Generalised Parsing and Combinator Parsing

A Happy Marriage?

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org

Royal Holloway, University of London

October 30, 2016
1 Conventional Parser Combinators
   - Recursive Descent
   - Monadic Parser Combinators
   - Benefits of Conventional Parser Combinators

2 Aspects of Generalised Parsing
   - Observable Sharing
   - Derivation Representation

3 Combinators for Generalised Parsing
   - Design Space Exploration
   - Challenges
Section 1

Conventional Parser Combinators
Recursive Descent Parsing

- Every symbol is implemented by a *parse function*

- A parse function:
  - Receives the input string, and a *pivot*
  - Returns a new pivot, and a bit of parse tree / semantic value

- The parse function for a nonterminal:
  - *Chooses* one of its productions (*somehow*)
  - Executes the symbols of the production in *sequence*

Combinator Approach

- Forget all about symbols and production

- Focus on parse functions and their composition!
Conventional Parser Combinators

Aspects of Generalised Parsing

Combinators for Generalised Parsing

Recursive Descent

Monadic Parser Combinators

Benefits of Conventional Parser Combinators

**type** `Parser t a = [t] → Int → [(Int, a)]`

- Full power of host language available to construct parsers

```
myparser :: ... → Parser t a
```

**Elementary matchers - create parsers from non-parser values**

```
term :: t → Parser t t
pred :: (t → Bool) → Parser t t
prefix :: [t] → Parser t [t]
```
type \textit{Parser} \( t \ a = [t] \rightarrow \text{Int} \rightarrow [(\text{Int}, a)] \)

- Full power of host language available to construct parsers

\textit{myparser} :: \ldots \rightarrow \text{Parser} \ t \ a

Elementary combinators - are defined by library internals

- e.g. \textit{seq} for placing parsers in sequence
- e.g. \textit{alt} for choosing between parsers

\textit{alt} :: \text{Parser} \ t \ a \rightarrow \text{Parser} \ t \ a \rightarrow \text{Parser} \ t \ a

\text{alt} \ p \ q \ \text{str} \ k = \ p \ \text{str} \ k \ \uplus \ q \ \text{str} \ k
• Define *return* and \( \gg = \)

• Prove monadic laws

**Parsers are monadic!**

\[
\text{return} :: a \rightarrow \text{Parser } t \ a \\
\text{return } a \ 	ext{str } l = \ldots
\]

\[
(\gg =) :: \text{Parser } t \ a \rightarrow (a \rightarrow \text{Parser } t \ b) \rightarrow \text{Parser } t \ b \\
(p \gg = a2q) \ 	ext{str } l = \ldots
\]
• Define $\text{return}$ and $\gg=$

• Prove monadic laws

**Parsers are monadic!**

\[
\text{return} :: a \rightarrow \text{Parser} \ t \ a \\
\text{return} \ a \ \text{str} \ l = [(l, a)]
\]

\[
(\gg=) :: \text{Parser} \ t \ a \rightarrow (a \rightarrow \text{Parser} \ t \ b) \rightarrow \text{Parser} \ t \ b \\
(p \gg= a2q) \ \text{str} \ l = [ \ (r, b) \\
\ | \ (k, a) \leftarrow p \ \text{str} \ l \\
, \ (r, b) \leftarrow (a2q \ a) \ \text{str} \ k]
\]
What do combinators offer beyond parser generators?

- Easy to define alternative *elementary matchers/combinators*
- Easy to define *derived combinators*
- Some form of context-sensitivity

- Advantages follow from the simplicity of the underlying algorithm
- There may be *other advantages specific to your application*
Section 2

Aspects of Generalised Parsing
• Conventional P.C. do not provide *any* grammar information

• Generalised Parsing requires *explicit grammar information*, for: *GSS construction, memoisation, curtailing left-recursive calls*...

• At the least, G.P. requires unique identifiers for symbols

• At the **very** least, G.P. requires identifiers for recursive positions

• All we need is...
• Conventional P.C. do not provide *any* grammar information

• Generalised Parsing requires *explicit grammar information*, for: GSS construction, memoisation, curtailing left-recursive calls...

• At the least, G.P. requires unique identifiers for symbols

• At the **very** least, G.P. requires identifiers for recursive positions

• All we need is... **observable sharing**

---

**Observable sharing**

- Detecting that two expressions arose from the same binding
- Simple pure ‘solution’: Ask the programmer!
- The way we obtain observable sharing influences how derived combinators are defined
\[ X \ ::= \ Y \ | \ Z \]
\[ Y \ ::= \ 'a' \ 'b' \]
\[ Z \ ::= \ 'c' \ Z \]
\[ X ::= Y \mid Z \]
\[ Y ::= \text{'a'} \mid \text{'b'} \]
\[ Z ::= \text{'c'} \mid Z \]

\[ \text{term} = 1 \]
\[ \oplus = (+) \]
\[ \otimes = (+) \]
X ::= Y | Z
Y ::= ‘a’ ‘b’
Z ::= ‘c’ Z

term = 1
⊕ = (+)
⊗ = (+)
Conventional Parser Combinators

Aspects of Generalised Parsing

Combinators for Generalised Parsing

Observable Sharing

Derivation Representation

\[
X \::= Y \mid Z
\]

\[
Y \::= 'a' \ 'b'
\]

\[
Z \::= 'c' \ Z
\]

\[
\text{term} = 1
\]

\[
\oplus = (+)
\]

\[
\otimes = (+)
\]
Conventional Parser Combinators
Aspects of Generalised Parsing
Combinators for Generalised Parsing

Observable Sharing
Derivation Representation

\[
X \quad Y \quad Z
\]

\[
X ::= Y \mid Z
Y ::= \text{'}a\text{'} \mid \text{'}b\text{'}
Z ::= \text{'}c\text{'} \mid Z
\]

\[
\text{term } = 1
\]

\[
\oplus = (+)
\otimes = (+)
\]

\[
\begin{array}{c}
X \\
Y \\
Z
\end{array}
\]

\[
\begin{array}{c}
Y \\
a \\
Z
\end{array}
\]

\[
\begin{array}{c}
Z \\
b \\
c
\end{array}
\]

\[
\begin{array}{c}
\oplus \\
\otimes \\
\otimes
\end{array}
\]

\[
\begin{array}{c}
\oplus \\
a \\
b \\
c
\end{array}
\]

\[
\begin{array}{c}
\otimes \\
\otimes \\
\otimes
\end{array}
\]

\[
\begin{array}{c}
\otimes
a \\
b \\
c
\end{array}
\]

\[
\begin{array}{c}
\otimes
\end{array}
\]

\[
\begin{array}{c}
\text{'}c\text{'}
\end{array}
\]

\[
\begin{array}{c}
\text{'}Z\text{'}
\end{array}
\]
\[ X \::= Y \mid Z \]
\[ Y \::= 'a' 'b' \]
\[ Z \::= 'c' Z \]

\[ \text{term} = 1 \]
\[ \oplus = (+) \]
\[ \otimes = (+) \]
Conventional Parser Combinators
Aspects of Generalised Parsing
Combinators for Generalised Parsing
Observable Sharing
Derivation Representation

\[
X \rightarrow \text{“X”} \\
Y \rightarrow \text{“Y”} \\
Z \rightarrow \text{“Z”}
\]

\[
Y \rightarrow a \otimes b \\
Z \rightarrow c \otimes Z
\]

\[X ::= Y \mid Z\]
\[Y ::= ‘a’ \ ‘b’\]
\[Z ::= ‘c’ Z\]

\[\text{term} = 1\]
\[\oplus = (+)\]
\[\otimes = (+)\]
A Generalised Parser produces a representation of all derivations.

Potentially exponentially many derivations may be embedded.

Downstream enumeration would result in exponential runtimes...
• A Generalised Parser produces a representation of all derivations
• Potentially exponentially many derivations may be embedded
• Downstream enumeration would result in exponential runtimes...
How may disambiguation strategies manifest themselves?

1. A more deterministic parsers being constructed
2. Pruning the representation of all derivations (top-down)
3. Choosing between derivations by evaluation (bottom-up)
How may disambiguation strategies manifest themselves?

1. A more deterministic parsers being constructed
2. Pruning the representation of all derivations (top-down)
3. Choosing between derivations by evaluation (bottom-up)

- How to provide expressive, but opaque, strategies?
How may disambiguation strategies manifest themselves?

1. A more deterministic parsers being constructed
2. Pruning the representation of all derivations (top-down)
3. Choosing between derivations by evaluation (bottom-up)

• How to provide expressive, but opaque, strategies?
• How do the strategies interact with user-derived combinators?
Section 3

Combinators for Generalised Parsing
<table>
<thead>
<tr>
<th>Parser Combinators</th>
<th>parsec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UU-lib</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Explicit Nonterminals</td>
<td>Scheme recognisers  (Johnson 1995)</td>
</tr>
<tr>
<td></td>
<td>Meerkat  (Izmaylova/Afroozeh 2015/16)</td>
</tr>
<tr>
<td>Grammar Combinators</td>
<td>P3       (Ridge 2014)</td>
</tr>
<tr>
<td></td>
<td>GLL.Combinators  (2015/16)</td>
</tr>
<tr>
<td></td>
<td>grammar-combinators (Devriese 2011/12)</td>
</tr>
<tr>
<td>Meta-Programming</td>
<td>BNFC-meta (Duregard 2011)</td>
</tr>
<tr>
<td>Parser Generators</td>
<td>Bison</td>
</tr>
<tr>
<td></td>
<td>yacc</td>
</tr>
<tr>
<td></td>
<td>Happy</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Properties of a “good” library

- Supports *semantic actions*
- Worst-case runtime $O(n^3)$ & Precomputation at compile-time
- Provides a DSL with little to learn
- Allows users to define derived combinators (with little or no knowledge of internals)

**Bonus**

- Allows users to define elementary matcher or combinators (without too much effort/risk)
- Supports some form of context-sensitivity (like monadic parsers)
Generalised Parsing and Combinator Parsing
A Happy Marriage?

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org

Royal Holloway, University of London

October 30, 2016
“To extend your library - How would you define the following?”

Alternative Elementary Combinators
- Combinator `pred`, matching terminals satisfying a predicate
- The internal equivalent of Kleene-closure

“With your library - How can a user define the following?”

Derived Combinators
- Combinator `many`, the ‘external’ Kleene-closure
- Combinator `chainl` for expression grammars
- Ideally knowing nothing or little of the library’s internals

“How do disambiguation strategies mix with derived combinators?”