Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Merging Parsers

IFIP 2.1, Rome meeting

Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

Feb 7, 2012

1. History

Originally we had two libraries

The original uulib library had two modules:

1. permuting parsers
2. parsing merged lists

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Parsing permuted structures

Permuting structures are abundant:

```
@inProceedings
{ BaarsLoehSwierstra2001,
  author      =  {  Baars, Arthur and Loeh, Andres
                     and Swierstra, S. Doaitse},
  title       =  {  Parsing Permutation Phrases},
  booktitle   =  {  Preliminary proceedings of
                    Haskell workshop 2001,
                    UU-CS-2001-23},
  year        =  2001,
  pages       =  {171--182},
  editor      =  {Hinze, Ralf},
}
```

Permuted structures

- The order of the elements is irrelevant
- Each item occurs exactly once
- Elements may have **different types**
- Some elements are optional

Permuted structures

- The order of the elements is irrelevant
- Each item occurs exactly once
- Elements may have **different types**
- Some elements are optional

Traditional ways of parsing such structures are clumsy.

Merged lists

Many inputs consist of a couple of merged lists, which we want to process separately:

1. Haskell: priorities, data definitions, types, classes, instances, type specifications, normal definitions
2. AG system: data definitions, attribute introductions, semantic functions, Haskell fragments

Observation

If we restrict lists to length < 1, the parser for merged lists boils donw to a permutation parser.

Observation

If we restrict lists to length < 1, the parser for merged lists boils donw to a permutation parser.

Can we generalise the way we parse merged lists to parse more general structures?

Observation

If we restrict lists to length < 1, the parser for merged lists boils donw to a permutation parser.

Can we generalise the way we parse merged lists to parse more general structures?

The aim of this talk is to present a binary combinator <||>, such that p <||> q runs p and q in an interleaved way, i.e. the input is split into two sublists which are consumed by p respectively q.

2. Demo

3. Grammars

Applicative

The class Applicative describes **sequential composition** of "parsers":

```
class Applicative p where
  (<*>) :: p (b → a) → p b → p a
  pure  :: a                → p a
```

Parsers are combined using <*>, where the result of the combined parser is produced by applying the result of the left operand (of type $b \to a$)) to the result of the right operand (of type b).

Alternative

The class Alternative describes choice:

$$\textbf{class}\ \text{Alternative p}\ \textbf{where}$$
$$(\langle|\rangle) :: \text{p a} \rightarrow \text{p a} \rightarrow \text{p a}$$
$$\text{empty} :: \text{p a}$$

Alternative parsers are combined using $<|>$, and empty describes the always failing parser.

4. Grammars

Unwanted ambiguity

The following parser is ambiguous:

```
pa = ... -- recognises the string "a"
pb = ... -- recognises the string "b"
p `opt` v = p <|> pure v
ap = (++) <$> (pa `opt` "x") <||> pb
```

This parser will recognise "ab", "ba", "b" and "b" again, since the empty string recognisable by (pa `opt` "x") can be thought to be located before or after the "b".

Unwanted ambiguity

The following parser is ambiguous:

```
pa = ... -- recognises the string "a"
pb = ... -- recognises the string "b"
p `opt` v = p <|> pure v
ap = (++) <*> (pa `opt` "x") <||> pb
```

This parser will recognise "ab", "ba", "b" and "b" again, since the empty string recognisable by (pa `opt` "x") can be thought to be located before or after the "b". We decide to only include the second result.

Gram

The data type Gramm and Alt are used to represent merging parsers.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Gram\ f\ a} = \qquad \mathsf{Gram}\ [\mathsf{Alt\ f\ a}]\ (\mathsf{Maybe\ a})\\
&\textbf{data}\ \mathsf{Alt\ f\ a} \quad = \forall b.\mathsf{Seq}\quad (\mathsf{f}\ (b \to a))\ (\mathsf{Gram\ f\ b})\\
&\qquad\qquad\qquad\ \mid \forall b.\mathsf{Bind}\ \ (\mathsf{f\ b})\qquad\quad (b \to \mathsf{Gram\ f\ a})\\
&\qquad\qquad\qquad\ \mid \qquad \mathsf{Single}\ (\mathsf{f\ a})
\end{aligned}
$$

The first elements in the Seq, Bind and Single alternatives are parsers which are ready to be "run", and which may not be interrupted, i.e. which accept a consecutive part of the input.

Gram

The data type Gramm and Alt are used to represent merging parsers.

$$
\begin{array}{llll}
\textbf{data}\ \text{Gram f a} = & \text{Gram}\ [\text{Alt f a}]\ (\text{Maybe a}) & & \\
\textbf{data}\ \text{Alt f a} = & \forall b.\text{Seq} & (\text{f}\ (b \to a)) & (\text{Gram f b}) \\
\quad\quad\quad\quad\quad\ \mid & \forall b.\text{Bind} & (\text{f b}) & (b \to \text{Gram f a}) \\
\quad\quad\quad\quad\quad\ \mid & \text{Single}\ (\text{f a}) & &
\end{array}
$$

The first elements in the Seq, Bind and Single alternatives are parsers which are ready to be “run”, and which may not be interrupted, i.e. which accept a consecutive part of the input.

- Alt f a wil not recognise the empty string
- the Maybe a part describes whether a grammar can accept the empty string, and the result if this is the case

Parsers can be lifted to Gramars

A requirement is that parsers can be split in a part recognising a non-empty string and a value to be returned when the empty string can be recognised:

```
getOneP :: Maybe (P t a)-- provided by the base parsing l
getZeroP :: Maybe a      -- provided by the base parsing l

mkGram :: P t a → Gram (P t) a
mkGram p = Gram (maybe [] (pure ∘ Single) (getOneP p))
                (getZeroP p)
```

5. Building Merging Parsers

Constructing parsers from Grammars

Grammars can be converted to parsers:

$$
\begin{aligned}
&\mathsf{mkParserM} :: (\mathsf{Monad}\ f, \mathsf{Applicative}\ f, \ldots) \Rightarrow \mathsf{Gram}\ f\ a \to f\ a\\
&\mathsf{mkParserM}\ (\mathsf{Gram}\ ls\ le)\\
&\quad = \mathsf{foldr}\ (<|>)\ (\mathsf{maybe}\ \mathsf{empty}\ \mathsf{pure}\ le)\ (\mathsf{map}\ \mathsf{mkParserAlt}\ ls)\\
&\mathsf{mkParserAlt}\ (pb2a\ \text{`Seq`}\ gb\quad) = pb2a \circledast \mathsf{mkParserM}\ gb\\
&\mathsf{mkParserAlt}\ (pc\quad \text{`Bind`}\ c2ga) = pc \ggg\!= (\mathsf{mkParserM} \circ c2ga)\\
&\mathsf{mkParserAlt}\ (\mathsf{Single}\ pa\quad) = pa
\end{aligned}
$$

<||>

We will fom now on ignore the Binds. The operator <||> follows the applicative interface:

```
(<||>) :: Functor f ⇒ Gram f (b → a) → Gram f b → Gram f a
pg@(Gram pl pe) <||> qg@(Gram ql qe)
   = Gram ([(uncurry <$> p) `Seq` (((,) <$> pp) <||> qg)
                | p `Seq` pp ← pl]
     ++      [p `Seq` qg | Single p ← pl]
     ++      maybe [] (λpv → map (pv<$>) ql) pe
             ...-- similar for qg
     )       (pe <*> qe)
```

<||>

We will fom now on ignore the Binds. The operator <||> follows the applicative interface:

```
(<||>) :: Functor f ⇒ Gram f (b → a) → Gram f b → Gram f a
pg@(Gram pl pe) <||> qg@(Gram ql qe)
   = Gram ([(uncurry <$> p) `Seq` (((,) <$> pp) <||> qg)
                | p `Seq` pp ← pl]
     ++       [p `Seq` qg | Single p ← pl]
     ++       maybe [] (λpv → map (pv<$>) ql) pe
              ...-- similar for qg
     )        (pe <*> qe)
```

Note that this huge structure is built lazily during the actual parsing, as need arises!

6. Class Instances for Gram

Gram **is a** Functor

Grammars obey the conventional interface for parsers. The only difference is that they describe the break points.

```
instance Functor f ⇒ Functor (Gram f) where
  fmap f (Gram alts e) = Gram (map (f<$>) alts) (f <$> e)
instance Functor f ⇒ Functor (Alt f) where
  fmap a2c (pb2a `Seq` gb) = ((a2c◦) <$> pb2a) `Seq` gb
  fmap a2c (Single pc) = Single (a2c <$> pc)
```

Gram **is** Alternative

```
instance Functor f ⇒ Alternative (Gram f) where
  empty = Gram [] Nothing
  Gram ps pe <|> Gram qs qe = Gram (ps ++ qs) (pe <|> qe)
```

Gram is Applicative

```
instance Functor f ⇒ Applicative (Gram f) where
   pure a = Gram [] (Just a)
   Gram l le <*> ~rg@(Gram r re)
      = Gram (    map (`fwdby`rg) l
                ++ maybe [] (λe → map (e<$>) r) le
             ) (le <*> re)
(pb2c2a `Seq` gb) `fwdby` gc
   = (uncurry <$> pb2c2a) `Seq` ((, ) <$> gb <*> gc)
(Single pb2a) `fwbby` gb
   = pb2a `Seq` gb
```

Universiteit Utrecht

Conclusions

- Grammars are like parsers, but with <||> added
- Grammars are constructed lazily
- code is actually very simple
- types do the work, and tell us how to glue
- limited requirements on underlying parsing strategy

Conclusions

- Grammars are like parsers, but with $<||>$ added
- Grammars are constructed lazily
- code is actually very simple
- types do the work, and tell us how to glue
- limited requirements on underlying parsing strategy

Constructs like:

> many p $=$ (:) p $<||>$ many p $<|>$ pure $[\,]$

look innocent, but branch infinitely! Special care is needed.