parsing regular expressions

We implement `match` as a case distinction.

```
match : (r : Regex) -> String -> Nondet (Tree r)
match Empty          xs       = Op Fail λ()
```

Parsing regular expressions

We implement `match` as a case distinction.

```
match : (r : Regex) -> String -> Nondet (Tree r)
match Empty          xs       = Op Fail λ()
match Epsilon        Nil      = Pure tt
match Epsilon        (_ :: _) = Op Fail λ()
```

Parsing regular expressions

We implement `match` as a case distinction.

```
match : (r : Regex) -> String -> Nondet (Tree r)
match Empty          xs        = Op Fail λ()
match Epsilon        Nil       = Pure tt
match Epsilon        (_ :: _) = Op Fail λ()
match (Singleton c) xs        =
    if xs = [c] then Pure c else Op Fail λ()
```

Parsing regular expressions

We implement `match` as a case distinction.

```
match : (r : Regex) -> String -> Nondet (Tree r)
match Empty          xs       = Op Fail λ()
match Epsilon        Nil      = Pure tt
match Epsilon        (_ :: _) = Op Fail λ()
match (Singleton c) xs        =
    if xs = [c] then Pure c else Op Fail λ()
match (l | r)        xs       = Op Choice (λ b ->
    if b then Inl <$> match l xs else Inr <$> match r xs)
```

We implement `match` as a case distinction.

```
match : (r : Regex) -> String -> Nondet (Tree r)
match Empty          xs       = Op Fail λ()
match Epsilon        Nil      = Pure tt
match Epsilon        (_ :: _) = Op Fail λ()
match (Singleton c) xs       =
    if xs = [c] then Pure c else Op Fail λ()
match (l | r)        xs       = Op Choice (λ b ->
    if b then Inl <$> match l xs else Inr <$> match r xs)
match (l · r) xs = do
    (ys, zs) <- allSplits xs
    (,) <$> match l ys <*> match r zs
```

Parsing regular expressions

We implement `match` as a case distinction.

```
match : (r : Regex) -> String -> Nondet (Tree r)
match Empty          xs       = Op Fail λ()
match Epsilon        Nil      = Pure tt
match Epsilon        (_ :: _) = Op Fail λ()
match (Singleton c) xs        =
    if xs = [c] then Pure c else Op Fail λ()
match (l | r)        xs       = Op Choice (λ b ->
    if b then Inl <$> match l xs else Inr <$> match r xs)
match (l · r) xs = do
    (ys, zs) <- allSplits xs
    (,) <$> match l ys <*> match r zs
match (r *) xs = match (Epsilon | r · (r *)) xs
```

We implement `match` as a case distinction.

```
match : (r : Regex) -> String -> Nondet (Tree r)
match Empty          xs        = Op Fail λ()
match Epsilon        Nil       = Pure tt
match Epsilon        (_ :: _)  = Op Fail λ()
match (Singleton c)  xs        =
    if xs = [c] then Pure c else Op Fail λ()
match (l | r)        xs        = Op Choice (λ b ->
    if b then Inl <$> match l xs else Inr <$> match r xs)
match (l · r) xs = do
    (ys, zs) <- allSplits xs
    (,) <$> match l ys <*> match r zs
match (r *) xs = match (Epsilon | r · (r *)) xs
```

Error: `match (r *) xs` does not terminate

Parsing regular expressions

For now, we will write:

```
match (r *) xs = Op Fail λ()
```

Parsing regular expressions

For now, we will write:

```
match (r *) xs = Op Fail λ()
```

To verify our implementation, we take a specification consisting of precondition and postcondition:

```
pre : Regex -> String -> Set
pre r xs = hasNo* r

post : (r : Regex) -> String -> Tree r -> Set
post r xs t = Match r xs t
```

And check that `match` *refines* this specification.

Refinement calculus

A predicate transformer `pt1` *is refined by* `pt2` if `pt2` satisfies more postconditions than `pt1`:

```
_⊑_ : (pt1 pt2 : (a -> Set) -> Set) -> Set
pt1 ⊑ pt2 = ∀ P -> pt1 P -> pt2 P
```

S ⊑ T expresses that T is “better” than S: S can be replaced with T everywhere, and all postconditions will still hold.

Refinement calculus

A predicate transformer `pt1` *is refined by* `pt2` if `pt2` satisfies more postconditions than `pt1`:

```
_⊑_ : (pt1 pt2 : (a -> Set) -> Set) -> Set
pt1 ⊑ pt2 = ∀ P -> pt1 P -> pt2 P
```

S ⊑ T expresses that T is “better” than S: S can be replaced with T everywhere, and all postconditions will still hold.

Predicate transformers are a semantic domain where programs and specifications can be related.

```
[[_,_]] : (pre : Set) (post : a -> Set) -> (a -> Set) -> Set
[[ pre , post ]] P = pre ∧ ∀ x, post x -> P x
```

With these ingredients, the correctness statement of `match` becomes:

```
matchSound : (r : Regex) (xs : String) ->
    [[ pre r xs , post r xs ]] ⊑ [[ match r xs ]] ptAll
```

The proof proceeds by case distinction and is uncomplicated, until we need to reason about the monadic bind operator _>>=_.

Verification

With these ingredients, the correctness statement of `match` becomes:

```
matchSound : (r : Regex) (xs : String) ->
    [[ pre r xs , post r xs ]] ⊑ [[ match r xs ]] ptAll
```

The proof proceeds by case distinction and is uncomplicated, until we need to reason about the monadic bind operator `_>>=_`.

The missing ingredient is the rule of consequence:

```
consequence : ∀ pt (S : Free es a) (f : a -> Free es b) ->
    [[ S >>= f ]] pt P ≡ [[ S ]] pt (λ x -> [[ f x ]] pt P)
```

Adding effects

The problem with `match` is that implementing the Kleene star also requires the effect of *general recursion*.

We can add more effects to the free monad by choosing the command and response types from a list of *effect signatures*:

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

We will add two new effects: general recursion and parsing.

Adding effects

Inspired by McBride's *Turing-Completeness Totally Free*, we use the `Rec I O` effect to represent a recursive function of type `(i : I) -> O i` calling itself. The commands are the arguments to the function and the responses are the returned values.

```
Rec : (I : Set) (O : I -> Set) -> Sig
Rec I O = mkSig I O
```

To specify the semantics of `Rec`, we need an invariant of type `(i : I) -> O i -> Set`, specifying which values of type `O i` can be returned from a call with argument `i : I`.

```
ptRec inv i P = ∀ o -> inv i o -> P o
```

Adding effects

The `Parser` effect represents a stateful parser with one command: advance the input string by one character.

```
Parser : Sig
Parser = mkSig T (λ _ -> Maybe Char)
```

`Parser` has stateful semantics: to return the next character, we need to keep track of the remaining characters. The state is the extra `String` arguments in `ptParser`.

```
ptParser : (Maybe Char -> String -> Set) -> String -> Set
ptParser P Nil = P Nothing Nil
ptParser P (x :: xs) = P (Just x) xs
```

Extending `match`

Now we can finish the definition and prove soundness unconditionally:

```
match (r *) = Op iRec (Epsilon | r · (r *))

matchSound : (r : Regex) (xs : String) ->
    [[ ⊤ , post r xs ]] ⊑ [[ match r xs ]]
```

Extending `match`

Now we can finish the definition and prove soundness unconditionally:

```
match (r *) = Op iRec (Epsilon | r · (r *))

matchSound : (r : Regex) (xs : String) ->
    [[ ⊤ , post r xs ]] ⊑ [[ match r xs ]]
```

`match` still does not terminate if `r` matches the empty string, our result is only *partial correctness*.
`ptRec` computes the WLP: all recursive calls immediately return.

Defining a derivative-based matcher

To guarantee termination, use recursion on `xs` rather than `r`.
The *Brzozowski derivative* `d r /d x` matches `xs` iff `r` matches `x :: xs`.

```
dmatch : (r : Regex) -> Free es (Tree r)
dmatch r = Op iParse
    λ { (Just x) -> Op iRec (d r /d x) (integralTree r)
        Nothing  -> if p <- matchEpsilon r
                    then Pure (Sigma.fst p)
                    else Op iND Fail λ() }
```

`integralTree r : tree (d r /d x) -> tree r` "integrates" parse trees.

Defining a derivative-based matcher

To guarantee termination, use recursion on `xs` rather than `r`.
The *Brzozowski derivative* `d r /d x` matches `xs` iff `r` matches `x :: xs`.

```
dmatch : (r : Regex) -> Free es (Tree r)
dmatch r = Op iParse
    λ { (Just x) -> Op iRec (d r /d x) (integralTree r)
        Nothing  -> if p <- matchEpsilon r
                    then Pure (Sigma.fst p)
                    else Op iND Fail λ() }
```

`integralTree r : tree (d r /d x) -> tree r` "integrates" parse trees.

```
dmatchSound : ∀ r xs -> [[ match r xs ]] ⊑ [[ dmatch r xs ]]
```

Termination checking

`ptRec` gives weakest liberal precondition semantics. For total correctness, we should check termination.

`terminates-in f S n` holds iff S terminates after calling f at most n times.

```
terminates-in : (f : (i : I) -> Free (Rec I O :: es) (O i))
    (S : Free (Rec I O :: es) a) → ℕ → Set
terminates-in f (Pure x)           n        = ⊤
terminates-in f (Op ∈Head c k)     Zero     = ⊥
terminates-in f (Op ∈Head c k)     (Succ n) =
    terminates-in pt f (f c >>= k) n
terminates-in f (Op (∈Tail i) c k) n        =
    pts i c (λ x -> terminates-in f (k x) n)
```

Total correctness

Partial correctness of `dmatch` follows from the chain of refinements:

```
[[ ⊤ , post r xs ]]
    ⊑ [[ match r xs ]]
    ⊑ [[ dmatch r xs ]]
    ⊑ [[ ⊤ , post r xs ]]
```

together with a proof of termination:

```
dmatchTerminates : (r : Regex) (xs : String) ->
    terminates-in dmatch (dmatch r xs) (length xs)
```

Discussion

In our paper, we illustrate how techniques from the refinement calculus can be used in functional programming. They provide a natural and uniform way to reason about effects in the setting of the `Free` monad.

A distinguishing characteristic of our approach is modularity: we add new effects and semantics to the system as we need them.

Formally verified parsers have been developed before, using specialized semantics to the domain of parsing. The modularity of predicate transformers allow us to reason about effects uniformly.

Most existing approaches to recursion in parsers deal with termination syntactically. Separation of syntax and semantics also cleanly separates partial and total correctness.