Advert: 2nd Metaprogramming Summer School

11–16 August 2019 Dagstuhl, Germany

https://www.cl.cam.ac.uk/events/metaprog/2019/

Oleg Kiselyov

Conor McBride

Matthew Flatt

Jonathan Protzenko

A typed, algebraic approach to parsing

Neelakantan R. Krishnaswami
Jeremy Yallop

Parsing, again?

Most practical questions in parsing are essentially solved.

Parser combinators offer full host language integration

```
let exprs = alt (expr ==> single)
                (expr ++ chr "," ++ exprs ==> cons)
```

Parser generators (e.g. yacc) guarantee linear-time execution

```
exprs: expr { [$1] }
     | expr COMMA exprs { $1 :: $2 }
```

Parsing, again?

Most practical questions in parsing are essentially solved.

Parser combinators offer full host language integration

```
let exprs = alt (expr ==> single)
                (expr ++ chr "," ++ exprs ==> cons)
```

Parser generators (e.g. yacc) guarantee linear-time execution

```
exprs: expr { [$1] }
     | expr COMMA exprs { $1 :: $2 }
```

This work: abstract like parser combinators / perform like yacc

Plan

Start with context-free expressions (regexes + μ) $\mu x.c \vee \bot \cdot \epsilon$
(Leiß 1991)

Add types (and a type system) for *unambiguous* expressions $\Gamma; \Delta \vdash g : \tau$
(build on Brüggemann-Klein & Wood, 1992)

Build parser combinators `chr 'a' ++ chr 'b'`
(à la Swierstra & Duponcheel, 1996; *unembed* following Atkey et al 2009)

Stage the combinators `.< if s.[i] > n then ... >.`
(cf. Jonnalagedda et al 2014)

Parser combinators

Abstract grammars (à la Swierstra & Duponcheel (1996), but with `fix`):

```
type α t

val chr: char → char t

val eps: unit t
val seq: α t → β t → (α * β) t

val bot: α t
val alt: α t → α t → α t

val fix: (α t → α t) → α t

val map: (α → β) → α t → β t
```

Parsers from grammars

```
val parser: α t → (char Stream.t → α)     (* deterministic parsing! *)
exception TypeError of string             (* 'parser' can fail *)
```

(Simplification: here we just use `char`; in our library everything's parameterized by the token type.)

Using combinators: building a library

```
let (++) = seq                      (* infix version of 'seq' *)
let (==>) p f = map f p             (* infix version of 'map *)
let always x = fun _ → x            (* for constant semantic actions *)
let any gs = fold_left alt bot gs   (* n-ary version of 'alt' *)

val option : α t → α option t       (* 'option g' parses zero or one gs *)
let option r =  any [eps ==> always None;
                     r   ==> fun x → Some x]

val star : α t → α list t           (* 'star g' parses zero or more gs *)
let star g = fix (fun rest → any [eps       ==> always [];
                                  g ++ rest ==> fun (x, xs) → x :: xs ])

val plus : α t → α list t           (* 'plus g' parses zero or more gs *)
let plus g = g ++ star g ==> fun (x, xs) → x :: xs
```

(More in the paper: e.g. combinators for dealing with precedence & associativity.)

Using combinators: tokenizers and parsers

Building tokenizers

```
let charset s = any (List.map chr (list_of_string s))

let lower = charset "abcdefghijklmnopqrstuvwxyz"
let upper = charset "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

type token = SYMBOL of string | LPAREN | RPAREN
let word = upper ++ star lower ==> fun (c,cs) → string_of_list (c ::cs)
let token = any [chr '(' ==> always LPAREN;
                 chr ')' ==> always RPAREN;
                 word    ==> fun s → SYMBOL s]
```

Building parsers

```
type sexp = Sym | Seq of sexp list
let paren p = lparen ++ p ++ rparen ==> fun ((_, x), _) → x
let sexp = fix (fun sexp → any [symbol            ==> always Sym;
                                paren (star sexp) ==> fun s → Seq s])
```

Rejecting bad grammars

```
any [chr 'a' ==> always 1;
     chr 'a' ==> always 2]
```

```
option (chr 'a') ++ option (chr 'a')
```

```
fix (fun left → any [eps       ==> always [];
                     left ++ p ==> fun (xs, x) → append xs [x]])
```

```
any [chr 'a' ++ chr 'b' ==> always 1;
     chr 'a' ++ chr 'c' ==> always 2]
```

Rejecting bad grammars

```
any [chr 'a' ==> always 1;
     chr 'a' ==> always 2]
```

Type error: *disjunctive non-determinism*

```
option (chr 'a') ++ option (chr 'a')
```

```
fix (fun left → any [eps       ==> always [];
                     left ++ p ==> fun (xs, x) → append xs [x]])
```

```
any [chr 'a' ++ chr 'b' ==> always 1;
     chr 'a' ++ chr 'c' ==> always 2]
```

Rejecting bad grammars

```
any [chr 'a' ==> always 1;
     chr 'a' ==> always 2]
```

Type error: *disjunctive non-determinism*

```
option (chr 'a') ++ option (chr 'a')
```

Type error: *sequential non-determinism*

```
fix (fun left → any [eps       ==> always [];
                     left ++ p ==> fun (xs, x) → append xs [x]])
```

```
any [chr 'a' ++ chr 'b' ==> always 1;
     chr 'a' ++ chr 'c' ==> always 2]
```

Rejecting bad grammars

```
any [chr 'a' ==> always 1;
     chr 'a' ==> always 2]
```

Type error: *disjunctive non-determinism*

```
option (chr 'a') ++ option (chr 'a')
```

Type error: *sequential non-determinism*

```
fix (fun left → any [eps       ==> always [];
                     left ++ p ==> fun (xs, x) → append xs [x]])
```

Type error: *left recursion*

```
any [chr 'a' ++ chr 'b' ==> always 1;
     chr 'a' ++ chr 'c' ==> always 2]
```

Rejecting bad grammars

```
any [chr 'a' ==> always 1;
     chr 'a' ==> always 2]
```

Type error: *disjunctive non-determinism*

```
option (chr 'a') ++ option (chr 'a')
```

Type error: *sequential non-determinism*

```
fix (fun left → any [eps       ==> always [];
                     left ++ p ==> fun (xs, x) → append xs [x]])
```

Type error: *left recursion*

```
any [chr 'a' ++ chr 'b' ==> always 1;
     chr 'a' ++ chr 'c' ==> always 2]
```

Type error: *not left-factored; needs lookahead*

Context-Free Expressions

$$g \quad ::= \quad c \mid \epsilon \mid g \cdot g' \mid \bot \mid g \vee g' \mid x \mid \mu x.\, g$$

Simple denotation

$$\begin{array}{lcl}
[\![c]\!]\,\gamma & = & \{c\} \\
[\![\epsilon]\!]\,\gamma & = & \{\varepsilon\} \\
[\![g \cdot g']\!]\,\gamma & = & \{w \cdot w' \mid w \in [\![g]\!]\,\gamma \wedge w' \in [\![g']\!]\,\gamma\} \\
[\![\bot]\!]\,\gamma & = & \emptyset \\
[\![g \vee g']\!]\,\gamma & = & [\![g]\!]\,\gamma \;\cup\; [\![g']\!]\,\gamma \\
[\![x]\!]\,\gamma & = & \gamma(x) \\
[\![\mu x.\, g]\!]\,\gamma & = & \mathit{fix}(\lambda X.\, [\![g]\!]\,(\gamma, X/x))
\end{array}$$

Nice equational theory — idempotent semiring plus some equations for fixed points:

$$\mu x.\, g = [\mu x.\, g/x]g \qquad\qquad \mu x.\, g_0 \vee x \cdot g_1 = \mu x.\, g_0 \cdot g_1 *$$

Types for context-free languages

Aim: characterize and exclude ambiguity

Types

$$\text{Types } \tau \in \{\text{Null} : 2;\ \text{First} : \mathcal{P}(\Sigma);\ \text{FLast} : \mathcal{P}(\Sigma)\}$$

Predicates on types (example)

$$\tau_1 \# \tau_2 \triangleq (\tau_1.\text{First} \cap \tau_2.\text{First} = \emptyset) \wedge \neg(\tau_1.\text{Null} \wedge \tau_2.\text{Null})$$

Operations on types (example)

$$\tau_1 \vee \tau_2 = \begin{cases} \text{Null} = \tau_1.\text{Null} \vee \tau_2.\text{Null} \\ \text{First} = \tau_1.\text{First} \cup \tau_2.\text{First} \\ \text{FLast} = \tau_1.\text{FLast} \cup \tau_2.\text{FLast} \end{cases}$$

Properties (example)

If $L \models \tau$ and $M \models \tau'$ and $\tau \# \tau'$, then $L \cup M \models \tau \vee \tau'$

A type system for CFGs

$$\Gamma; \Delta \vdash g : \tau$$

$$\frac{}{\Gamma; \Delta \vdash \epsilon : \tau_\epsilon}\ \text{TEps} \qquad \frac{\Gamma; \Delta \vdash g : \tau \qquad \Gamma; \Delta \vdash g' : \tau' \qquad \tau \# \tau'}{\Gamma; \Delta \vdash g \vee g' : \tau \vee \tau'}\ \text{TVee}$$

$$\frac{\Gamma; \Delta, x : \tau \vdash g : \tau}{\Gamma; \Delta \vdash \mu x : \tau . g : \tau}\ \text{TFix}$$

Typing rules also track guardedness to prevent left recursion.

Semantic soundness

If $\Gamma; \Delta \vdash g : \tau$ and $\gamma \models \Gamma$ and $\delta \models \Delta$ then $[\![g]\!]\ (\gamma, \delta) \models \tau$

Unambiguous parse derivations

If $\bullet; \bullet \vdash g : \tau$ then $w \in [\![g]\!]\ \bullet$ iff there is a unique derivation $\mathcal{D} :: g \Rightarrow w$.

Recursive descent parsing for typed grammars

$$\mathcal{P}\,(\Gamma; \Delta \vdash g : \tau) \in \mathrm{Env}(\Gamma) \to \mathrm{Env}(\Delta) \to \Sigma^* \rightharpoonup \Sigma^*$$

$$\mathcal{P}\,(\Gamma; \Delta \vdash g \vee g' : \tau \vee \tau')\ \hat{\gamma}\ \hat{\delta}\ [] =$$

$$\begin{cases} [] & \text{when } (\tau \vee \tau').\mathrm{Null} \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

$$\mathcal{P}\,(\Gamma; \Delta \vdash g \vee g' : \tau \vee \tau')\ \hat{\gamma}\ \hat{\delta}\ ((c :: \_)\ \mathrm{as}\ s) =$$

$$\begin{cases} \mathcal{P}\,(\Gamma; \Delta \vdash g : \tau)\ \hat{\gamma}\ \hat{\delta}\ s & \text{when } c \in \tau.\mathrm{First} \\ & \text{or } \tau.\mathrm{Null} \wedge c \notin (\tau \vee \tau').\mathrm{First} \\ \mathcal{P}\,(\Gamma; \Delta \vdash g' : \tau')\ \hat{\gamma}\ \hat{\delta}\ s & \text{when } c \in \tau'.\mathrm{First} \\ & \text{or } \tau'.\mathrm{Null} \wedge c \notin (\tau \vee \tau').\mathrm{First} \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

Parsing is sound

If a parser succeeds, it consumed a word of L

Parsing is complete

Given a stream prefixed with $w \in L$ and a delimiter, the parser will consume precisely w

First-order representation

Higher-order
programming interface

```
fix (fun x → seq (chr 'a') x)
```

unembedding →

First-order
internal representation

```
Fix (Seq (Chr 'a', Var Z))
```

Representation: a first-order data type with de Bruijn variables, convenient for analysis

```
type (γ, α) t = Eps : (γ, unit) t
              | Seq : (γ, α) t * (γ, β) t → (γ, α * β) t
              | Chr : char → (γ, char) t
              | Bot : (γ, α) t
              | Alt : (γ, α) t * (γ, α) t → (γ, α) t
              | Map : (α → 'b) * (γ, α) t → (γ, 'b) t
              | Fix : (α * γ, α) t → (γ, α) t
              | Var : (γ, α) var → (γ, α) t

type (γ, α) var = Z : (α * γ, α) var
                | S : (γ, α) var → (β * γ, α) var
```

Typechecking in the library

Types are value-level records; the type-checker is a function

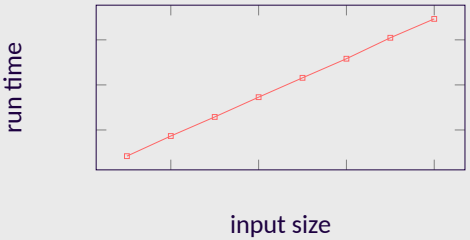
```
type t = { first: CharSet.t;  flast: CharSet.t;  null: bool;  guarded: bool }

val typeof : γ tp_env → (γ, α) grammar → typ
```

Parser combinators use the type information

```
let alt tp1 p1 tp2 p2 s = match Stream.peek s with
  | None   → if tp1.null then p1 s else
             if tp2.null then p2 s else
             error "Stream unexpectedly ended"
  | Some c → if      CharSet.mem c tp1.first then p1 s
             else if CharSet.mem c tp2.first then p2 s
             else if tp1.null                then p1 s
             else if tp2.null                then p2 s
             else error "No progress possible"
```

Performance

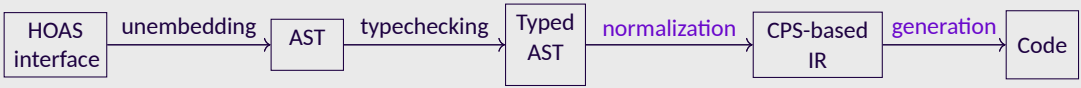
Good news: parser combinators have guaranteed linear-time performance



Bad news: there is a large constant
(Combinators are 4.5–125 times slower than ocamlyacc-generated parsers)

Plan: use multi-stage programming, specializing on the grammar

Staging



Interface unchanged, except for `map`:

$$\texttt{val map : } (\alpha \texttt{ code} \to \beta \texttt{ code}) \to \alpha \texttt{ t} \to \beta \texttt{ t}$$

CPS-based IR (disentangle static & dynamic)

```
let alt t1 p1 t2 p2 s = match Stream.peek s with
  | Some c → if CharSet.mem c t1.first
             then p1 s
             else ...
```

Use types to prune branches

`alt (chr 'a') (chr 'b')`

`alt` calls `peek {'c'}`, `chr` calls `peek {'c'}`; remove 2nd call

The Trick (generate branches)

```
let alt t1 p1 t2 p2 = peek t1.first
 (function `Eof → ...
         | `Yes → ...
         | `No →  ...
```

`let rec` **insertion**

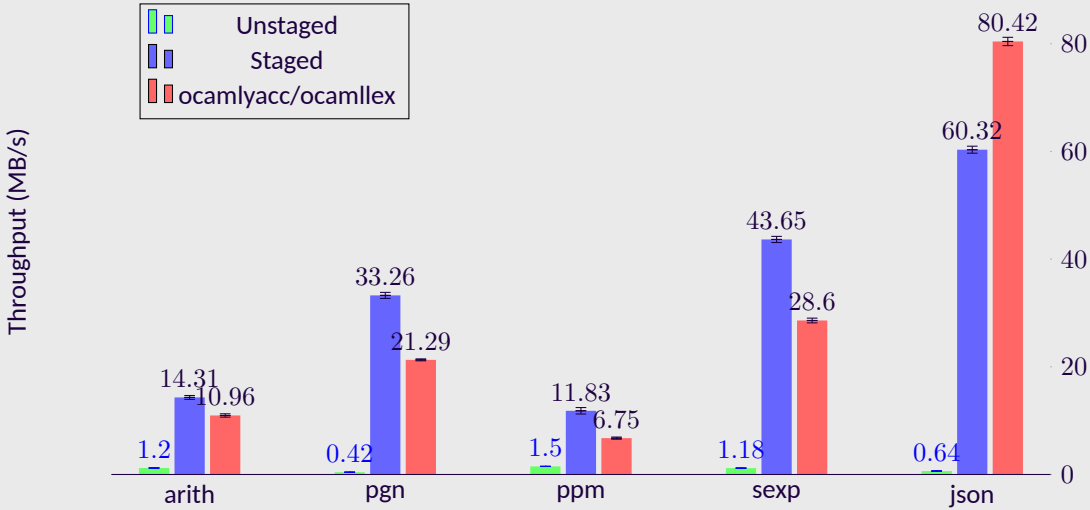
`fix` ⇝ `let rec`

Staged output

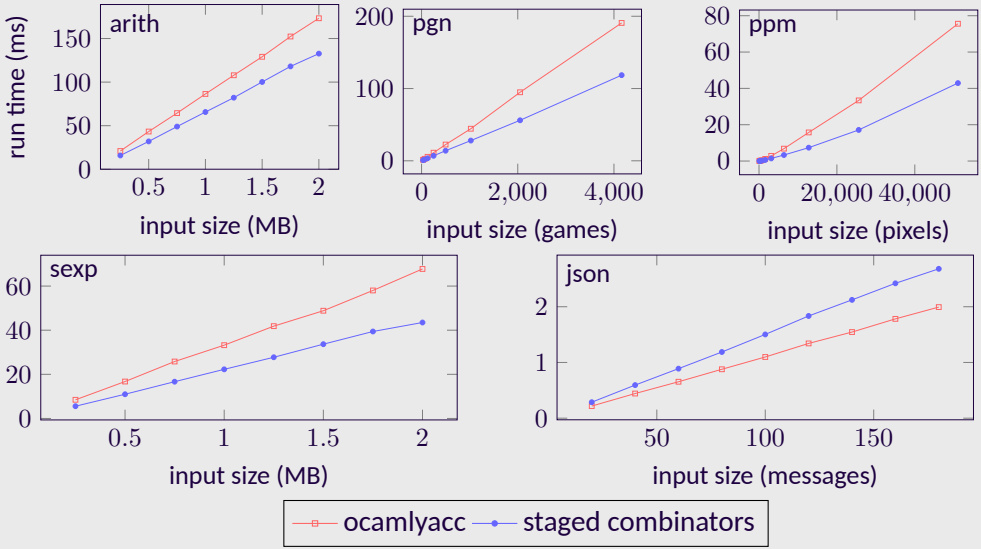
(Note: no backtracking, pointless tests, explicit grammar representation, intermediate structure, etc.)

```
let rec f1 i n s = if i >= n then failwith "Expected chr"
                   else let c1 = s.[i] in
                        if 'a' = c1 then ((), (1 + i))
                        else if '(' = c1 then
                          let (x,j) = f2 (1 + i) n s in
                          if j >= n then failwith "Expected chr"
                          else let c2 = s.[j] in
                               match c2 with ')' → ((), (1 + j))
                                           | _ → failwith "wrong token"
                        else failwith "No progress possible!"
and f2 i n s = if i >= n then ([], i)
               else let c = s.[i] in
                    if ('(' = c) || ('a' = c) then
                      let (x1,j1) = f1 i n s in
                      let (x2,j2) = f2 j1 n s in
                      ((x1 :: x2), j2)
                    else ([], i)
```

Throughput: unstaged vs staged vs ocamllex/ocamlyacc



Everything is linear



Scanners and parsers

We use our combinators to write both a scanner and a parser for each grammar

```
let token = any [
    chr '(' ==> always LPAREN;
    chr ')' ==> always RPAREN;
    word    ==> fun s → SYMBOL s]
```

```
let sexp = fix
   (fun sexp → any [
     symbol            ==> always Sym;
     paren (star sexp) ==> fun s → Seq s])
```

Overhead: materialize tokens from scanner, consume tokens in parser

Cause: scanner & parser compiled separately

Plan: specialize generated code together to eliminate tokens

Hope: perform like scannerless parsers without losing modularity