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

Anne Baanen Wouter Swierstra

September 1, 2020

The history

Predicate transformer semantics for imperative programs have been around for 45 years. A statement becomes a map from postcondition (a predicate on the state space) to the weakest precondition that ensures the postcondition will be satisfied.

The history

Predicate transformer semantics for imperative programs have been around for 45 years. A statement becomes a map from postcondition (a predicate on the state space) to the weakest precondition that ensures the postcondition will be satisfied.
Parser combinators have been around in functional programming for 35 years. Parsing becomes an effectful (monadic) operation, nondeterministically modifying state (the unparsed part of the string).

The history

Predicate transformer semantics for imperative programs have been around for 45 years. A statement becomes a map from postcondition (a predicate on the state space) to the weakest precondition that ensures the postcondition will be satisfied.
Parser combinators have been around in functional programming for 35 years. Parsing becomes an effectful (monadic) operation, nondeterministically modifying state (the unparsed part of the string).
Algebraic effects have recently gained traction in the programming language community.

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

```
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

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

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 Fail k) = []
handleList (Op Choice k) = k True ++ k False
```

The generic fold that computes a predicate of type Set:

```
[[_]] : Free C R a -> ((c : C) -> (R c -> 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

Consider the following syntax of regular expressions and their parse trees:

```
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)
```

Let's write a parser returning these `Tree` s.

Parsing regular expressions

A nondeterministic Regex matcher is straightforward to write:

```
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

A nondeterministic Regex matcher is straightforward to write:

```
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
```

... until we reach the last case:

```
match (r *) xs = match (r · (r *)) xs
    -- Error: 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 pt_2 satisfies more postconditions than pt_1:

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

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

By assigning a predicate transformer to specifications, we can also relate specifications and programs:

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

This is the ‘weakest precondition’ necessary so that the desired post-condition P holds: pre should hold and any result satisfying post should imply the postcondition P.

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 _>>=_.

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 ]] pt (λ x → [[ f x ]] pt P) ≡ [[ mx >>= f ]] pt P
```

Adding effects

The problem with `match` is that implementing the Kleene star 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
```

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

Adding effects

The `Rec I O` effect represents 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 ⊤ (λ _ -> Maybe Char)
```

The semantics of Parser are stateful: returning the next character of the input string requires keeping track of the remaining characters. This state can be found in the ptParser semantics as an extra argument to the predicates.

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

Defining a derivative-based matcher

```
dmatch : (Forall(es)) □ iP : Parser ∈ es □ □ iND : Nondet ∈ es □
dmatch r = symbol >>= maybe
    (λ x -> integralTree r <$> call (hiddenInstance(∈Head)) (d r
    (if p <- matchEpsilon r then Pure (Sigma.fst p) else (hiddenC
```

Termination checking

```
terminates-in : (Forall(I O es a)) (pts : PTs es) (f : (RecArr I e
terminates-in pts f (Pure x)              n          = ⊤
terminates-in pts f (Op ∈Head c k)        Zero       = ⊥
terminates-in pts f (Op ∈Head c k)        (Succ n)   = terminates-in
terminates-in pts f (Op (∈Tail i) c k)     n         =
    lookupPT pts i c (λ x → terminates-in pts f (k x) n)
```

Refining `match` with `dmatch`

```
dmatchSound : ∀ r xs -> (wpMatch (match (hiddenInstance(∈Head)) (r
```