Generalised Parsing with Parser Combinators

L. Thomas van Binsbergen

Royal Holloway, University of London

5 January, 2016

Goals

- Introduce and motivate generalised parsing.
- Explain Earley's generalised parsing algorithm.
- Explain Johnson's combinators for generalised recognition.
- Suggest a method to extend the combinators to parsers.

Generalised Parsing in Context

The PLanCompS project

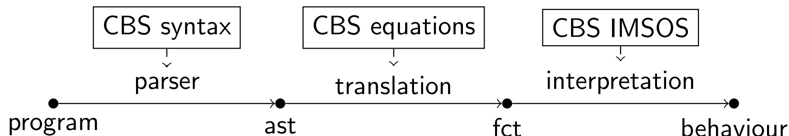


Figure : PLanCompS: generate interpreters from reusable specification.

Joint project

- **semantics** @ Swansea University (Peter Mosses):
  IMSOS, Implicitly Modular *Structural Operational Semantics*.
- **parsing** @ Royal Holloway, University of London.
  "*Wait a second, is parsing not a finished topic?*"
  RHUL delivers *Generalised Parsing* (E. Scott & A. Johnstone).

The PLanCompS project

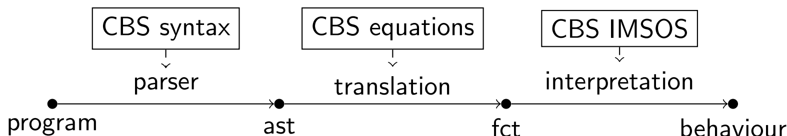


Figure : PLanCompS: generate interpreters from reusable specification.

Relying on your background, can you study and explain:

- Generalised parsing as part of parser combinators.
- IMSOS and Swansea's specification language CBS.

Background @ Utrecht University

- **semantics**:
  Haskell, Attribute Grammars, Parser Combinators, SOS, ...

Conventional Parsing

- Parsing is a major success story in Computer Science:
- We have a well-understood and simple formalism (BNF).
- And many algorithms:
  LR, LR(k), SLR, LALR,
  LL, LL(k), ...
- (a variant of) BNF is used in all modern language definitions.
- Many tools exist that generate fast parsers from BNF specifications: yacc, happy, ...

Conventional Parsing

- Parsing is a major success story in Computer Science:
- We have a well-understood and simple formalism (BNF).
- And many algorithms:
  LR, LR(k), SLR, LALR, shift/reduce conflicts
  LL, LL(k), left-recursion, non-left-factored...
- (a variant of) BNF is used in all modern language definitions.
- Many tools exist that generate fast parsers from BNF specifications: yacc, happy, ...
- The only problem arises when your grammar does not satisfy the restrictions of the chosen parsing technology.

Generalised Parsing

- A generalised parser works for all grammars, including grammars that are (highly) ambiguous.
- A parser is only general if it outputs all valid derivations (potentially infinitely or exponentially many).
- To do so efficiently, a sharing representation must be used.
- State of the art: runtime and space complexity of $O(n^3)$.
- Algorithms: Earley's algorithm (1970), GLR (Tomita 1984), GLL (Scott & Johnstone 2010).

Generalised Parsing helps Semantics-oriented

- Main motivation: any grammar admits a parser (fail-safe).
- After designing the grammar:
  - What are the sources of ambiguity?
  - How to disambiguate?
  - What can I do to improve runtime of the parser?
- Additional grammar annotations for disambiguation and transformation.
- Especially helpful in semantics-oriented tools: Spoofax, K framework, Ott, ..., UUAGC(??)

Parsing terminology

- A symbol is either a terminal or a nonterminal.
- A *language* is a set of *sentences* (sequence of terminals).
- A *grammar* Γ is a set of productions (generating a language).
- A production $X ::= \alpha \in \Gamma$ has left-hand side X (nonterminal) and right-hand side α (a sequence of symbols).
- *Parsers* and *recognisers* for Γ determine whether a sentence I can be derived from Γ.
- This is denoted as $\Gamma \vdash S \rightarrow I_{0,m}$.

Inference rules

$$\textsc{term}\ \frac{I_{l,r} = t}{\Gamma \vdash t \to I_{l,r}}$$

$$\textsc{nterm}\ \frac{\exists X ::= \beta \in \Gamma \qquad \Gamma \vdash X ::= \beta \to I_{l,r}}{\Gamma \vdash X \to I_{l,r}}$$

$$\textsc{prod}\ \frac{\begin{array}{c}\exists k_1, \ldots, k_{j-1} \\ l = k_0 \qquad \forall i.\, \Gamma \vdash x_i \to I_{k_{i-1},k_i} \qquad r = k_j\end{array}}{\Gamma \vdash X ::= x_1 \ldots x_j \to I_{l,r}}$$

More terminology

Ambiguity

- Ambiguity means there are multiple derivations of a sentence.
- There are two kinds of ambiguity:
  - Multiple productions of X derive the same substring.
  - Multiple sets of pivots work for a production.

Parsers and Recognisers

- A *recogniser* computes whether there is a derivation.
- A *parser* computes a single derivation (if there is one).
- A *generalised parser* computes all derivations.

Earley's algorithm (1970)

Earley's algorithm

- A *slot* $X ::= \alpha \cdot \beta$ denotes a partially matched production.
- *Earley sets* contain *Earley items*: $\langle slot, index \rangle$.
- Earley sets $E_1 \ldots E_m$ are initially empty.
- Earley set E_0 initially contains $\langle S' ::= \cdot S, 0 \rangle$.

Earley's algorithm (2)

1. Starting with $k = 0$.
2. Process unprocessed items from E_k in this order:
    1. $\langle X ::= \alpha \cdot Y\beta, l\rangle$,
       by adding $\langle Y ::= \cdot\beta, k\rangle$ to E_k, for all $Y ::= \beta \in \Gamma$.
    2. $\langle Y ::= \beta\cdot, l\rangle$,
       by adding $\langle X ::= \alpha Y \cdot \beta, l'\rangle$ to E_k, iff $\langle X ::= \alpha \cdot Y\beta, l'\rangle \in E_l$.
    3. $\langle X ::= \alpha \cdot t\beta, l\rangle$,
       by adding $\langle Y ::= \alpha t \cdot \beta, l\rangle$ to E_{k+1}, iff $I_{k,k+1} \equiv t$.
3. If all items in E_k are processed, continue with E_{k+1}.

Parsing

Store pivot when 'the dot is carried across a symbol', i.e. iff $\langle Y ::= \alpha \cdot x\beta, l\rangle \in E_k$ adds $\langle Y ::= \alpha x \cdot \beta, l\rangle$ to E_r via (2.2) or (2.3), insert $(Y ::= \alpha x \cdot \beta, l, k, r)$ in set P (Scott 2010).

Earley's algorithm (Example)

```
X ::= AB
A ::= a | aa
B ::= a | aa
input = "aaa"

E0 ('a')
< S ::= .X , 0 >
< X ::= .AB, 0 >
< A ::= .a , 0 >
< A ::= .aa, 0 >
E1 ('a')
< A ::= a. , 0 >
< A ::= a.a, 0 >
< X ::= A.B, 0 >
< B ::= .a , 1 >
< B ::= .aa, 1 >
```

```
E2 ('a')
< A ::= aa., 0 >
< B ::= a. , 1 >
< B ::= a.a, 1 >
< X ::= A.B, 0 >
< X ::= AB., 0 >
< B ::= .a , 2 >
< B ::= .aa, 2 >
< S ::= X.  ,0 >
E3 ('$')
< B ::= aa., 1 >
< B ::= a. , 2 >
< B ::= a.a, 2 >
< X ::= AB., 0 >
</X/::=/AB.,/0/>
< S ::= X. , 0 >
```

```
PIVOTS:
(A ::= a. ,0,0,1)
(A ::= a.a,0,0,1)
(X ::= A.B,0,0,1)
(A ::= aa.,0,1,2)
(B ::= a. ,1,1,2)
(B ::= a.a,1,1,2)
(X ::= A.B,0,0,2)
(X ::= AB.,0,1,2)
(S ::= X., 0,0,2)
(B ::= aa.,1,2,3)
(B ::= a. ,2,2,3)
(B ::= a.a,2,2,3)
(X ::= AB.,0,1,3)
(X ::= AB.,0,2,3)
(S ::= X. ,0,0,3)
```

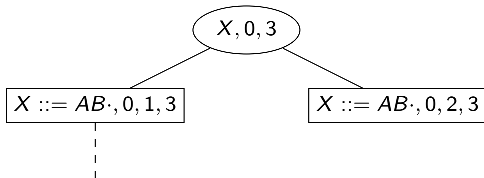
Constructing derivations from pivots

```
PIVOTS:
(A ::= a.  ,0,0,1)
(A ::= a.a,0,0,1)
(X ::= A.B,0,0,1)
(A ::= aa.,0,1,2)
(B ::= a.  ,1,1,2)
(B ::= a.a,1,1,2)
(X ::= A.B,0,0,2)
(X ::= AB.,0,1,2)
(S ::= X., 0,0,2)
(B ::= aa.,1,2,3)
(B ::= a.  ,2,2,3)
(B ::= a.a,2,2,3)
(X ::= AB.,0,1,3)
(X ::= AB.,0,2,3)
(S ::= X.  ,0,0,3)
```

$X, 0, 3$

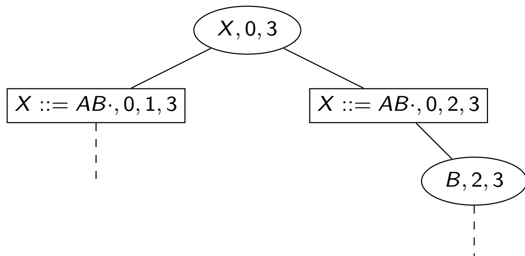
Constructing derivations from pivots

```
PIVOTS:
(A ::= a. ,0,0,1)
(A ::= a.a,0,0,1)
(X ::= A.B,0,0,1)
(A ::= aa.,0,1,2)
(B ::= a. ,1,1,2)
(B ::= a.a,1,1,2)
(X ::= A.B,0,0,2)
(X ::= AB.,0,1,2)
(S ::= X., 0,0,2)
(B ::= aa.,1,2,3)
(B ::= a. ,2,2,3)
(B ::= a.a,2,2,3)
(X ::= AB.,0,1,3)
(X ::= AB.,0,2,3)
(S ::= X. ,0,0,3)
```



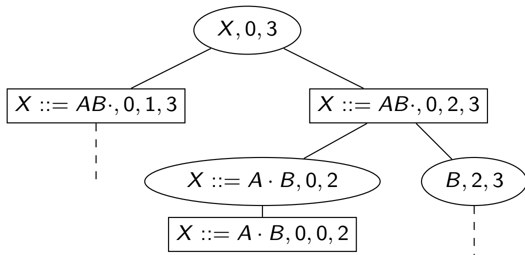
Constructing derivations from pivots

```
PIVOTS:
(A ::= a.  ,0,0,1)
(A ::= a.a,0,0,1)
(X ::= A.B,0,0,1)
(A ::= aa.,0,1,2)
(B ::= a.  ,1,1,2)
(B ::= a.a,1,1,2)
(X ::= A.B,0,0,2)
(X ::= AB.,0,1,2)
(S ::= X., 0,0,2)
(B ::= aa.,1,2,3)
(B ::= a.  ,2,2,3)
(B ::= a.a,2,2,3)
(X ::= AB.,0,1,3)
(X ::= AB.,0,2,3)
(S ::= X.  ,0,0,3)
```



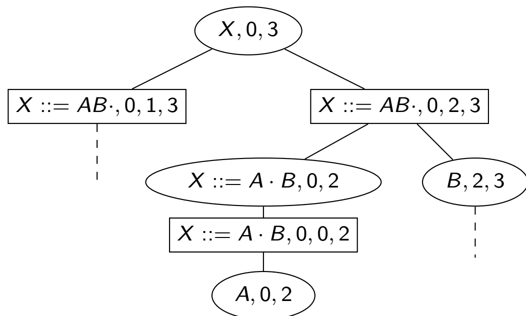
Constructing derivations from pivots

```
PIVOTS:
(A ::= a.  ,0,0,1)
(A ::= a.a,0,0,1)
(X ::= A.B,0,0,1)
(A ::= aa.,0,1,2)
(B ::= a.  ,1,1,2)
(B ::= a.a,1,1,2)
(X ::= A.B,0,0,2)
(X ::= AB.,0,1,2)
(S ::= X., 0,0,2)
(B ::= aa.,1,2,3)
(B ::= a.  ,2,2,3)
(B ::= a.a,2,2,3)
(X ::= AB.,0,1,3)
(X ::= AB.,0,2,3)
(S ::= X.  ,0,0,3)
```



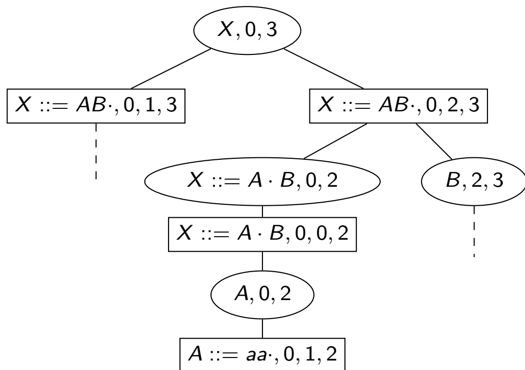
Constructing derivations from pivots

```
PIVOTS:
(A  ::= a.  ,0,0,1)
(A  ::= a.a,0,0,1)
(X  ::= A.B,0,0,1)
(A  ::= aa.,0,1,2)
(B  ::= a.  ,1,1,2)
(B  ::= a.a,1,1,2)
(X  ::= A.B,0,0,2)
(X  ::= AB.,0,1,2)
(S  ::= X., 0,0,2)
(B  ::= aa.,1,2,3)
(B  ::= a.  ,2,2,3)
(B  ::= a.a,2,2,3)
(X  ::= AB.,0,1,3)
(X  ::= AB.,0,2,3)
(S  ::= X.  ,0,0,3)
```



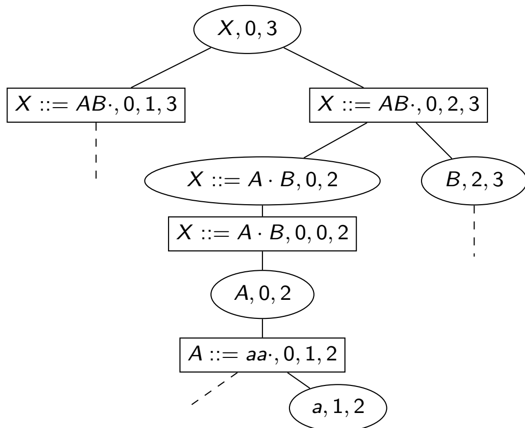
Constructing derivations from pivots

```
PIVOTS:
(A  ::= a.  ,0,0,1)
(A  ::= a.a,0,0,1)
(X  ::= A.B,0,0,1)
(A  ::= aa.,0,1,2)
(B  ::= a.  ,1,1,2)
(B  ::= a.a,1,1,2)
(X  ::= A.B,0,0,2)
(X  ::= AB.,0,1,2)
(S  ::= X., 0,0,2)
(B  ::= aa.,1,2,3)
(B  ::= a.  ,2,2,3)
(B  ::= a.a,2,2,3)
(X  ::= AB.,0,1,3)
(X  ::= AB.,0,2,3)
(S  ::= X.  ,0,0,3)
```



Constructing derivations from pivots

```
PIVOTS:
(A  ::= a.  ,0,0,1)
(A  ::= a.a,0,0,1)
(X  ::= A.B,0,0,1)
(A  ::= aa.,0,1,2)
(B  ::= a.  ,1,1,2)
(B  ::= a.a,1,1,2)
(X  ::= A.B,0,0,2)
(X  ::= AB.,0,1,2)
(S  ::= X., 0,0,2)
(B  ::= aa.,1,2,3)
(B  ::= a.  ,2,2,3)
(B  ::= a.a,2,2,3)
(X  ::= AB.,0,1,3)
(X  ::= AB.,0,2,3)
(S  ::= X.  ,0,0,3)
```



Generalised Recognition with Combinators

Parser Combinator Approach

Parser generators

- Parsers can be generated from a grammar description.
- Relying on the formalism fixed by the parser generator.

Handwritten parsers

- A parser can also be written 'by hand'.
- Full power of the chosen implementation language is available.
- How to reason about the correctness of a handwritten parser?

Parser Combinator Approach

Parser Combinators

- Parser combinators are (ho-)functions for writing parsers.
- The elementary combinators are:
  term, *epsilon*, $\otimes$ (sequencing) and $\oplus$ (alternation).
- They implement a top-down parsing algorithm.

Advantages

- Parsers 'look like' BNF specifications.
- Parsers are easy to write and reason about.
- Derived combinators for common patterns.
- Easy to debug, as sub-parsers can be tested individually.

Example parser

$$
\begin{aligned}
X &::= AB \\
A &::= a \\
  &\quad \mid aa \\
B &::= a \\
  &\quad \mid aa
\end{aligned}
$$

$$
\begin{aligned}
pX &= pA \otimes pB \\
pA &= \mathit{term}\ \texttt{'a'} \\
   &\quad \oplus \mathit{term}\ \texttt{'a'} \otimes \mathit{term}\ \texttt{'a'} \\
pB &= \ldots
\end{aligned}
$$

Parser Combinator Approach

Disadvantages

- A parser may look like a BNF specification, but it does not actually specify a grammar!
- Combinators can only implement top-down parsing.

Disclaimer

- Combinator libraries exist for specifying grammars, with parsing algorithms that work on these grammars.
- However, they often severely limit the ease with which derived combinators can be defined.

Basic Combinators

$$\textbf{type}\ \mathit{Recogniser} = \mathit{String} \rightarrow \mathit{Int} \rightarrow [\mathit{Int}]$$

$$\begin{aligned}
&\mathit{term} :: \mathit{Char} \rightarrow \mathit{Recogniser} \\
&\mathit{term}\ t\ \mathit{str}\ k \mid \mathit{match}\ \mathit{str}\ k\ t = [k+1] \\
&\qquad\qquad\quad \mid \mathit{otherwise} \quad\ = [\,]
\end{aligned}$$

$$\begin{aligned}
&\mathit{epsilon} :: \mathit{Recogniser} \\
&\mathit{epsilon}\ \mathit{str}\ k = [k]
\end{aligned}$$

Basic Combinators (2)

$$(\otimes), (\oplus) :: \mathit{Recogniser} \rightarrow \mathit{Recogniser} \rightarrow \mathit{Recogniser}$$
$$(p \otimes q)\ \mathit{str}\ l = [r \mid k \leftarrow p\ \mathit{str}\ l, r \leftarrow q\ \mathit{str}\ k]$$
$$(p \oplus q)\ \mathit{str}\ k = p\ \mathit{str}\ k +\!\!+\ q\ \mathit{str}\ k$$

$$\mathit{recognises} :: \mathit{Recogniser} \rightarrow \mathit{String} \rightarrow \mathit{Bool}$$
$$\mathit{recognises}\ p\ \mathit{str} = \textbf{let}\ \mathit{pivots} = p\ \mathit{str}\ 0$$
$$m = \mathit{length}\ \mathit{str}$$
$$\textbf{in}\ \mathit{any}\ (\equiv m)\ \mathit{pivots}$$

Combinators in Continuation Passing Style

$$\textbf{type}\ Recogniser = String \to Cont \to Int \to Bool$$
$$\textbf{type}\ Cont = Int \to Bool$$

$$term :: Char \to Recogniser$$
$$term\ t\ str\ c\ k \mid match\ str\ k\ t = c\ (k+1)$$
$$\mid otherwise = False$$

$$epsilon :: Recogniser$$
$$epsilon\ str\ c\ k = c\ k$$

Combinators in Continuation Passing Style (2)

$$
\begin{aligned}
&(\otimes), (\oplus) :: \mathit{Recogniser} \rightarrow \mathit{Recogniser} \rightarrow \mathit{Recogniser}\\
&(p \otimes q)\ \mathit{str}\ c\ l\ \ = p\ \mathit{str}\ (q\ \mathit{str}\ c)\ l\\
&(p \oplus q)\ \mathit{str}\ c\ k = p\ \mathit{str}\ c\ k \vee q\ \mathit{str}\ c\ k
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{recognises} :: \mathit{Recogniser} \rightarrow \mathit{String} \rightarrow \mathit{Bool}\\
&\mathit{recognises}\ p\ \mathit{str} = \textbf{let}\ c_0\ r = r \equiv m\\
&\qquad\qquad\qquad\qquad\quad m\ \ \ = \mathit{length}\ \mathit{str}\\
&\qquad\qquad\qquad\quad \textbf{in}\ p\ \mathit{str}\ c_0\ 0
\end{aligned}
$$

Combinators are not general

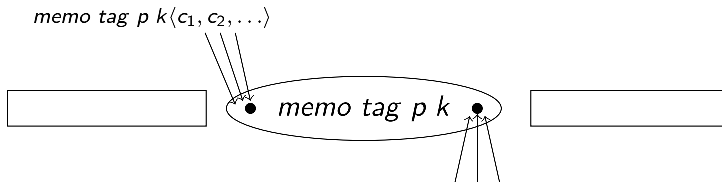
(Left-)Recursion problem

- If running parser X requires running parser X with the same arguments, the parser will not terminate.
- Possible solution:
  - Tag recursion.
  - Add additional argument that remembers encountered tags.

Duplication of work

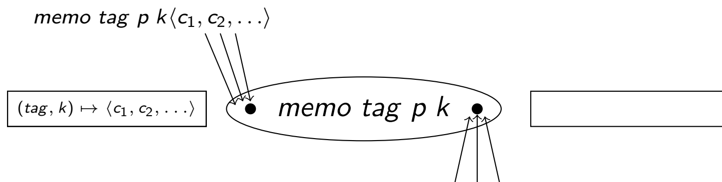
- Exponential runtime on highly ambiguous grammars.
- Clever memoisation solves both problems. (Johnson 1995)

Johnson's memo combinator



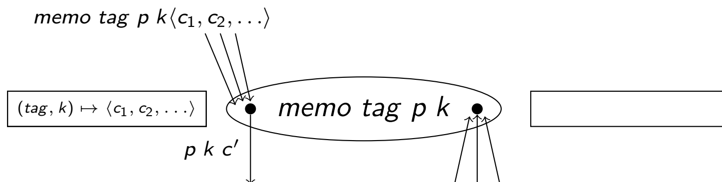
- All arriving continuations are applied to all discovered pivots.
- No duplication:
  - Only the first application of $memo\ tag\ p\ k$ calls $p\ k$.
  - No continuation $c_1, c_2, \ldots$ is applied to the same r_i twice.

Johnson's memo combinator



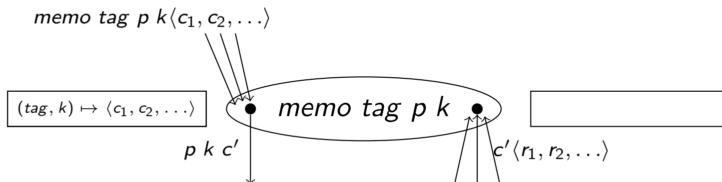
- All arriving continuations are applied to all discovered pivots.
- No duplication:
  - Only the first application of *memo tag p k* calls *p k*.
  - No continuation $c_1, c_2, \ldots$ is applied to the same r_i twice.

Johnson's memo combinator



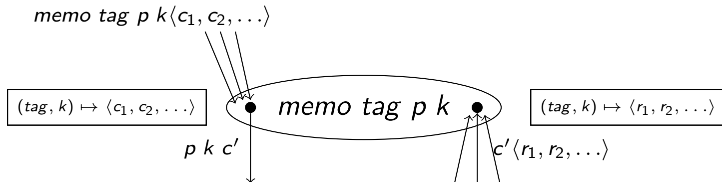
- All arriving continuations are applied to all discovered pivots.
- No duplication:
  - Only the first application of *memo tag p k* calls *p k*.
  - No continuation $c_1, c_2, \ldots$ is applied to the same r_i twice.

Johnson's memo combinator



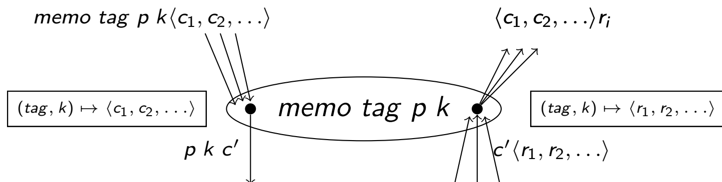
- All arriving continuations are applied to all discovered pivots.
- No duplication:
  - Only the first application of *memo tag p k* calls *p k*.
  - No continuation $c_1, c_2, \ldots$ is applied to the same r_i twice.

Johnson's memo combinator



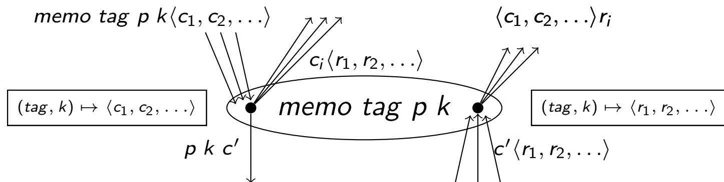
- All arriving continuations are applied to all discovered pivots.
- No duplication:
  - Only the first application of *memo tag p k* calls *p k*.
  - No continuation $c_1, c_2, \ldots$ is applied to the same r_i twice.

Johnson's memo combinator



- All arriving continuations are applied to all discovered pivots.
- No duplication:
  - Only the first application of *memo tag p k* calls *p k*.
  - No continuation $c_1, c_2, \ldots$ is applied to the same r_i twice.

Johnson's memo combinator



- All arriving continuations are applied to all discovered pivots.
- No duplication:
  - Only the first application of *memo tag p k* calls *p k*.
  - No continuation $c_1, c_2, \ldots$ is applied to the same r_i twice.

From recognition to parsing

- The presented combinators are for recognisers only.
- Currently no extension to parsers in literature.
- Frost et al. (1998) propose an alternative.
- Generalised parsing is implemented in 'grammar combinators':
  - PhD thesis Ljunglöf (2002).
  - Ridge (2014).
  - Hackage package 'gll': https://hackage.haskell.org/package/gll

Generalised Parsing with Combinators

Desired Properties of Generalised Parser Combinators

- Same properties as state of the art generalised parsers:
  - $O(n^3)$ space and time complexity.
  - Disambiguation.
  - (explicit underlying grammar, for debugging)
- User-specified semantic actions (Applicative-like).
- Fully compositional.
- Easy to define derived combinators.
- (Impossible to write non-terminating expressions)

Recognition is not Parsing

Semantic actions on the fly

- Combinators can be extended with 'semantic actions'.
- Exponential runtime if exponentially many derivations exist.

Count occurrences of "ab" in a string of "ab"s?

$$\begin{aligned} pX &= plus \odot term\ \texttt{'a'} \otimes term\ \texttt{'b'} \otimes pX \\ &\quad \oplus epsilon\ 0 \\ &\quad \textbf{where}\ plus\ a\ b\ x = 1 + x \\ &\qquad\qquad zero\ () \quad = 0 \end{aligned}$$

Recognition is not Parsing

Post-parse semantic actions

- Instead, pivots must be remembered efficiently.
- Disambiguation takes place in a post-parse phase.
- The parse is re-run in reverse, guided by the (reduced) pivots, whilst applying semantic actions. (Ridge 2014)

Tagging recursion with Strings

- User needs to invent (unique) names.
- Potential for name clashes and unexpected behaviour.
- Derived combinators can be defined (but not as easily).
- Used in: https://hackage.haskell.org/package/gll

Optional

$$\mathit{optional} :: \mathit{Parser}\ a \rightarrow \mathit{Parser}\ (\mathit{Maybe}\ a)$$
$$\mathit{optional}\ p = \mathit{epsilon}\ \mathit{Nothing}$$
$$\qquad\qquad \oplus\ \mathit{Just} \odot p$$

Iteration

$$
\begin{array}{l}
\mathit{many_as'} :: \mathit{Parser}\ [\mathit{Char}] \\
\mathit{many_as'} = \mathit{epsilon}\ [\,] \\
\qquad\qquad \oplus\ \mathit{cons} \odot \mathit{many_as'} \otimes \mathit{term}\ \texttt{'a'} \\
\quad \textbf{where}\ \mathit{cons}\ \mathit{as}\ a = \mathit{as} +\!\!+ [a]
\end{array}
$$

$$
\begin{array}{l}
\mathit{many_as} :: \mathit{Parser}\ [\mathit{Char}] \\
\mathit{many_as} = \mathit{memo}\ \texttt{"many_as"}\ (\\
\qquad\qquad \mathit{epsilon}\ [\,] \\
\qquad\qquad \oplus\ \mathit{cons} \odot \mathit{many_as} \otimes \mathit{term}\ \texttt{'a'}) \\
\quad \textbf{where}\ \mathit{cons}\ \mathit{as}\ a = \mathit{as} +\!\!+ [a]
\end{array}
$$

Iteration (2)

$$
\begin{aligned}
&\mathit{many} :: \mathit{Parser}\ a \rightarrow \mathit{Parser}\ [a] \\
&\mathit{many}\ p = \mathit{memo}\ \mathit{tag}\ (\\
&\qquad\qquad \mathit{epsilon}\ [\,] \\
&\qquad\quad \oplus (:) \odot \mathit{many}\ p \otimes p) \\
&\quad \textbf{where}\ \mathit{cons}\ \mathit{xs}\ x = \mathit{xs} +\!\!+ [x] \\
&\qquad\qquad \mathit{tag} = ???
\end{aligned}
$$

Iteration (2)

$$
\begin{aligned}
&\mathit{many} :: \mathit{Parser}\ a \rightarrow \mathit{Parser}\ [a] \\
&\mathit{many}\ p = \mathit{memo}\ \mathit{tag}\ (\\
&\qquad\qquad \mathit{epsilon}\ [\,] \\
&\qquad\quad \oplus\ (:) \odot \mathit{many}\ p \otimes p) \\
&\quad \textbf{where}\ \mathit{cons}\ \mathit{xs}\ x = \mathit{xs} +\!\!+ [x] \\
&\qquad\qquad \mathit{tag} = ???
\end{aligned}
$$

- $\mathit{tag} = $ `"_many("` $+\!\!+\ \mathit{tag_of}\ p +\!\!+$ `")"`

Observable Sharing

- Tagfull.
- Finally tagless (Carette 2009, Devriese 2012).
  Difficult to define even simple derived combinators.
- Pointer equality on references (Claessen 1999).
  Relies on *unsafePerformIO*.
- Stable names (Gill 2009).
  Computes a graph representation of program in IO monad.
  How to combine with semantic actions / type arguments?
- ... ?

Future work

- Combinators for top-down disambiguation.
- Is there a method for Observable Sharing that provides the desired flexibility?
- Demonstrate practicality, by implementing parsers for Caml Light and Haskell.