Advanced Functional Programming

2012-2013, periode 2

Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

December 11, 2012

10. Advanced parser Combinators

10.1 Problems with “List of Successes”

Recap: parser Combinators

The naïve implementation of parser combinators uses the list-of-successes method, which is a combination of a state mondad and a list monad:

```
(<*>) :: Parser (b → a) → Parser b → Parser a
p <*> q = λinp → [(b2a b, qrest) | (b2a, prest) ← p inp
                                  , (b, qrest)    ← q prest
                 ]
(<||>) :: Parser a → Parser a → Parser a
p <||> q = λinp → p inp ++ q inp
```

Problems with Erroneous Input

- If your input does not conform to the language recognized by the parser all you get as a result is: $[\,]$.

Problems with Erroneous Input

- If your input does not conform to the language recognized by the parser all you get as a result is: [ ].
- It may take quite a while before you get this negative result, since the backtracking may try all other alternatives at all positions.

Problems with Erroneous Input

- If your input does not conform to the language recognized by the parser all you get as a result is: $[\,]$.
- It may take quite a while before you get this negative result, since the backtracking may try all other alternatives at all positions.
- There is no indication of where things went wrong.

Problems with Space Consumption

- A **complete result** has to be constructed before any part of it is returned
- The **complete input** is present in memory as long as no parse has been found
- Efficiency may depend critically on the ordering of the alternatives, and thus on how the grammar was written

For all of these problems we have found solutions.

10.2 History Parsers

Replace depth-first by breath-first

We introduce a Steps data type which contains a computed result (using a GADT and an existential type, to which we will come back later).

```
data Steps a where
       Step  ::        Progress → Steps a → Steps a
       Apply :: ∀a b.(b → a)  → Steps b → Steps a
       Fail  :: . . .
```

Replace depth-first by breath-first

We introduce a Steps data type which contains a computed result (using a GADT and an existential type, to which we will come back later).

```
data Steps a where
     Step  ::          Progress → Steps a → Steps a
     Apply :: ∀a b.(b → a)  → Steps b → Steps a
     Fail  :: . . .
```

The Progress field describes how much progress we made in the input (i.e. how much of the input was consumed by this step)

Computing a result

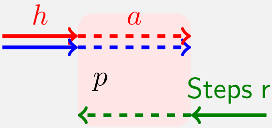
We compute a result on the fly, and change the parser type into a "continuation monad":

$$
\begin{aligned}
&\textbf{newtype}\ \mathsf{HP\ st\ a} \\
&\quad = \quad \mathsf{HP}\ (\forall \mathsf{r}.(\mathsf{a} \rightarrow \mathsf{st} \rightarrow \mathsf{Steps\ r}) \rightarrow \mathsf{st} \rightarrow \mathsf{Steps\ r})
\end{aligned}
$$

Computing a result

We compute a result on the fly, and change the parser type into a "continuation monad":

$$
\begin{array}{ll}
\textbf{newtype} & \text{HP st a} \\
\quad = & \text{HP } (\forall r.(a \to st \to \text{Steps } r) \to st \to \text{Steps } r)
\end{array}
$$



best

We define the function which compares two alternatives.

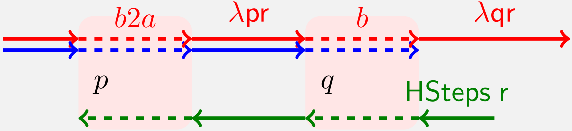
```
best′ :: Steps b → Steps b → Steps b
Fail ...  ‘best′‘  ...       = Fail ...
Fail ...  ‘best′‘  r         = r
l         ‘best′‘  Fail ...  = l
Step n l ‘best′‘ Step m r
    | n == m  = Step n (l ‘best′‘ r)
    | n < m   = Step n (l ‘best′‘  Step (m − n) r)
    | n > m   = Step m (Step (n − m) l ‘best′‘ r)
```

History parsers are Functor **and** Applicative

```
instance Functor (T st) where
   fmap f (HP ph)      = HP (λk → ph (k ∘ f))
instance Applicative (HP state) where
   HP ph <*> ~(HP qh)
               = HP (λk → ph (λpr → qh (λqr → k (pr qr))))
   pure a      = HP ($a)
instance Alternative (T state) where
   HP ph <|> HP qh = HP (λk inp → ph k inp ‘best‘ qh k inp)
   empty           = HP (λk inp → noAlts)
```

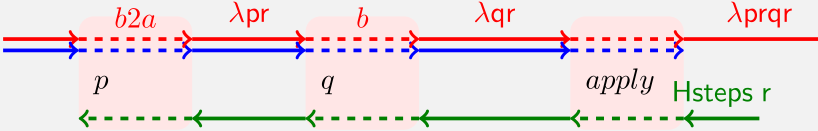
History parsers are Functor and Applicative

```
instance Functor (T st) where
   fmap f (HP ph)      = HP (λk → ph (k ∘ f))
instance Applicative (HP state) where
   HP ph <*>  ~(HP qh)
                  = HP (λk → ph (λpr → qh (λqr → k (pr qr))))
   pure a         = HP ($a)
instance Alternative (T state) where
   HP ph <|> HP qh = HP (λk inp → ph k inp `best` qh k inp)
   empty           = HP (λk inp → noAlts)
```



History parsers are Functor and Applicative

```
instance Functor (T st) where
   fmap f (HP ph)      = HP (λk → ph (k ∘ f))
instance Applicative (HP state) where
   HP ph <*>  ∼(HP qh)
              = HP (λk → ph (λpr → qh (λqr → k (pr qr))))
   pure a     = HP ($a)
instance Alternative (T state) where
   HP ph <|> HP qh = HP (λk inp → ph k inp `best` qh k inp)
   empty           = HP (λk inp → noAlts)
```



10.3 Online Result Construction

Online results

One of the problems which remains is that we only have access to the result once we have found a complete parse.

Online results

One of the problems which remains is that we only have access to the result once we have found a complete parse.

- for our just introduced parsers this is obvious

$$\begin{aligned} p \mathbin{<\!\!*\!\!>} q = \lambda \mathsf{inp} \rightarrow [\,(\mathsf{b2a}\ \mathsf{b}, \mathsf{rr}) \mid\ & (\mathsf{b2a}, \mathsf{prest}) \leftarrow \mathsf{p}\ \mathsf{inp} \\ ,\ & (\mathsf{b}, \quad \mathsf{qrest}) \leftarrow \mathsf{q}\ \mathsf{prest} \\ ]\ & \end{aligned}$$

We only get the first element of the list of results once q has found a match!

Online results

One of the problems which remains is that we only have access to the result once we have found a complete parse.

- for our just introduced parsers this is obvious
- but this also holds for the “list-of-successes” method; it is caused by the pattern-matching in the sequential composition

$$\begin{array}{llll}
\mathsf{p} \mathbin{<\!\!\ast\!\!>} \mathsf{q} = \lambda \mathsf{inp} \rightarrow [(\mathsf{b2a}\ \mathsf{b}, \mathsf{rr}) & \mid & (\mathsf{b2a}, \mathsf{prest}) & \leftarrow \mathsf{p}\ \mathsf{inp} \\
 & , & (\mathsf{b}, \quad \mathsf{qrest}) & \leftarrow \mathsf{q}\ \mathsf{prest} \\
]
\end{array}$$

We only get the first element of the list of results once q has found a match!

Change of Specification

In principle the non-online behaviour is correct: we ask for a complete result, and we can only get a result once we have found at least one complete parse!

Change of Specification

In principle the non-online behaviour is correct: we ask for a complete result, and we can only get a result once we have found at least one complete parse!

We observe that, while parsing according to our breadth-first stategy, once we have only **one living alternative left** we could just as well return the result corresponding to the recognised part!

Change of Specification

In principle the non-online behaviour is correct: we ask for a complete result, and we can only get a result once we have found at least one complete parse!

We observe that, while parsing according to our breadth-first stategy, once we have only **one living alternative left** we could just as well return the result corresponding to the recognised part!

This is especially useful if we incorporate error-correction in such a way that we are guaranteed to get at least one "possibly succesfully corrected" parse.

Future Based Parsers

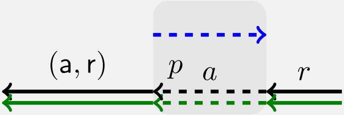
```
newtype FP st a = FP (∀r.(st → Steps      r) →
                          st → Steps (a, r)
                         )
```

We merge fragments of the result we are constructing with the progress information:

Future Based Parsers

```
newtype FP st a = FP (∀r.(st → Steps      r) →
                          st → Steps (a, r)
                         )
```

We merge fragments of the result we are constructing with the progress information:



best

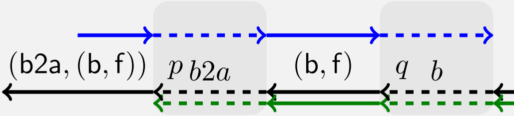
We have to make sure that if we compare two alternatives we have progress information at the head:

```
norm :: Steps a → Steps a
norm   (Apply f (Step   p l  )) =    Step p (Apply f l)
norm   (Apply f (Fail     ...)) =    Fail   ...
norm   (Apply f (Apply g l  )) =    norm (Apply (f ∘ g) l)
norm   steps                    =    steps
x ‘best‘ y = norm x ‘best′‘ norm y
best′ :: Steps b → Steps b → Steps b
```

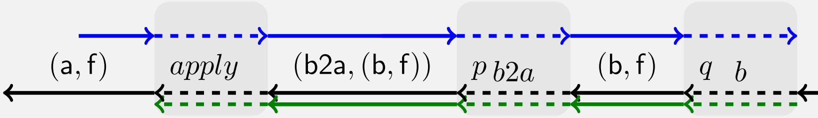
FP **is** Applicative

```
instance Applicative (FP state) where
  FP pf <*> ~(FP qf) = FP ((apply∘) ∘ (pf ∘ qf))
  pure a             = FP ((push a)∘)
instance Alternative (FP state) where
  FP pf <|> FP qf = FP (λk inp → pf k inp ‘best‘ qf k inp)
  empty           = FP (λk inp → noAlts)
```

Sequential composition for FParser



Sequential composition for FParser



FParser **is** ISParser

```
pSym a = FP (λk inp →
   case inp of (s : ss) → if s == a then addStep ◦ push s $ k ss
                                    else Fail . . .
               [ ]      → Fail . . .
               )
```

Helper code

```
eval :: Steps  r → r
eval (Step n  l ) = (eval l)
eval (Fail  ss ls) = . . .
eval (Apply f l ) = f (eval l)
```

```
push   :: v → Steps r → Steps (v, r)
push v = Apply (λr → (v, r))
```

```
apply :: Steps (b → a, (b, r)) → Steps (a, r)
apply = Apply (λ(b2a, ∼(b, r)) → (b2a b, r))
```

Helper code

```
eval :: Steps  r → r
eval (Step n  l ) = (eval l)
eval (Fail  ss ls) = . . .
eval (Apply f l ) = f (eval l)
```

```
push   :: v → Steps r → Steps (v, r)
push v = Apply (λr → (v, r))
```

```
apply :: Steps (b → a, (b, r)) → Steps (a, r)
apply = Apply (λ(b2a, ~(b, r)) → (b2a b, r))
```

Notice the ∼ in apply. This makes that the function can already produce something!

10.4 A Monadic Interface

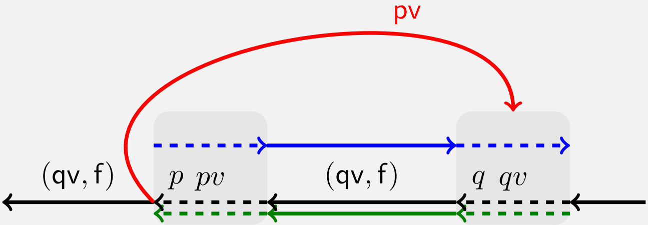
Monadic Interface: Parsing XML

Using a Monadic interface we can e.g. check an XML file for well balanced tags:

```
data XML = Tag t [XML]
pMany p  = (:) <$> p <*> pMany p <|> pSucceed []
pXML     = do t ← pOpenTag
              Tag t <$> pMany pXML <*  pCloseTag t
```

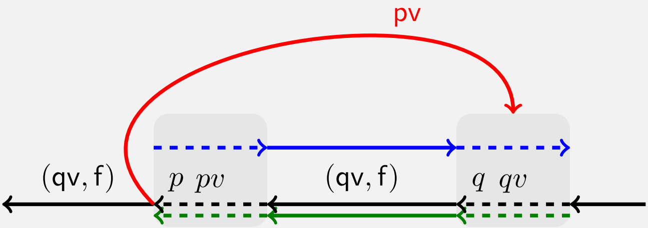
Our first attempt" FP

instance Monad FP s **where**
    p $\ggg$ q $= \lambda$k i $\rightarrow$ **let** steps $=$ p (q pv k) i
                                   $(pv, \_)$ $=$ eval steps
                               in Apply snd steps
    return v $= pSucceed$ v



Our first attempt” FP

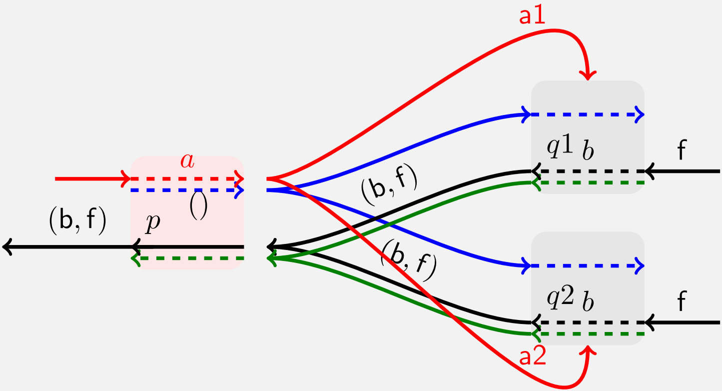
```
instance Monad FP s where
   p ≫= q = λk i → let steps   = p (q pv k) i
                       (pv, _) = eval steps
                   in Apply snd steps
   return v = pSucceed v
```



Unfortunately this is not correct. This may lead to a black hole, since the value *pv* may not be available yet in q, when needed in producing Steps. The function norm may push the value

Solution: Combining HP and FP

```
(≫=) :: HP st a → (a → FP st b) → FP st b
p ≫= q = FP (λk st → p (λpv st′ → q pv k st′) st)
```



Making the solution into a Monad

Our next kind of parser is a tupling between a history based and a future based parser:

```
data Parser s a = P (HP s a) (FP s a)
instance Applicative (Parser s) where
(P hp fp) <*> ~(P hq fq) = P (hp <*> hq)  (fp <*> fq)
(P hp fp) <|>  (P hq fq) = P (hp <|> hq)  (fp <|> fq)
pSucceed a               = P (pSucceed a) (pSucceed a)
pFail                    = P pFail        pFail
```

The Monadic Interface Code

```
instance Monad (Parser s) where
   (P (HP p) _) ≫= qq
       = P (HP (λk st → p (λa st′ → unHP (qq a) k st′) st))
           (FP (λk st → p (λa st′ → unFP (qq a) k st′) st))
         where unHP (P (HP h) _) = h
               unFP (P _ (FP f) ) = f
   return x = P (pSucceed x) (pSucceed x)
```

Note that from left hand side of the bind we always take the history based parser, whereas for the right hand side we have two cases to take care of.

Further optimisations

Once we have started to tuple various variants of parsers we might just as well:

Further optimisations

Once we have started to tuple various variants of parsers we might just as well:

- also tuple a pure recogniser, so we can avoid construction of results which will be discarded anyway

Further optimisations

Once we have started to tuple various variants of parsers we might just as well:

- also tuple a pure recogniser, so we can avoid construction of results which will be discarded anyway
- tuple a possibly empty parser, which is needed for an efficient implementation of the permutation parser with components that may be empty

Further optimisations

Once we have started to tuple various variants of parsers we might just as well:

- also tuple a pure recogniser, so we can avoid construction of results which will be discarded anyway
- tuple a possibly empty parser, which is needed for an efficient implementation of the permutation parser with components that may be empty
- a list of possible starter symbols to be used in error messages

Further optimisations

Once we have started to tuple various variants of parsers we might just as well:

- also tuple a pure recogniser, so we can avoid construction of results which will be discarded anyway
- tuple a possibly empty parser, which is needed for an efficient implementation of the permutation parser with components that may be empty
- a list of possible starter symbols to be used in error messages
- ...

10.5 Error Correction

Error correction

We can extend the system with an error correcting mechanism.

- we may delete a symbol, at a certain cost
- we may insert a symbol, at a certain cost
- the function best does not select the longest sequence of steps, but the cheapest
- limited look-ahead is needed in order to get fast parsers

The correction function $pSym$

We show a simplified error correcting parser:

```
data Steps result   = Shift  (Steps result)
                    | Fail   (Steps result)
                    | Done
pSym a =
  FP $ let  pSym'
          = λk input →
            case input of
            inp@(b : bs) → if a == b
                           then Step ∘ push b $ k bs
                           else Fail  ∘ push a $ k bs
                                           `best`
                                Fail (pSym' k bs)
            []           → Fail ∘ push a       $ k input
       in pSym'
```

Refinement of Error-correcting Process

1. We may associate a cost with each insertion of deletion step, so we can take the "cheapest future"; some symbols are unlikely to have been forgotten.
2. Limited look-ahead in order to speed-up correction process
3. Store a report about the corrections taken in the state
4. Collect a list of expected symbols, in order to generate nice error messages.
5. Use an abstract interpretation to find a non-recursive alternative, in order to avoid infinite insertions.

Computing the minimal length of an alternative

In each tuple which represents a parser we incorporate a value of type Nat:

```
data Nat = Zero
    | Succ Nat deriving Show
nat_min :: Nat → Nat → Int → (Nat, Bool)
nat_min _          Zero     _ = (Zero, False)
nat_min Zero       _        _ = (Zero, True)
nat_min l          Infinite _ = (l, True)
nat_min (Succ ll) (Succ rr) n
    = if n > 1000 then error "problem with comparing length
        else let (v, b) = nat_min ll rr (n + 1)
             in (Succ v, b))
nat_add Zero     r = r
nat_add (Succ l) r = Succ (nat_add l r))
```

The Actual Parser Types

```
data P st a
   = P (T st a)              -- HP, FP and recogniser
       (Maybe (T st a))      -- non-empty parsers
       Nat                   -- minimal length
       (Maybe a)             -- possibly empty
```

The Actual Parser Types

```
data P st a
   = P (T st a)            -- HP, FP and recogniser
       (Maybe (T st a))    -- non-empty parsers
       Nat                 -- minimal length
       (Maybe a)           -- possibly empty
```

And the parsing triple:

```
data T st a
   = T -- history
       (∀r.(a → st → Steps r) → st → Steps      r )
       -- future
       (∀r.(     st → Steps r) → st → Steps (a, r))
       -- recogniser
       (∀r.(     st → Steps r) → st → Steps      r )
```

Dealing with Fail

We have been a bit sloppy about failing parsers. We now give the full Fail-alternative of the Steps a type:

```
type Syms = [String]
data Steps a where
  Step  ::           Progress → Steps a                               → Steps a
  Apply :: ∀a b.(b → a)  → Steps   b                              → Steps a
  Fail  ::           Syms       → [Syms → (Int, Steps a)] → Steps a
```

The Strings field keeps track of symbols which were expected.

Dealing with Fail

We have been a bit sloppy about failing parsers. We now give the full Fail-alternative of the Steps a type:

```
type Syms = [String]
data Steps a where
   Step  ::          Progress → Steps a                  → Steps a
   Apply :: ∀a b.(b → a)   → Steps   b                   → Steps a
   Fail  ::          Syms     → [Syms → (Int, Steps a)] → Steps a
```

The Strings field keeps track of symbols which were expected. The are collected in the function best

```
best′ :: Steps b → Steps b → Steps b
Fail sl ll      ‘best′‘ Fail sr rr = Fail (sl ++ sr) (ll ++ rr)
Fail _ _        ‘best′‘ r          = r
l               ‘best′‘ Fail _ _   = l
```

Getting rid of Fail

In case a repair was really necessary the function eval will encounter a Fail in the list of steps:

1. all the expected symbols are apssed to all the alternatives
2. the resulting tree is examined upto a certain depth
3. the cheapest branch is taken

Getting rid of Fail

In case a repair was really necessary the function eval will encounter a Fail in the list of steps:

1. all the expected symbols are apssed to all the alternatives
2. the resulting tree is examined upto a certain depth
3. the cheapest branch is taken

```
eval  (Fail expected ls)
    = eval (getCheapest 3 (map ($expected) ls))
eval . . .
```