Fusing Lexing and Parsing

ANONYMOUS AUTHOR(S)

Lexers and parsers are typically defined separately and connected by a token stream. This separate definition is important for modularity, but harmful for performance.

We show how to *fuse* together separately-defined lexers and parsers, drastically improving performance without compromising modularity. Our staged parser combinator library, `flap`, provides a standard parser combinator interface, but generates specialized token-free code that runs several times faster than `ocamlyacc` on a range of benchmarks.

1 INTRODUCTION

Software systems are easiest to understand when their components have clear interfaces that hide internal details. For example, a typical compiler includes separate lexer and parser components that communicate via a token stream.

Unfortunately, while interfaces improve clarity, they can harm performance, since hiding internal details reduces optimization opportunities. Parsers exemplify this tension: the token stream interface isolates parser definitions from character syntax details like whitespace, but it also carries overheads that reduce parsing speed.

Parsers built for efficiency avoid backtracking: only the initial token of the stream is typically needed at any time. However, even with this restriction, materializing and case-switching on tokens comes with a cost.

Contributions. This paper presents a transformation that significantly improves the performance of parsing by entirely eliminating tokens and fusing together lexers and parsers. Specifically, we present lexer-parser fusion using a parser combinator library, `flap` (***f**used **l**exing **a**nd **p**arsing*). The lexers and parsers are built using standard tools: Brzozowski's derivatives [Brzozowski 1964] for lexers (as reformulated by Owens et al. [2009]), and Krishnaswami and Yallop's typed algebraic parser combinators [Krishnaswami and Yallop 2019]. We review these standard tools in Section 2.

We present the following contributions:

- We propose *Deterministic Greibach Normal Form*, a variant of Greibach Normal Form [Greibach 1965] for deterministic languages that captures syntactically the constraints enforced by the types in Krishnaswami and Yallop's typed context-free expressions (Section 3.1). We then formalize a translation from typed context-free expressions into Deterministic Greibach Normal Form, which serves as a basis for follow-up optimizations. We prove that the translation is well-defined and preserves the semantics of context-free expressions (Section 4).
- We present *lexer-parser fusion*, showing how to transform a separately-defined lexer and a normalized parser into a single piece of code that is specialized for calling contexts, entirely avoids materializing tokens, and case-switches only on individual characters, not on intermediate structures (Section 5).
- We implement our techniques as a parser combinator library `flap` and use multi-stage programming to generate efficient token-free code from the fused grammar (Section 6).
- We demonstrate the performance of `flap`, by showing that lexer-parser fusion results in efficient code that runs several times faster than code produced by standard tools such as `ocamllex`, `ocamlyacc` and `menhir`. We also assess other metrics, such as code size and compilation time (Section 7).

Finally, Section 8 surveys related work and Section 9 sets out some directions for further development.

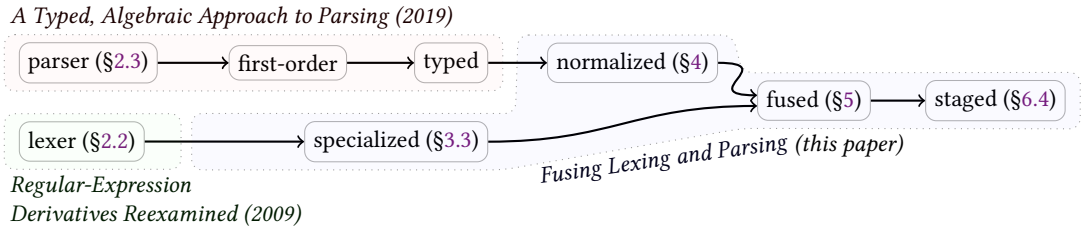


Fig. 1. Architecture of `flap`

2 BACKGROUND: LEXER AND PARSER COMBINATORS

Figure 1 presents the novel code generation architecture of `flap`. As the figure shows, `flap`'s interface is built from standard lexer [Owens et al. 2009] and parser combinators [Krishnaswami and Yallop 2019] drawn from existing work. This section gives an overview of those combinators.

2.1 Derivatives of regular expressions

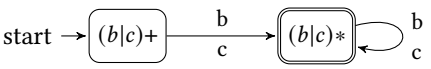
Since lexical syntax is typically regular, lexers are typically defined using regular expressions (*regexes*). One particularly elegant formulation of regex matching, introduced almost six decades ago [Brzozowski 1964], is based on the idea of *derivatives*.

The *derivative* of a regex r with respect to a character c is another regex $\partial_c\ r$ that matches s exactly when r matches $c \cdot s$. For example, the regex $(b|c)+$ matches a sequence of one or more occurrences of b and c in any order. For a string that begins with c, either an empty suffix or some further sequence of b and c is acceptable, and so we have:

$$\partial_c\ (b|c)+ \quad = \quad (b|c)*$$

The full rules, shown in Figure 2, are defined inductively on the syntax of regexes, with cases for the standard constructs $\bot$ (which matches nothing), ϵ (which matches only the empty string), characters b and c, sequencing, alternation, and Kleene star, and for the less commonly-supported constructs intersection and negation. We refer the reader to Owens et al. [2009] for a fuller exposition.

There is a simple relationship between derivatives and automata for regular expressions: one way to construct an automaton is to take regular expressions r as states, and add a transition from r_i to r_j via character c whenever $\partial_c\ r_i = r_j$. For example, here is an automaton for $(b|c)+$:



In this example, the transitions all target the same state, since $\partial_b\ (b|c)+ = \partial_c\ (b|c)+ = \partial_b\ (b|c)* = \partial_c\ (b|c)* = (b|c)*$. Additionally, since that state also accepts the empty string, it is marked as an accepting state for the whole automaton.

Constructing an automaton is a common way to implement a regex matcher, and derivatives make it straightforward to built an automaton that is deterministic and compact. This process often involves representing the automaton as a graph or table. Alternatively, *multi-stage programming* [Taha 1999], makes it possible to directly generate code embodying the automaton.

Regex matching is an archetypal example of staged computation [Davies and Pfenning 1996] as found in languages like BER MetaOCaml [Kiselyov 2014]: although matching is a function of two inputs, regex and string, since the former is typically available first, it can be used to construct specialized code for processing the latter. In other words, while an unstaged matcher might have the following two-argument OCaml type

$$\text{regular expression } r ::= \bot \mid \epsilon \mid c \mid r \cdot s \mid (r \mid s) \mid r* \mid (r \,\&\, s) \mid \neg r$$

$$\begin{array}{llll} \partial_c \bot = \bot & \partial_c c = \epsilon & & \partial_c r* = \partial_c r \cdot r* \\ \partial_c \epsilon = \bot & \partial_c (r \cdot s) = \partial_c r \cdot s \mid \nu(r) \cdot \partial_c s & \partial_c (r \,\&\, s) = \partial_c r \,\&\, \partial_c s & \\ \partial_c b = \bot & \partial_c (r \mid s) = \partial_c r \mid \partial_c s & \partial_c \neg r = \neg(\partial_c r) & \end{array}$$

Fig. 2. Derivatives of regular expressions (The *nullability* function $\nu(r)$ expands to ϵ if r matches ϵ and $\bot$ otherwise).

```
val matchr : regex → string → bool
```

a staged matcher is instead a function of one argument which generates code for another function of one argument:

```
val smatchr : regex → (string → bool) code
```

For the regex $(b|c)+$, `smatchr` might generate code found in the corresponding automaton:

```
≪ let rec bcplus s i = (i <> length s) &&                    (* (b|c)+ *)
    match s.[i] with 'b'|'c' → bcstar s (i+1)
                   | _ → false
  and bcstar s i = (i = length s) ||                         (* (b|c)* *)
    match s.[i] with 'b'..'c' → bcstar s (i+1)
                   | _ → false
  in fun s → bcplus s 0 ≫
```

The quotation notation $\ll e \gg$ in this example indicates a typed code value; it is one of MetaOCaml's two fundamental constructs (along with antiquotation ~e) for staging. The function for each regex r and string s follows a simple pattern, returning **`true`** if r is nullable and s is empty, and moving to the function for $\partial_c\ r$ if s begins with c.

2.2 Lexing with derivatives

Regexes built from derivatives are convenient for building lexers. A lexer is typically defined as an ordered mapping from regular expressions to *actions*, where an action might return a token, raise an error or invoke the lexer recursively to skip over some input. Figure 3a gives an example lexer with four actions: three of which return tokens atom, lpar and rpar, and one of which skips over whitespace.

Using MetaOCaml it is straightforward to express lexers as functions that accept lists of regex-action pairs and return code:

```
type 'a action = Skip | Return of 'a code
val slex : (re * 'a action) list →(string → 'a) code
```

In practice, it is useful to extend these definitions with additional actions (such as `Error`) and additional information (such as the position of matched strings), which we omit here for succinctness.

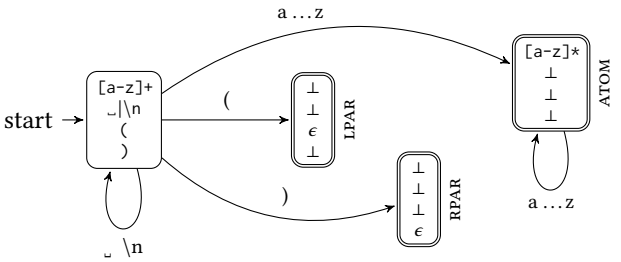
Regex derivatives extend naturally to lexers by matching the input string against multiple regexes in parallel. Figure 3b is the automaton for matching one token with the sexp lexer, where each state corresponds to a vector of regexes, one for each lexer rule. The transition function ∂_c acts pointwise on the regex vector. **Return** rules correspond to labeled accepting states, and the **Skip** rule resets the vector to its initial state.

As Owens et al. [2009] show, lexers based on derivatives provide a practical basis for real-world lexing tools such as ml-ulex and the PLT Scheme scanner generator. One particularly useful feature for implementing lexers is the support derivatives provide for negation and disjunction, which make

id ⇒ **Return** ATOM
space ⇒ **Skip**
(⇒ **Return** LPAR
) ⇒ **Return** RPAR

id $\stackrel{\text{def}}{=}$ [a-z]+
space $\stackrel{\text{def}}{=}$ ␣ | \n

(a) S-expression lexer



(b) An automaton for matching one token

Fig. 3. Example: S-expression lexer and an automaton

it straightforward to transform implicitly-ordered clauses for regexes r and s into order-independent disjoint clauses for r and $\neg(r)\&s$ with the same semantics.

2.3 Parsing with typed context-free expressions

Parser combinators, introduced almost four decades ago by Wadler [1985], provide an elegant way to define parsers using functions. A parser combinator library provides functions denoting token-matching, sequencing, disjunction, and so on, allowing the library user to describe a parser by combining these functions in a way that reflects the structure of the corresponding grammar. Here are partial interfaces for constructing both regexes (type `re`) and parsers (type `pa`) in this way:

```
(* Regex combinators *)                    (* Parser combinators *)
type re                                     type 'a pa
val chr: char → re       (* token match *) val tok: 'a tok → 'a pa
val (>>>): re → re → re (* sequence  *)    val (>>>): 'a pa → 'b pa → ('a * 'b) pa
val star: re → re         (* recursion *)  val fix: ('a pa → 'a pa) → 'a pa
...                          ...            ...
```

Both interfaces provide functions for token matching, sequencing and recursion. However, there are some important differences: first, regexes act on characters, while parsers act on tokens (type `tok`, a parameter of the library); second, parsers provide a general-purpose recursion operator `fix`, while regexes offer only the more restrictive Kleene `star`; finally, the parser type is parameterized, allowing parsers to construct and return suitably-typed syntax trees.

The earliest parser combinator libraries represented nondeterministic parsers, with support for arbitrary backtracking and multiple results. Parsers defined in this way enjoyed various pleasant properties (such as a rich equational theory), but suffered from potentially disastrous performance. In a recent departure from the nondeterministic tradition, Krishnaswami and Yallop [2019] define *typed context-free expressions*, whose types track properties such as FIRST sets and nullability in order to preclude nondeterminism and ensure linear-time parsing using a single token of lookahead. Krishnaswami and Yallop's design provides the standard set of parser combinators (as defined above), but adds an additional type-checking step. They further apply multi-stage programming to ensure that type-checking is completed before parsing begins, and to generate specialized parsing code based on type information, leading to performance competitive with `ocamlyacc`.

Figure 4 summarizes Krishnaswami and Yallop's type system. A type is a triple recording nullability, the first set, and FLAST (analogous to the FOLLOW set). There is one typing rule for each combinator (e.g. sequencing $g_1 \cdot g_2$ and recursion $\mu\alpha : \tau.g$); types are constructed using corresponding combinators (e.g. $\tau_1 \cdot \tau_2$). The two contexts Γ and Δ restrict the positions in which variables can occur to disallow left recursion, and the side conditions *separation* $\tau_1 \circledast \tau_2$ and *apartness* $\tau_1 \# \tau_2$

Context-free expression $g \quad ::= \quad \epsilon \mid t \mid \bot \mid \alpha \mid g_1 \cdot g_2 \mid g_1 \vee g_2 \mid \mu\alpha : \tau . g$

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

$$\tau_\epsilon = \{\textsc{Null} = \text{true};\ \textsc{First} = \emptyset;\ \textsc{FLast} = \emptyset\}$$
$$\tau_t = \{\textsc{Null} = \text{false};\ \textsc{First} = \{t\};\ \textsc{FLast} = \emptyset\}$$
$$\tau_\bot = \{\textsc{Null} = \text{false};\ \textsc{First} = \emptyset;\ \textsc{FLast} = \emptyset\}$$

$$\tau_1 \cdot \tau_2 = \begin{cases} \textsc{Null} = \tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null} \\ \textsc{First} = \tau_1.\textsc{First} \cup \tau_1.\textsc{Null}\,?\,\tau_2.\textsc{First} \\ \textsc{FLast} = \tau_2.\textsc{FLast} \cup \tau_2.\textsc{Null}\,?\,(\tau_2.\textsc{First} \cup \tau_1.\textsc{FLast}) \end{cases}$$

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

$$\tau_1 \circledast \tau_2 \stackrel{\text{def}}{=} \tau_1.\textsc{FLast} \cap \tau_2.\textsc{First} = \emptyset \wedge \neg\tau_1.\textsc{Null}$$
$$\tau_1 \,\#\, \tau_2 \stackrel{\text{def}}{=} (\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset) \wedge \neg(\tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null})$$
$$b\,?\,S \stackrel{\text{def}}{=} \text{if } b \text{ then } S \text{ else } \emptyset$$

$$\frac{}{\Gamma;\Delta \vdash \epsilon : \tau_\epsilon} \qquad \frac{}{\Gamma;\Delta \vdash t : \tau_t}$$

$$\frac{}{\Gamma;\Delta \vdash \bot : \tau_\bot} \qquad \frac{\alpha : \tau \in \Gamma}{\Gamma;\Delta \vdash \alpha : \tau}$$

$$\frac{\Gamma;\Delta,\alpha : \tau \vdash g : \tau}{\Gamma;\Delta \vdash \mu\alpha : \tau . g : \tau}$$

$$\frac{\Gamma;\Delta \vdash g_1 : \tau_1 \quad \Gamma,\Delta;\bullet \vdash g_2 : \tau_2 \quad \tau_1 \circledast \tau_2}{\Gamma;\Delta \vdash g_1 \cdot g_2 : \tau_1 \cdot \tau_2}$$

$$\frac{\Gamma;\Delta \vdash g_1 : \tau_1 \quad \Gamma;\Delta \vdash g_2 : \tau_2 \quad \tau_1 \,\#\, \tau_2}{\Gamma;\Delta \vdash g_1 \vee g_2 : \tau_1 \vee \tau_2}$$

Fig. 4. Krishnaswami and Yallop's type system for context-free expressions

on the rules for sequencing and alternation reject ambiguous grammars, ensuring respectively that strings matched by sequenced parsers have a unique decomposition, and that the languages matched by alternated parsers do not overlap.

As an example, consider the following well-typed s-expression (we often omit τ in $\mu\alpha : \tau . g$):

$$\mu\ \text{sexp}\ .(\textsc{lpar} \cdot (\mu\ \text{sexps}\ .\ \epsilon \vee\ \text{sexp}\ \cdot\ \text{sexps}\ ) \cdot \textsc{rpar}) \vee \textsc{atom}$$

The following code shows how to use Krishnaswami and Yallop's parser combinators to define this grammar, using a token type with LPAR, RPAR and ATOM constructors, with the explicit fixed point represented using the Kleene star:

```
fix (fun sexp → (tok LPAR >>> star sexp >>> tok RPAR)
                  $ fun p →≪ Sexp (snd (fst ~p)) ≫
                <|> tok ATOM $ fun s →≪ Atom ~s ≫)
```

The `fix`, `tok`, `>>>` and `star` constructors are from the parser interfaces. The example additionally uses alternation `<|>` and map `$`, which transforms the parsing result via a user-defined function, and MetaOCaml's quotation and antiquotation constructs ≪ *exp* ≫ and *~exp* to build code values.

3 OVERVIEW

The algorithm for parsing with typed context-free expressions introduced by Krishnaswami and Yallop is efficient at a high level, since it uses only a single token of lookahead and its execution time is linear in the length of its input. However, it is less efficient at a low level, since it examines each token multiple times: once at each alternation in the grammar, and then once again at the token combinator.

As an example, consider again μ sexp .(LPAR · (μ sexps . $\epsilon \vee$ sexp · sexps) · RPAR) $\vee$ ATOM. In the sub-grammar $\epsilon \vee$ sexp · sexps, neither alternative explicitly matches a token. Determining which branch to take therefore requires analysing the types to calculate whether the next tokens

in the input fall into the First set of either ϵ or sexp $\cdot$ sexps. Furthermore, once the token has been examined and the branch taken, the parsing algorithm must examine it a second time. For example, since lpar $\in$ First(sexp $\cdot$ sexps), the token lpar causes the parsing algorithm to switch to the sexp $\cdot$ sexps branch. After the switch, parsing again uses the typing information for the selected branch to determine which of sexp's sub-branches to take (i.e. lpar or atom). Eventually, the parsing algorithm encounters the lpar node in the grammar, and the token is consumed. To address these low-level inefficiencies, Krishnaswami and Yallop applied a variety of multi-stage programming techniques, such as CPS conversion [Bondorf 1992; Nielsen and Sørensen 1995] to improve the results of staging, ultimately achieving performance that is competitive with `ocamllex` and `ocamlyacc`.

In this work, we take a more systematic approach, making use of the guarantees offered by the types to transform grammars into a normal form that is amenable to a sequence of further optimizations. More precisely,

(1) We first propose a novel normal form, *Deterministic Greibach Normal Form (DGNF)*, which gathers together the places in the grammar that involve branching on tokens, allowing tokens to be discarded immediately after inspection (Section 3.1).
(2) We then formalize a normalization algorithm that traverses a context-free expression and returns a DGNF grammar. Normalization works well for well-typed context free expressions, and the resulting DGNF grammar sets the basis for follow-up optimizations (Section 3.2).
(3) Based on the normal form, we present a fusion process that ultimately eliminates the need to materialize tokens altogether. The fusion algorithm starts with a separately-defined lexer and parser, connected together via tokens, and produces entirely token-free code, in which the only branches involve inspecting individual characters (Section 3.3).
(4) Finally, `flap` uses MetaOCaml's staging facilities to generate code for the fused grammar. Since the normalized grammar representation is already amenable to generating optimized code, `flap` does not need the sophisticated techniques used by Krishnaswami and Yallop [2019] (Section 3.4).

As we shall see, these optimizations make parsers built from typed context-free expressions significantly more efficient than both `ocamlyacc` and Krishnaswami and Yallop's system (Section 7).

The running example. This section illustrates `flap`'s key ideas through a running example given in Figure 5. Figure 5a presents the grammars that will be used throughout this paper. Figure 5b and 5c repeat our previous s-expression lexer and example grammar respectively for better readability.

From now on we use colors to distinguish different grammars. Regular expressions r include $\bot$ for nothing, ϵ for the empty string, characters c, sequencing $r \cdot s$, alternation $r \mid s$, Kleene star $r*$, intersection $r \;\&\; s$, and negation $\neg r$. Lexers L are a set of regex-action pairs where each action either returns a token or skips. Context-free expressions g are $\bot$ for the empty language, ϵ for the language containing only the empty string, t which matches the language containing only the single-element string t, variables α, sequences $g_1 \cdot g_2$, unions $g_1 \vee g_2$, and the least fixed point operator $\mu\alpha : \tau.g$. We will introduce the normal form grammar G and fused grammar F later.

3.1 Deterministic Greibach Normal Form

Like Krishnaswami and Yallop's system, `flap` first ensures that input grammars are well-typed according to the rules in Figure 4. It then applies a normalization algorithm that transforms well-typed grammars into a novel normal form that avoids the need for repeated branching.

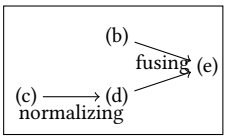
Specifically, to ensure that grammars can be used for deterministic parsing with a single token of lookahead, we introduce *Deterministic Greibach Normal Form* (DGNF), a variant of *Greibach Normal Form* (GNF) [Greibach 1965]. More precisely, in GNF, all the productions of a grammar take the

regular expression	r	::=	$\bot \mid \epsilon \mid c \mid r \cdot s \mid (r \mid s) \mid r* \mid (r \,\&\, s) \mid \neg r$
lexer	L	::=	$\{\, r \Rightarrow \textbf{Return}\ t \,\} \cup \{\, r \Rightarrow \textbf{Skip} \,\}$
context-free expression	g	::=	$\bot \mid \epsilon \mid t \mid \alpha \mid g_1 \cdot g_2 \mid g_1 \vee g_2 \mid \mu\alpha : \tau.\, g$
normal form	N	::=	$\epsilon \mid t\ \overline{n} \mid \alpha\ \overline{n}$
normal form grammar	G	::=	$\{\, n \longrightarrow N \,\}$
fused grammar	F	::=	$\{\, n \longrightarrow r\ \overline{n} \,\} \cup \{\, n \longrightarrow ?r \,\}$

(a) Syntax of lexers, forms, and grammars in `flap`

```
id    ⇒ Return ATOM
space ⇒ Skip
(     ⇒ Return LPAR
)     ⇒ Return RPAR

id    =def= [a-z]+
space =def= ␣ | \n
```



(b) S-expression lexer (2.2)

$$\mu\ \text{sexp}\ .(\textsc{lpar} \cdot (\mu\ \text{sexps}\ .\ \epsilon \vee\ \text{sexp}\ \cdot\ \text{sexps}\ ) \cdot \textsc{rpar}) \vee \textsc{atom}$$

(c) A well-typed s-expression grammar (2.3)

```
sexp ::= LPAR sexps rpar | ATOM
rpar ::= RPAR
sexps ::= LPAR sexps rpar sexps | ATOM sexps | ϵ
```

(d) The above s-expression grammar in Deterministic Greibach Normal Form (3.2)

```
id    ⇒ Return ATOM
space ⇒ Skip
(     ⇒ Return LPAR
)     ⇒ Return RPAR

sexp ::= ( sexps rpar | id | space sexp
rpar ::= ) | space rpar
sexps ::= ( sexps rpar sexps | id sexps | space sexps | ?¬(id | space | ()
```

(e) Fusing drops lexing rules that return non-matchable tokens (top); the fused s-expr grammar (bottom) (3.3)

Fig. 5. `flap`: running example of an s-expression. Grammars in the example are written in BNF form.

form $n \longrightarrow tn_1n_2 \ldots n_k (k \geq 0)$ where n and n_i are nonterminals and t is a terminal. DGNF further imposes the following syntactic constraints:

Definition 1 (Deterministic Greibach Normal Form (Syntax)). *A grammar G is in Deterministic Greibach normal form if all productions are of form $n \longrightarrow tn_1n_2 \ldots n_k (k \geq 0)$, and moreover,*

- *(Determinism) for any pair of a nonterminal n and a terminal t, there is at most one production beginning $n \longrightarrow tn_1n_2 \ldots n_k$;*

- *(Guarded ϵ-productions) Additional, each nonterminal may have an optional ϵ-production $n \longrightarrow \epsilon$, with the proviso that the ϵ-production may only be used when no terminal symbol in the non-ϵ productions matches the input string.*

Figure 5a includes the definition of normal form and normal form grammar, where normal form N is either an epsilon ϵ or a terminal followed by a list of nonterminals $t\ \overline{n}$, with $\overline{n}$ denoting $n_1 n_2 \ldots n_k (k \geq 0)$. Note that N also includes a special internal form $\alpha\ \overline{n}$, highlighted with gray, which is explained in Section 3.2. The normal form grammar G is a set of productions that map nonterminals to normal forms.

Intuitively speaking, the constraints on the DGNF grammar are a syntactic analogue of the constraints enforced by the types in Krishnaswami and Yallop's typed context-free expressions. The constraints have a simple practical motivation in parsers. That is, each $n \longrightarrow t n_1 n_2 \cdots n_k$ production represents one branch that matches a distinct terminal t, and guarded ϵ-productions represent an else branch that is taken if none of the active productive branches matches the input.

Examples. We consider a few examples. For readability, we write the grammar in BNF form, e.g., $n ::= \textsc{A}\, n_1 n_2 \mid \textsc{B}$ corresponds to $n \longrightarrow \textsc{A}\, n_1 n_2$ and $n \longrightarrow \textsc{B}$.

(1)
$$\begin{array}{lcl} n & ::= & \textsc{A}\, n_1 n_2 \\ & \mid & \textsc{B} \\ n_1 & ::= & \textsc{C} \\ n_2 & ::= & \textsc{D} \end{array}$$

(2)
$$\begin{array}{lcl} n & ::= & \textsc{AB}\, n_1 \\ n_1 & ::= & \textsc{C} \end{array}$$

(3)
$$\begin{array}{lcl} n & ::= & \textsc{A}\, n_1 \\ & \mid & \textsc{A}\, n_2 \\ n_1 & ::= & \textsc{C} \\ n_2 & ::= & \textsc{D} \end{array}$$

(4)
$$\begin{array}{lcl} n & ::= & \textsc{A}\, n_1 n_2 \\ n_1 & ::= & \textsc{C} \\ & \mid & \epsilon \\ n_2 & ::= & \textsc{C} \end{array}$$

Here (1) is in DGNF, while (2) (3) are not. In (2), n starts with two terminals, while in (3), n has two productions starting with A, violating the determinism condition.

(4) is the most subtle case. Consider matching n with AC. First, n expands to $\textsc{A}\, n_1 n_2$. But should n_1 then expand to C or ϵ? In a general nondeterministic grammar, it is impossible to tell simply by looking ahead at the next token C: we may first consider $n_1 \longrightarrow \textsc{C}$ and, finding that n_2 fails to match, backtrack to the other branch to consider $n_1 \longrightarrow \epsilon$ and $n_2 \longrightarrow \textsc{C}$ and succeed. However, the proviso in Definition 1 eliminates this choice: only $n_1 \longrightarrow \textsc{C}$ applies, and so the grammar does *not* match AC. In fact, the semantics of DGNF (Section 4.2) will rule out (4) as a DGNF grammar, ensuring that parsing is deterministic.

As the examples demonstrate, the determinism and guarded ϵ-production conditions ensure that there is never any ambiguity about whether a production rule applies during parsing.

3.2 Normalization

We formalize a normalization algorithm, which, given any well-typed context-free expression, turns it into a DGNF grammar. As an example, Figure 5d presents the result in BNF form of normalizing the s-expression grammar in Figure 5c. It is straightforward to check that the normalized grammar represents exactly the same language as the original context-free expression.

Importantly, the normalized DGNF presentation addresses the problem of repeated branching discussed in the beginning of Section 3. With this normalized form, parsing a sexps involves reading the next token, and branching to the first, second or third branch depending on whether the token is LPAR, ATOM or something else. In the first two cases the token is consumed immediately, and parsing moves on to the next token in the input. The last case is somewhat more costly, since the token may be needed again immediately afterwards, and more care is needed to avoid wasted work.

Section 4 discusses the normalization algorithm, which traverses a context-free expression, and builds grammar productions according to the expression structure. As we will see, defining the algorithm poses significant challenges, particularly around fixed points. When normalizing $\mu\alpha.\ g$, although we do not yet know the normalized grammar for α, we must proceed with normalizing g

regardless. In this case, it is necessary to "tie the knot" when the result of normalizing g becomes available, which requires us to introduce an intermediate non-DGNF form $n \longrightarrow \alpha\ \overline{n}$, causing complication and subtleties during normalization. This non-DGNF form is purely internal, and does not appear in the normalization results of closed expressions.

Finally, great care needs to be taken to guarantee that normalization produces indeed DGNF grammars. That requires us to ensure that the normalization captures the constraints enforced by the types in Krishnaswami and Yallop's system. We prove that normalization is correct, and that normalized grammars preserve the denotational semantics of context-free expressions (Section 4).

3.3 Fusion

Next, `flap` applies lexer-parser fusion, one of our central contributions. Fusion acts on a lexer and a normalized parser, connected together via tokens, and products a grammar representation that is entirely token-free, in which the only branches involve inspecting individual characters.

Figure 5a defines the syntax of fused grammars, where the fused form f is either a regex followed by a list of nonterminals $r\ \overline{n}$, or a single-token lookahead $?r$ for tokens matched by r. The fused grammar F is a set of productions $\{\ n \longrightarrow f\ \}$.

We illustrate the key idea of fusion through Figure 5e, which fuses the s-expression lexer in Figure 5b and the normalized parser in Figure 5d. As the first step, the fusion algorithm implicitly specializes the lexer to each nonterminal n in the normalized grammar, and lexing rules that return tokens not in productions for the nonterminal n are discarded. We take rpar as an example: the rpar nonterminal has only a single production, which begins with the terminal RPAR. We then look at the lexing rules, and discard those rules that do not return RPAR. However, the skip rule is retained, since skipped characters can precede any token. Then, the algorithm fuses the lexing rules and the parsing rules, by substituting the tokens in the parsing rules by regexes in the lexing rules that return corresponding tokens. The bottom of Figure 5e presents the fused grammar for rpar, which has two branches. The first branch fuses lexing and parsing, by having the original token RPAR replaced with the regex `)`. The second branch is an extra production corresponding to the skip rule in the lexer, allowing rpar to match an arbitrary number of space. Notably, after fusion rpar now directly matches space or `)`, without referring to the token RPAR.

The bottom of Figure 5e presents the complete result of fusing the s-expression lexer and normalized grammar. Like the case for rpar, the tokens ATOM, LPAR and RPAR in the grammar have been replaced with the regular expressions `id`, `(` and `)` associated with those tokens in the lexer. Moreover, for each nonterminal n there is an extra production $n ::= \texttt{space}\ n$, corresponding to the skip rule in the lexer. Finally, the ϵ-production sexps $\longrightarrow \epsilon$ has been replaced with a lookahead rule sexp $\longrightarrow ?\neg(\texttt{id} \mid \texttt{space} \mid \texttt{(})$, consisting of the complement of the three regular expressions that appear at the start of the right hand side of the other productions.

Normalization allows the fusion algorithm to be defined concisely (Section 5). In particular, the constraints on the positions of terminals make it straightforward to fuse the lexing rules into the grammar without disrupting its structure. More importantly, the fused grammar inherits the properties of DGNF: the productions of a nonterminal start with distinct regexes, and an optional lookahead rule may only be used when no regexes in other productions match the input string.

In summary, fusion transforms a separately-defined lexer and a normalized parser into branches on individual characters, entirely eliminating intermediate tokens. As Section 7 shows, fusion significantly improves the performance of parsing. Furthermore, Section 7.3 suggests that the size increase resulting from normalization and fusion in `flap` is relatively modest, and Section 7.4 reports that compilation times are sufficiently low for interactive use.

normal form	N	$::=$	$\epsilon \mid t\ \overline{n} \mid \alpha\ \overline{n}$
normal form grammar	G	$::=$	$\{\, n \to N \,\}$

$\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow G$, with a grammar G and the start nonterminal n
Each rule allocates a fresh nonterminal n, except for rule (*fix*), which allocates a fresh α

$$
\begin{array}{lll}
(\textit{epsilon}) & \mathcal{N}[\![\, \epsilon \,]\!] & = n \Rightarrow \{\, n \to \epsilon \,\} \\
(\textit{token}) & \mathcal{N}[\![\, t \,]\!] & = n \Rightarrow \{\, n \to t \,\} \\
(\textit{bot}) & \mathcal{N}[\![\, \bot \,]\!] & = n \Rightarrow \emptyset \\
(\textit{seq}) & \mathcal{N}[\![\, g_1 \cdot g_2 \,]\!] & = n \Rightarrow \{\, n \to N_1\ n_2 \mid n_1 \to N_1 \in G_1 \,\} \cup G_1 \cup G_2 \\
 & & \quad \text{where} \quad \mathcal{N}[\![\, g_1 \,]\!] = n_1 \Rightarrow G_1 \wedge \mathcal{N}[\![\, g_2 \,]\!] = n_2 \Rightarrow G_2 \\
(\textit{alt}) & \mathcal{N}[\![\, g_1 \vee g_2 \,]\!] & = n \Rightarrow \{\, n \to N_1 \mid n_1 \to N_1 \in G_1 \,\} \cup \{\, n \to N_2 \mid n_2 \to N_2 \in G_2 \,\} \cup G_1 \cup G_2 \\
 & & \quad \text{where} \quad \mathcal{N}[\![\, g_1 \,]\!] = n_1 \Rightarrow G_1 \wedge \mathcal{N}[\![\, g_2 \,]\!] = n_2 \Rightarrow G_2 \\
(\textit{fix}) & \mathcal{N}[\![\, \mu\alpha.\ g \,]\!] & = \alpha \Rightarrow \{\, \alpha \to N \mid n \to N \in G \,\} \qquad (1) \\
 & & \qquad \cup \{\, n' \to N\ \overline{n}' \mid n' \to \alpha\ \overline{n}' \in G \wedge n \to N \in G \,\} \qquad (2) \\
 & & \qquad \cup G \backslash_{n' \to \alpha\ \overline{n}'} \qquad (3) \\
 & & \quad \text{where} \quad \mathcal{N}[\![\, g \,]\!] = n \Rightarrow G \\
 & & \qquad G \backslash_{n' \to \alpha\ \overline{n}'} \text{ is } G \text{ with all } n' \to \alpha\ \overline{n}' \text{ removed for any } n' \text{ and } \overline{n}' \\
(\textit{var}) & \mathcal{N}[\![\, \alpha \,]\!] & = n \Rightarrow \{\, n \to \alpha \,\}
\end{array}
$$

Fig. 6. Normalization of well-typed context-free expressions.

3.4 Staging

In the last step, `flap` uses MetaOCaml's staging facilities to generate code for the fused grammar. The normalized grammar representation used in `flap` makes this process comparatively straightforward; it does not involve sophisticated optimization techniques such as the binding-time improvements applied by Krishnaswami and Yallop [2019]. Furthermore, `flap` does not rely on compiler optimizations to further simplify the code it generates; instead, it directly generates efficient code, containing no indirect calls, no higher-order functions and no allocation, except where these elements are inserted by the user of `flap` in semantic actions.

Specifically, the staging step in `flap` generates one function for each parser state (i.e. for each pair of a nonterminal and a regex vector), following a parsing algorithm with fused grammars, but eliminating information that is statically known, such as the nullability and derivatives of the regexes associated with each state.

Section 6 presents the algorithm underlying `flap`'s staged parsing implementation, and Section 6.5 describes the implementation itself in more detail.

4 NORMALIZING CONTEXT-FREE EXPRESSIONS

This section presents a normalization algorithm that transforms context-free expressions into grammars in Deterministic Greibach Normal Form (DGNF). The normalization sets the basis for follow-up optimizations of fusion and staging.

4.1 Normalization to DGNF

Figure 6 defines a normalization algorithm for typed context-free expressions. We repeat here the syntax of normal forms and normal form grammars. As discussed in Section 3.2, normal forms

include a non-DGNF internal form $\alpha\,\overline{n}$ used when normalizing fixpoints, where α is interpreted as a special kind of nonterminals. Since α is a nonterminal, α itself may appear as part of a $t\,\overline{n}$ (e.g., $t\,\alpha$). Because of this internal form, normal forms N are actually *not* in DGNF. But as we will show later, $\alpha\,\overline{n}$ is an intermediate form which will be entirely eliminated in the final result, eventually turning the grammar into DGNF.

The key to normalization is the function $\mathcal{N}[\![\, g \,]\!]$ that normalizes a context-free expression g, yielding a normalized grammar G and a distinguished start nonterminal n of the grammar.

There are seven cases for the seven context-free expression constructors. Each case involves allocating a fresh nonterminal (n or α) to use as the start symbol. The cases with sub-expressions ($g_1 \cdot g_2$, $g_1 \vee g_2$ and $\mu\alpha.\ g$) are defined compositionally in terms of the normalization of those sub-expressions. Since normalization simply merges together all the production sets resulting from sub-expressions, the situation frequently arises where some productions are not reachable from the start symbol; the definition here ignores this issue, since it is easy to trim unreachable productions in the implementation.

Rules *(epsilon)*, *(token)*, and *(bot)* are straightforward. For each of ϵ and c, normalization produces a grammar with a single production whose right-hand side is ϵ or c respectively. For $\bot$, normalization produces an empty grammar, with a start symbol and no productions.

Rule *(seq)* is defined compositionally in terms of the normalization of their sub-expressions. For sequencing $g_1 \cdot g_2$, normalizing g_1 and g_2 produces the start symbol n_1 and n_2, respectively. Now what we want is $n \longrightarrow n_1\ n_2$. That is:

$$\textit{(seq1)} \quad \mathcal{N}[\![\, g_1 \cdot g_2 \,]\!] \;=\; n \Rightarrow \{n \longrightarrow n_1\ n_2\} \cup G_1 \cup G_2 \qquad \text{where} \quad \mathcal{N}[\![\, g \,]\!] = n \Rightarrow G$$

However, while this is semantically correct, $n \longrightarrow n_1\ n_2$ is not in normal form. Therefore, normalization in rule *(seq)* copies each production N_1 of n_1 and appends to each the start symbol n_2, producing $N_1\ n_2$. To see why that is correct, consider if N_1 is of form $t\,\overline{n}$ or $\alpha\,\overline{n}$. Then $N_1\ n_2$, i.e., $t\,\overline{n}\ n_2$ or $\alpha\,\overline{n}\ n_2$, is indeed of normal form. However, we need to prevent N_1 from being ϵ, or otherwise $\epsilon\ n_2$ would be ill-formed. The case for sequencing is one of several places where the correctness of normalization depends on the types. In particular, if $g_1 \cdot g_2$ is well-typed, then N_1 cannot be ϵ. Specifically, the typing rule for sequencing (Figure 4) depends on the separation relation $\tau_1 \circledast \tau_2$, which guarantees that the Null of g_1 is false. With that guarantee, we show that N_1 cannot be ϵ (Lemma 4.2), ensuring that the result of the normalization function is in normal form.

Rule *(alt)* for alternation $g_1 \vee g_2$ is similar. In this case, normalization merges the productions for the start symbols n_1 of g_1 and n_2 of g_2 into the productions for the new start symbol n. During typing (Figure 4), the apartness relation $\tau_1 \,\#\, \tau_2$ ensures that the First sets of g_1 and g_2 do not intersect, and that at most one of g_1 and g_2 is nullable. This guarantees that the result is well-formed.

The final two rules *(fix)* and *(var)* deal with the binding fixed point operator $\mu\alpha.\ g$ and with bound variables α. In rule *(fix)*, we assume we can always alpha-rename a fixed point so α is unique. Normalization for the fixed point operator $\mu\alpha.\ g$ takes place in two stages. First, the body g is normalized. The normalization result suggests that the grammar of the body g has a start symbol n. Then, according to the semantics of fixed point, we should proceed to tie the knot by producing $\alpha \longrightarrow n$ and return α as the start symbol. That is:

$$\textit{(fix1)} \quad \mathcal{N}[\![\, \mu\alpha.\ g \,]\!] \;=\; \alpha \Rightarrow \{\alpha \longrightarrow n\} \cup G \qquad \text{where} \quad \mathcal{N}[\![\, g \,]\!] = n \Rightarrow G$$

However, the rule $\alpha \longrightarrow n$ is not in normal form. Therefore, instead of directly returning $\alpha \longrightarrow n$, we proceed to copy the productions for n into the rules for α:

$$\textit{(fix2)} \quad \mathcal{N}[\![\, \mu\alpha.\ g \,]\!] \;=\; \alpha \Rightarrow \{\alpha \longrightarrow N \mid n \longrightarrow N \in G\} \cup G \qquad \text{where} \quad \mathcal{N}[\![\, g \,]\!] = n \Rightarrow G$$

But there is some extra work before we return the result. In particular, productions in G might start with α (e.g., $n' \longrightarrow \alpha\,\overline{n}'$). While such form is allowed by the syntax of N, our ultimate goal is to turn

$$\dfrac{\dfrac{\mathcal{N}[\![\textsc{lpar}]\!] = \cdots \quad \dfrac{\cdots}{\mathcal{N}[\![\mu\ \text{sexps}\ .\ \epsilon \vee\ \text{sexp}\ \cdot\ \text{sexps}\ ]\!] = \text{sexps} \Rightarrow \{\ \text{sexps} \to \epsilon, \text{sexps} \to \text{sexp sexps}\ \}}}{\mathcal{N}[\![\textsc{lpar} \cdot (\mu\ \text{sexps}\ .\ \epsilon \vee\ \text{sexp}\ \cdot\ \text{sexps}\ )]\!] = n_1 \Rightarrow \{\ n_1 \to \textsc{lpar}\ \text{sexps}\ , \text{sexps} \to \epsilon, \text{sexps} \to \text{sexp sexps}\ \}} \quad \mathcal{N}[\![\textsc{rpar}]\!] = \cdots}{\mathcal{N}[\![\textsc{lpar} \cdot (\mu\ \text{sexps}\ .\ \epsilon \vee\ \text{sexp}\ \cdot\ \text{sexps}\ ) \cdot \textsc{rpar}]\!] = n_2 \Rightarrow \{\ n_2 \to \textsc{lpar}\ \text{sexps rpar}\ , \text{sexps} \to \epsilon, \text{sexps} \to \text{sexp sexps}, \text{rpar} \to \textsc{rpar}\ \}} \quad \mathcal{N}[\![\textsc{atom}]\!] = \cdots$$

$$\dfrac{\mathcal{N}[\![(\textsc{lpar} \cdot (\mu\ \text{sexps}\ .\ \epsilon \vee\ \text{sexp}\ \cdot\ \text{sexps}\ ) \cdot \textsc{rpar}) \vee \textsc{atom}]\!] = n_3 \Rightarrow \{\ n_3 \to \textsc{lpar}\ \text{sexps rpar}\ , n_3 \to \textsc{atom}\ , \text{sexps} \to \epsilon, \text{sexps} \to \text{sexp sexps}, \text{rpar} \to \textsc{rpar}\ \}}{\mathcal{N}[\![g]\!] = \text{sexp} \Rightarrow \{\ \text{sexp} \to \textsc{lpar}\ \text{sexps rpar}\ , \text{sexp} \to \textsc{atom}\ , \text{sexps} \to \epsilon, \text{sexps} \to \textsc{lpar}\ \text{sexps rpar sexps}\ , \text{sexps} \to \textsc{atom}\ \text{sexps}\ , \text{rpar} \to \textsc{rpar}\ \}}$$

Fig. 7. Normalizing s-expression $g = \mu\ \text{sexp}\ .(\textsc{lpar} \cdot (\mu\ \text{sexps}\ .\ \epsilon \vee\ \text{sexp}\ \cdot\ \text{sexps}\ ) \cdot \textsc{rpar}) \vee \textsc{atom}$

the productions into DGNF, where every nonterminal either starts with a terminal or is ϵ. Now that we learn the rules of α, we can look up and substitute in G all productions that start with α. For example, if $\alpha \to \textsc{b}$ and $n' \to \alpha\ \overline{n}$, then after substitution we have $n' \to \textsc{b}\ \overline{n}$. Note that we still allow α, as a special kind of nonterminal, to appear inside G if it is not the start of a production. That is, α in $n' \to t\ \alpha$ won't get substituted. It would actually be wrong to perform the substitution: if $\alpha \to \textsc{b}$, then after substitution $n' \to t\textsc{b}$ is not in DGNF.

Rule *(fix)* in Figure 6 presents the final form of normalizing a fixed point. The normalization first copies the productions for n into the rules for α (1), then substitutes in G all productions that start with α (2), and finally adds back all productions in G that do not start with α (3). As we will see, rule *(fix)* effectively guarantees that normalizing closed context-free expressions produces DGNF.

Lastly, in rule *(var)*, we create a fresh start symbol n with a singleton production $n \to \alpha$. Combining rule *(fix)* with rule *(var)*, normalization essentially treats α as a placeholder for the productions denoted by the fixed point. As soon as we know what α should be, we substitute α with its productions if necessary (as in rule *(fix)*). It may be tempting here to return $\alpha \Rightarrow \emptyset$ with α as a start symbol and no productions. That would be wrong, as $\alpha \Rightarrow \emptyset$ means an empty grammar, causing problems when, for example, rule *(alt)* copies productions.

Example. Figure 7 presents the simplified derivation of normalizing

$$g = \mu\ \text{sexp}\ .(\textsc{lpar} \cdot (\mu\ \text{sexps}\ .\ \epsilon \vee\ \text{sexp}\ \cdot\ \text{sexps}\ ) \cdot \textsc{rpar}) \vee \textsc{atom}$$

where we highlight grammar changes in light gray, and omit some details via $\cdots$ for space reasons and also since normalizing tokens is straightforward. We include the complete derivation tree in the appendix. Of particular interest is the last step, which normalizes a fixed point. In this case, sexp is used as the variable bound by the fixed point, and we have a production sexps $\to$ sexp sexps which is not DGNF. First, sexp copies all productions from n_4. Then, since the production sexps $\to$ sexp sexps starts with sexp, the production expands to two productions where sexp is replaced by its two normal forms respectively, making the resulting grammar in DGNF.

4.2 Semantics of DGNF

Our Definition 1 of DGNF presented in Section 3.1 gives a high-level description of a DGNF grammar. To prove that our normalization actually results in DGNF grammars, we first define the meaning of DGNF in terms of our formalization.

We start with the expansion relation:

Definition 2 (Expansion ($G \vdash \rightsquigarrow$)). *Given a grammar G, we define the expansion relation by (1) (Base) $G \vdash n \rightsquigarrow n$; (2) (Step) if $G \vdash n \rightsquigarrow \overline{t}\ n'\ \overline{n}$, and $(n' \to N' \in G)$, we have $G \vdash n \rightsquigarrow \overline{t}\ N'\ \overline{n}$. We write $G \vdash n \rightsquigarrow w$ when n expands to a complete word w.*

The expansion relation essentially captures what a nonterminal can expand to. For example, if $n \to \textsc{b}\ n_1 \in G$ and $n_1 \to \textsc{c} \in G$, then we have $G \vdash n \rightsquigarrow \textsc{bc}$. We enforce a left-to-right expansion

order for clarity and to stay close to the parsing behavior, but that is not necessary: it is easy to imagine an arbitrary order expansion, but any order leads to the same set of words.

With the notion of expansion, we define what it means for a grammar to be in DGNF precisely. The definition below follows the syntax given in Definition 1:

Definition 3 (Deterministic Greibach Normal Form (Semantics)). *A grammar G is in Deterministic Greibach normal form if all productions are of form $n \to t n_1 n_2 \dots n_k (k \geq 0)$, and moreover,*

- *(Determinism) for any pair of a nonterminal n and a terminal t, if there are two distinct productions $(n \to t_1\ \overline{n}_1) \in G$ and $(n \to t_2\ \overline{n}_2) \in G$, we have $t_1 \neq t_2$;*
- *(Guarded ϵ-productions) If $G \vdash n \rightsquigarrow \overline{t}\, n_1\, n_2\, \overline{n}$, if $(n_1 \to \epsilon) \in G$, then for any t either $(n_1 \to t\ \overline{n}_1) \notin G$ or $(n_2 \to t\ \overline{n}_2) \notin G$.*

The Determinism condition is straightforward, while the Guarded ϵ-productions condition needs more explanation. In Definition 1, we mentioned that the ϵ-production may only be used when no terminal symbol in other productions matches the input string. Consider that the next character to match is c. The case when both the ϵ-production $n_1 \to \epsilon$ and a production $n_1 \to \mathrm{c}$ can match raises when n_1's follow-up nonterminal n_2 can also match c, making it possible to use the ϵ-production while $n_1 \to \mathrm{c}$ also matches. Definition 3 captures such cases, requiring that n_1 and n_2 cannot match the same terminal if n_1 has an ϵ-production, and thus rules out example (4) in Section 3.1.

Now we can formally define the important property of DGNF that makes it practically useful.

Theorem 4.1 (Deterministic Parsing). *If G is a DGNF grammar, then for any expansion $G \vdash n \rightsquigarrow w$, there is a unique derivation for such expansion.*

4.3 Well-definedness and correctness

The normalization process serves as the basis for the parsing algorithm, and thus establishing its correctness is crucial for `flap`. In this section, we prove three key properties of normalization: first, normalization always succeeds for well-typed expressions (Section 4.3.1); then, normalization result will eventually get rid of the internal form $\alpha\ \overline{n}$ (Section 4.3.2); and finally, the result of normalization is a DGNF grammar (Section 4.3.3).

4.3.1 Normalization is well-defined. As we have briefly discussed in Section 4.1, correctness of normalization depends on types. For example, when normalizing sequencing $g_1 \cdot g_2$, rule *(seq)* returns $N_1\ n_2$ with $n_1 \to N_1$ from g_1, and n_2 from g_2. In order for $N_1\ n_2$ to be well-formed, we must ensure that N_1 is not ϵ, or otherwise $\epsilon\ n_2$ is ill-formed.

To this end, we make use of the typing information during normalization. In the case of sequencing, since $g_1 \cdot g_2$ is well-typed, the separation relation ($\tau_1 \circledast \tau_2$ in Figure 4) ensures g_1 is not nullable. We then prove that if an expression is not nullable, its normalization cannot have an ϵ production.

Lemma 4.2 (Productions of Null). *Given $\Gamma; \Delta \vdash g : \tau$ and $\mathcal{N}[\![g]\!]$ returns $n \Rightarrow G$, $\tau.\textsc{Null} = \mathsf{true}$ if and only if (1) $n \to \epsilon \in G$; or (2) $n \to \alpha \in G$ where $(\alpha : \tau') \in \Gamma$ and $\tau'.\textsc{Null} = \mathsf{true}$.*

In other words, if $\tau.\textsc{Null} = \mathsf{false}$, then $n \to \epsilon \notin G$.

With Lemma 4.2 and similar reasoning about typing during normalization, we prove that normalization is well-defined for well-typed expressions.

Theorem 4.3 (Well-definedness). *If $\Gamma; \Delta \vdash g : \tau$, then $\mathcal{N}[\![g]\!]$ returns $n \Rightarrow G$ for some G and n.*

4.3.2 Normalizing closed expressions produces no $\alpha\ \overline{n}$ form. Theorem 4.3 says that if an expression is well-typed, then normalization returns a grammar successfully. However, this grammar may not be in DGNF. For example, the grammar may include $n \to \alpha\ \overline{n}$, which is not a valid DGNF form.

Therefore, in order for the normalization result to be in DGNF, we need to prove that all $n \longrightarrow \alpha\ \overline{n}$ productions are removed from the normalization result.

In this part, we prove that the normalization result cannot contain any $\alpha\ \overline{n}$ production for a *closed* well-typed expression. To do so, we need to reason about the occurrences of α. The following lemma says that all α returned as the head of a production must be in the typing context.

Lemma 4.4 (Internal normal form). *Given* $\Gamma; \Delta \vdash g : \tau$ *and* $\mathcal{N}[\![\, g \,]\!]$ *returns* $n \Rightarrow G$,

- *if* $(n \longrightarrow \alpha\ \overline{n}) \in G$, *then we have* $\alpha \in \mathsf{dom}\,(\Gamma)$;
- *if* $(n' \longrightarrow \alpha\ \overline{n}) \in G$ *for any* n', *then we have* $\alpha \in \mathsf{fv}\,(g)$, *and thus* $\alpha \in \mathsf{dom}\,(\Gamma, \Delta)$.

Note that the first result applies only to the start symbol n, and its proof replies on the typing rule where α is well-typed only if $\alpha \in \Gamma$ (Figure 4). The second result applies to any n', and is proved making use of the first result. Specifically, the proof goes by induction on $\Gamma; \Delta \vdash g : \tau$, and the most interesting case is when normalizing $\mu\alpha.\ g$, where we need to prove that the productions of the start symbol of g cannot start with α, or otherwise normalizing $\mu\alpha.\ g$ would copy all productions from g for α which would result to, e.g., $\alpha \longrightarrow \alpha$ that fails the lemma. Fortunately, that is exactly what the first result tells us: when typing $\mu\alpha.\ g$, we add α in Δ (Figure 4), and thus normalizing g cannot have α at the head of a production for its start symbol.

Our goal then follows as a corollary of Lemma 4.4, which says that normalizing any closed well-typed expression only produces the desired normal form.

Corollary 4.5 (Normal form). *Given* $\bullet; \bullet \vdash g : \tau$ *and* $\mathcal{N}[\![\, g \,]\!]$ *returns* $\_ \Rightarrow G$, *then for all* $(n \longrightarrow N) \in G$, N *is* ϵ *or* $t\ \overline{n}$ *for some* t *and* $\overline{n}$.

4.3.3 Normalization returns DGNF grammars. Finally, we prove that normalization returns DGNF grammars. That requires productions to satisfy the conditions given in Definition 3.

We start with Determinism: all $n \longrightarrow t\ \overline{n}$ for the same n to start with different t. To prove that, we again make use of the typing information. The following lemma says that a production can start with t if and only if it belongs to the First set of the type.

Lemma 4.6 (Terminals in First). *Given* $\Gamma; \Delta \vdash g : \tau$ *and* $\mathcal{N}[\![\, g \,]\!]$ *returns* $n \Rightarrow G$, *we have* $t \in \tau.\mathit{First}$ *if and only if (1)* $(n \longrightarrow t\ \overline{n}) \in G$; *or (2)* $(n \longrightarrow \alpha\ \overline{n}) \in G$ *where* $(\alpha : \tau') \in \Gamma$ *and* $t \in \tau.\mathit{First}$.

This lemma is particularly important when proving the case for normalizing $g_1 \vee g_2$, where the typing condition $\tau_1 \# \tau_2$ ensures that g_1 and g_2 have disjoint First, which in turn ensures that rule (*alt*) only copies distinct head terminals from g_1 and g_2.

The proof for guarded ϵ-productions is more involved, which essentially requires us to show that during expansion $G \vdash n \rightsquigarrow \overline{t}\ n_1\ n_2\ \overline{n}$, the First of n_1 is disjoint with the First of n_2 (if n_1 is nullable). The proof replies on showing that "expansion preserves typing". Think about that from the well-typed context free expressions' point of view: if $(g_1 \vee g_2) \cdot g_3$ is well-typed, then $g_1 \cdot g_3$ (and $g_2 \cdot g_3$) must also be well-typed, and going from $(g_1 \vee g_2) \cdot g_3$ to $g_1 \cdot g_3$ is one step of branching, similar to one step of expansion. We refer the reader to the appendix for more details.

With all the conditions proved, we conclude our goal.

Theorem 4.7 ($\mathcal{N}[\![\, g \,]\!]$ produces DGNF). *If* $\bullet; \bullet \vdash g : \tau$ *and* $\mathcal{N}[\![\, g \,]\!]$ *returns* $\_ \Rightarrow G$, *then* G *is DGNF.*

4.4 Normalization Soundness

Normalization transforms a context-free expression into a DGNF grammar. As the final piece of metatheory for normalization, we would like to establish that normalization is sound with respect to the denotational semantics of well-typed context-free expressions defined in Krishnaswami and Yallop [2019]. The denotational semantics $[\![g]\!]_\gamma$ interprets g as a language (i.e., the set of all strings w matched by g), where γ gives interpretation of free variables in g.

$[\![g]\!]_\gamma$ interprets g as a language given the interpretation of the free variables in γ

$$
\begin{array}{lcllcl}
[\![\epsilon]\!]_\gamma & = & \{\epsilon\} & [\![g_1 \cdot g_2]\!]_\gamma & = & \{w \cdot w' \mid w \in [\![g_1]\!]_\gamma \wedge w' \in [\![g_2]\!]_\gamma\} \\
[\![t]\!]_\gamma & = & \{t\} & [\![\alpha]\!]_\gamma & = & \gamma(a) \\
[\![\bot]\!]_\gamma & = & \emptyset & [\![\mu\alpha.\, g]\!]_\gamma & = & \mathsf{fix}(\lambda \mathsf{L}.[\![g]\!]_{(\gamma, \mathsf{L}/\alpha)}) \\
[\![g_1 \vee g_2]\!]_\gamma & = & [\![g_1]\!]_\gamma \cup [\![g_2]\!]_\gamma & \multicolumn{3}{l}{\mathsf{fix}(f) = \bigcup_{i \in \mathbb{N}} \mathsf{L}_i \text{ where } \begin{array}{ll} \mathsf{L}_0 & = \emptyset \\ \mathsf{L}_{n+1} & = f(\mathsf{L}_n) \end{array}}
\end{array}
$$

Most definitions are straightforward. The semantics interpret ϵ as the singleton set containing the empty set, t as the singleton set that matches a one-character string t, $\bot$ as an empty language, and $g_1 \vee g_2$ as unions of sets. The interpretation of $g_1 \vee g_2$ appends a string from g_1 to a string from g_2. Variables α have their interpretation from the environment γ, and $\mu\alpha.\, g$ is interpreted as the least fixed point of g with respect to α.

To prove that our normalization is sound, we show that the normalized DGNF denotes exactly the same language as the denotation semantics of an expression. Recall that we have defined the expansion relation in Definition 2, where $G \vdash n \rightsquigarrow w$ denotes that n expands to a complete string w, where all non-terminals have been expanded. We prove the normalized grammar can expand to a string if and only if the string is included in the denotational semantics of the expression. The proof is done by first induction on the length of w and then on the structure of g.

Theorem 4.8 (Soundness). *Given $\bullet; \bullet \vdash g : \tau$ and $\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow G$, we have $w \in [\![g]\!]_\bullet$ if and only if $G \vdash n \rightsquigarrow w$ for any w.*

4.5 Implementation

The compositionality of the normalization algorithm simplifies the implementation of normalization in `flap`. Since the normalization of each term is defined in terms of the normal forms of its subterms, `flap` can represent terms in normal form. For example, if g and g' are `flap` parsers in normal form, then g `>>>` g' is also a parser in normal form, built from g and g' using the rules in Figure 6.

The simplicity of the normalization algorithm is reflected in the implementation of `flap`. Additionally, the most intricate part of the algorithm — dealing with fixed points — is also the subtlest part of the implementation. The implementation follows the formal algorithm closely, inserting placeholders (αs) that are tracked using an environment and resolved later. This kind of "backpatching" mirrors the way in which recursion is commonly implemented in eager functional languages such as OCaml [Reynaud et al. 2021]; if `flap` were instead implemented in a lazy language then it would be possible to implement fixed point normalization with significantly less fuss.

5 FUSION

This section shows how `flap`, starting with a separately-defined lexer and normalized parser, makes use of the syntactic restrictions of DGNF to implement lexer-parser fusion, eliminating tokens from generated code altogether.

5.1 Canonicalizing lexer

We assume a more restrictive definition of lexers than the interface in Section 2 provides. In particular, we assume that rules are disjoint on the left (i.e. there is no string that is matched by more than one regular expression in a set of rules), and disjoint on the right (i.e. there is exactly one **Skip** rule, and no token appears in more than one **Return** rule). Using negation and intersection, it is easy to transform a lexer that does not obey these constraints into an equivalent lexer that does,

$$\text{fused grammar} \quad F \quad ::= \quad \{\, n \to r\,\overline{n} \,\} \cup \{\, n \to ?r \,\}$$

$$\begin{aligned}
\mathcal{F}[\![\, L, G \,]\!] &= \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \\
\text{where } \mathcal{F}_1 &= \{\, n \to r\,\overline{n} \mid r \Rightarrow \textbf{Return}\ t \in L \wedge n \to t\,\overline{n} \in G \,\} && \textit{(inline the lexer)} \\
\mathcal{F}_2 &= \{\, n \to r\,n \mid r \Rightarrow \textbf{Skip} \in L \wedge n \in G \,\} && \textit{(whitespace)} \\
\mathcal{F}_3 &= \{\, n \to ?\neg r \mid n \to \epsilon \in G \wedge r = \textstyle\bigvee \{\, r \mid n \to r\,\overline{n} \in \mathcal{F}_1 \cup \mathcal{F}_2 \,\} \,\} && \textit{(epsilon productions)}
\end{aligned}$$

Fig. 8. Lexer-parser fusion

so there is no need to restrict the interface exposed to the user. With the lexer thus canonicalized and the parser translated into DGNF, it is straightforward to define fusion.

5.2 The fusion algorithm

Figure 8 formally defines the fusion algorithm.

The fusing function $\mathcal{F}[\![\, L, G \,]\!]$ operates on a canonicalized lexer L and a normalized grammar G, yielding a fused grammar F. The fused result consists of three parts.

First, we replace each production $n \to t\,\overline{n}$ with a new production $n \to r\,\overline{n}$, retrieving the regex r that is associated with the token t in the lexer L ($\mathcal{F}_1$). This is where the fusion function implicitly specializes the lexer to each nonterminal in the normalized grammar, and discards lexing rules that return tokens not in productions for the nonterminal. Canonicalizing the lexer to enforce disjointness simplifies this discarding of rules.

Then, we add an additional production $n \to r\,n$ for the **skip** regex r (which may be $\bot$) for each nonterminal, allowing each nonterminal to match an arbitrary number of the skip regex ($\mathcal{F}_2$).

Finally, for nonterminals with an epsilon production, the discarded regexes, along with the skip regex, are incorporated into a lookahead regex ($\mathcal{F}_3$). That is, we add a lookahead production $n \to ?\neg r$ for the regex that is the complement of the regexes that appear in other productions for n.

Fusion with normalized grammars is strikingly simple; it would be significantly more involved to directly fuse the context-free expressions with the lexing rules. Furthermore, as with normalized grammars, an expansion relation for fused grammars would carry the guarantee that every expansion has a unique derivation.

6 IMPLEMENTATION OF PARSING

This section describes the lexing and parsing algorithms, shows how to stage the parsing algorithm to improve performance, and explains details of the implementation of the algorithms in `flap`.

6.1 The lexing algorithm

Figure 9 presents the lexing algorithm. The algorithm has the *longest-match* semantics conventional for lexers: each token returned corresponds to the lexing rule that matches the longest possible prefix of the input string. This behaviour is implemented by repeatedly updating the best match seen *so far* until none of the lexing rules matches the input string.

$\mathcal{L}ex$ is the top-level lexing algorithm that takes the lexing rules L and an input string s, with two key utility functions $\mathcal{L}$ and $\mathcal{M}$. For simplicity, we assume utility functions can freely access the L argument to $\mathcal{L}ex$. At a high level, $\mathcal{L}$ reads a single token from a prefix of a string, pairs the token action with the remainder of the string, and passes it to $\mathcal{M}$. $\mathcal{M}$ constructs a sequence of tokens, updating the sequence according to the action passed from $\mathcal{L}$.

The $\mathcal{L}$ function takes four arguments: L' is a set of lexing rules; k is a token action representing the best match so far; rs is the remainder string for the best match; s is the input string. For empty input strings the best match information k and rs is passed to $\mathcal{M}$. For non-empty input strings

$$\mathcal{L}ex(L, s) = \mathcal{L}(L, \text{NO}, [], s)$$

$$\begin{array}{lcl}
\mathcal{L}(L', k, rs, []) & = & \mathcal{M}(k, rs) \\
\mathcal{L}(L', k, rs, c{::}cs) & = & \text{if } L'_c \stackrel{?}{=} \emptyset \text{ then } \mathcal{M}(k, rs) \\
 & & \text{else case } K \text{ of } \emptyset \mapsto \mathcal{L}(L'_c, k, rs, cs) \\
 & & \quad \{k'\} \mapsto \mathcal{L}(L'_c, k', cs, cs) \\
\multicolumn{3}{l}{\text{where } L'_c = \{\partial_c(r) \Rightarrow k \mid r \Rightarrow k \in L' \wedge \partial_c(r) \neq \bot\}} \\
\multicolumn{3}{l}{\quad\quad K = \{k \mid r \Rightarrow k \in L'_c \wedge \nu(r)\}}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{M}(\text{NO}, rs) & = & \text{FAIL} \\
\mathcal{M}(\text{Skip}, []) & = & [] \\
\mathcal{M}(\text{Skip}, c{::}cs) & = & \mathcal{L}(L, \text{NO}, [], c{::}cs) \\
\mathcal{M}(\text{Return } t, []) & = & [t] \\
\mathcal{M}(\text{Return } t, c{::}cs) & = & t :: \mathcal{L}(L, \text{NO}, [], c :: cs)
\end{array}$$

Fig. 9. Lexing algorithm

$$\mathcal{P}arse(n \Rightarrow G, s) = \mathcal{P}(n, s)$$

$$\begin{array}{lcl}
\mathcal{P}(n, []) & = & \text{if } n \to \epsilon \in G \text{ then } [] \text{ else FAIL} \\
\mathcal{P}(n, t{::}ts) & = & \text{if } n \to t\overline{n} \in G \text{ then } \mathcal{Q}(\overline{n}, ts) \\
 & & \text{else if } n \to \epsilon \in G \text{ then } t{::}ts \text{ else FAIL}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{Q}([], ts) & = & ts \\
\mathcal{Q}(n{::}ns, ts) & = & \mathcal{Q}(ns, \mathcal{P}(n, ts))
\end{array}$$

Fig. 10. Parsing algorithm for DGNF grammars

$$\mathcal{FP}arse(n \Rightarrow F, s) = \mathcal{G}([n], s)$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{F}(F_n, k, rs, s) =} \\
\multicolumn{3}{l}{\quad \text{case } s \text{ of } [] \mapsto Step(k, rs)} \\
 & & \quad\quad c{::}cs \mapsto \text{if } F'_n \stackrel{?}{=} \emptyset \text{ then } Step(k, rs) \\
 & & \quad\quad\quad \text{else case } K \text{ of } \emptyset \mapsto \mathcal{F}(F'_n, k, rs, cs) \\
 & & \quad\quad\quad\quad \{ns\} \mapsto \mathcal{F}(F'_n, \text{ON } ns, cs, cs) \\
\multicolumn{3}{l}{\text{where } F'_n = \{\langle \partial_c(r), k\rangle \mid \langle r, k\rangle \in F_n \wedge \partial_c(r) \neq \bot\}} \\
\multicolumn{3}{l}{\quad\quad K = \{k \mid \langle r, k\rangle \in F'_n \wedge \nu(r)\}}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{G}([], s) & = & s \\
\mathcal{G}(n{::}ns, s) & = & \mathcal{G}(ns, \mathcal{F}(F_n, k, s, s)) \\
\multicolumn{3}{l}{\text{where } F_n = \{\langle r, \overline{n}\rangle \mid n \to r\overline{n} \in F\}} \\
\multicolumn{3}{l}{\quad\quad k = \text{if } n \to ?r \in F \text{ then BACK else NO}}
\end{array}$$

$$\begin{array}{lcl}
Step(\text{BACK}, s) & = & s \\
Step(\text{ON } ns, s) & = & \mathcal{G}(ns, s) \\
Step(\text{NO}, s) & = & \text{FAIL}
\end{array}$$

Fig. 11. Parsing algorithm for fused grammars

$$\mathcal{SP}arse_{n \Rightarrow F}(s) = \mathcal{T}([n], s)$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{S}_{F_n,k}(rs, s) =} \\
\multicolumn{3}{l}{\quad \text{case } s \text{ of } [] \mapsto Step(k, rs)} \\
 & & \quad\quad c_i{::}cs \mapsto \text{if } F'_{n,i} \stackrel{?}{=} \emptyset \text{ then } Step(k, rs) \\
 & & \quad\quad\quad \text{else case } K_i \text{ of } \emptyset \mapsto \mathcal{S}_{F'_{n,i},k}(rs, cs) \\
 & & \quad\quad\quad\quad \{ns\} \mapsto \mathcal{S}_{F'_{n,i},\text{ON } ns}(cs, cs) \\
 & & \quad\quad c_j{::}cs \mapsto \ldots \\
\multicolumn{3}{l}{\text{where } F'_{n,i} = \{\langle \partial_{c_i}(r), k\rangle \mid \langle r, k\rangle \in F_n \wedge \partial_{c_i}(r) \neq \bot\}} \\
\multicolumn{3}{l}{\quad\quad K_i = \{k \mid \langle r, k\rangle \in F'_{n,i} \wedge \nu(r)\}}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{T}([], s) & = & s \\
\mathcal{T}(n{::}ns, s) & = & \mathcal{T}(ns, \mathcal{S}_{F_n,k}(s, s)) \\
\multicolumn{3}{l}{\text{where } F_n = \{\langle r, \overline{n}\rangle \mid n \to r\overline{n} \in F\}} \\
\multicolumn{3}{l}{\quad\quad k = \text{if } n \to ?r \in F \text{ then BACK else NO}}
\end{array}$$

$$\begin{array}{lcl}
Step(\text{BACK}, s) & = & s \\
Step(\text{ON } ns, s) & = & \mathcal{T}(ns, s) \\
Step(\text{NO}, s) & = & \text{FAIL}
\end{array}$$

Fig. 12. Staged parsing algorithm

c::cs, the result depends on L'_c, the set of lexing rules updated to use the non-empty *derivatives with respect to* c (Figure 2) of the string. If L' is empty, lexing cannot proceed any further, and so $\mathcal{L}$ transfers control to $\mathcal{M}$, passing the best match information. Otherwise, the result depends on the rule $r \Rightarrow a$ that matches the string up to this point including c (i.e. the rule that accepts ϵ after consuming c). If there is no such rule, then lexing continues with k. If there is such a rule, it is unique (since lexing rules are disjoint (Section 2.2)), and it represents a new longest-match k', and lexing continues with k' and the remainder for the best match cs.

The $\mathcal{M}$ function takes two arguments k, a token action and rs, a remainder string. There are five cases, one for the sentinel NO action, two for Skip actions, and two for Return actions. The sentinel NO indicates that lexing has failed. For Skip, lexing continues if the remainder rs is non-empty. For Return t, t is added to the output sequence, and lexing continues if the remainder rs is non-empty. In the cases where lexing continues, it commences by supplying NO for the best-match-so-far, so that reading the next token only succeeds if $\mathcal{L}$ matches a non-empty prefix of the remaining input.

6.2 The DGNF parsing algorithm

Figure 10 presents the parsing algorithm for grammars in Deterministic Greibach Normal Form. Deterministic parsing makes the algorithm simple, since there is no need for backtracking.

$\mathcal{P}arse$ is the top-level parsing algorithm which takes the parsing grammar $n \Rightarrow G$ and a sequence of tokens s. There are two key functions: $\mathcal{P}$ parses using a single nonterminal n, and $\mathcal{Q}$ parses using a sequence of nonterminals ns. Again, we assume $\mathcal{P}$ and $\mathcal{Q}$ can freely access G.

$\mathcal{P}$ takes the nonterminal n and a sequence of tokens and returns the remainder of the sequence after parsing. For empty sequences parsing succeeds only if the grammar has a rule $n \rightarrow \epsilon$. For non-empty sequences t::ts, if the grammar has a rule $n \rightarrow t\overline{n}$, $\mathcal{P}$ consumes t and parses ts with $\mathcal{Q}$. Otherwise, parsing succeeds (consuming nothing) only if the grammar has a rule $n \rightarrow \epsilon$.

$\mathcal{Q}$ takes a sequence of nonterminals ns and a sequence of tokens ts and parses successive prefixes of s with each nonterminal in ns.

6.3 The parsing algorithm for fused grammars

In practice, `flap` does not need separately-defined lexing and DGNF parsing algorithms, since it fuses lexing and parsing. We presented those algorithms to allow a direct comparison with the parsing algorithm for fused grammars.

Figure 11 shows an algorithm for parsing with fused grammars. The algorithm combines the features of the lexing algorithm (Figure 9) and the parsing algorithm (Figure 10): like the lexing algorithm it maintains a set of derivatives and an action and remainder string for the current *best match*; like the parsing algorithm, it keeps track of the current non-terminal.

$\mathcal{FP}arse$ takes the fused grammar $n \Rightarrow F$ and an input string s, with two key functions: $\mathcal{F}$ parses using a single nonterminal n, and $\mathcal{G}$ parses using a sequence of nonterminals ns using F.

$\mathcal{F}$ takes four arguments: F_n, a set of pairs representing non-epsilon productions for n; k, an action; rs, a remainder string; and s, an input string. For empty input strings the best match information k and rs is passed to $\mathcal{G}$ (via the auxiliary function $Step$). For non-empty input strings c::cs, the result depends on F'_n, the production pairs for n updated to use the non-empty *derivatives with respect to* c (Figure 2) of the string. If F'_n is empty, parsing cannot proceed any further, and so $\mathcal{F}$ transfers control to $\mathcal{G}$ (via $Step$), passing the best match information. Otherwise, the result depends on the production pair $\langle r, \overline{n} \rangle$ for which r matches the string up to this point including c (i.e. the rule that accepts ϵ after consuming c). If there is no such rule, then parsing continues with k. If there is such a rule, it is unique (since the regular expressions for a particular nonterminal are disjoint), and it represents a new longest-match $\overline{ns}$, and parsing continues, updating the best match information to ON $\overline{ns}$. Here ON $\overline{ns}$ represents one of three continuation types, and indicates that parsing should

continue using the nonterminal sequence $\overline{ns}$; the others are back, indicating that parsing with n should succeed, consuming no input, and no, indicating that parsing with n should fail. The $Step$ function matches these three cases, and takes an action appropriate to each continuation.

The $\mathcal{G}$ function takes a sequence of nonterminals ns and a sequence of characters s and parses successive prefixes of s with each nonterminal in ns by calling $\mathcal{F}$. The value of $\mathcal{F}$'s k argument depends on whether there is an epsilon rule for n in the fused grammar: if so, then a parsing failure with n should backtrack, consuming no input; if not, then parsing returns FAIL.

We draw attention to two salient features of the fused parsing algorithm: first, it consists of elements from the lexing and parsing algorithms of Sections 6.1 and 6.2; second, it does not materialize the tokens produced by the lexing algorithm, instead operating directly on the character string. The final algorithm in the next section makes this even more apparent.

6.4 The staged parsing algorithm

The parsing algorithm for fused grammars described in Section 6.3 is impractically inefficient. For each character of the input, the algorithm computes derivatives and checks emptiness and nullability for sets of regular expressions. However, since the regular expressions and other information about the grammar are known in advance of parsing, the inefficient algorithm can be *staged* [Taha 1999] to produce an efficient algorithm. The idea of staging is to identify those parts of the algorithm that do depend only on static information — i.e. on the grammar — and execute them first, leaving only the parts that depend on dynamic information — i.e. on the input string — for later. The result of staging, as illustrated below, is to transform the unstaged parser into a parser generator that produces as output a parser specialized to the input grammar.

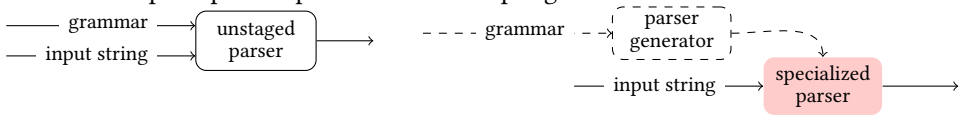


Figure 12 shows a staged version of the fused parsing algorithm. The structure of the algorithm is very close to the fused grammar parsing algorithm of Section 6.3: $\mathcal{S}$ corresponds to $\mathcal{F}$ and $\mathcal{T}$ corresponds to $\mathcal{G}$. However, there are three key differences.

First, those parts of the algorithm that depend on the input string are marked as *dynamic*, indicated with red highlighting. These dynamic elements are not executed immediately; instead they become part of the generated specialized parser produced by the first stage of execution.

Second, in the function $\mathcal{S}$, F_n and k have become indexes rather than arguments. Consequently, rather than being passed to the function at run-time, those arguments serve to distinguish generated functions: each instantiation of F_n and k generate a distinct function $\mathcal{S}$ in the specialized parser.

Finally, the case match in $\mathcal{S}$ is expanded to include a distinct case for each character c_i, c_j, etc. This expansion resolves a tension in the distinction between static and dynamic data: the static computation of derivatives $\partial_c(r)$ in the first stage depends on the value of c, which is only available dynamically. In the expanded case match the value of c_i is known on the right-hand side of the corresponding case, making it possible to compute derivatives valid within that program context. This scrutiny of a statically-unknown expression using a case match over its statically-known set of possible values is known as "The Trick" in the partial evaluation literature [Danvy et al. 1996].

The evaluation of the staged parsing algorithm is largely standard: the unhighlighted (static) expressions are executed first, producing the highlighted (dynamic) expressions as output. Each call to a dynamic indexed function $\mathcal{S}_{F_n,k}$ triggers the generation of a dynamic function whose body consists of the result of executing the right-hand side of $\mathcal{S}_{F_n,k}$ in Figure 12. To ensure that the generation process terminates, the generation of these indexed functions is memoized: there is at

most one generated function $\mathcal{S}_{F_n,k}$ for any particular F_n and k. The result of the algorithm is a set of mutually recursive functions that operate only on strings, not on components of the grammar:

$$
\begin{array}{lcll}
S_{n\to r\overline{n},\ldots,\text{BACK}}(r,s) & = & \text{case } s \text{ of} & [] \mapsto s \\
 & & & \texttt{'a'} :: cs \mapsto S_{n\to r_a\overline{n},\text{BACK}}(r,cs) \\
 & & & \texttt{'b'} :: cs \mapsto S_{n\to r_a\overline{n},\text{ON }\overline{ns}}(cs,cs) \\
 & & & \ldots \\
S_{n\to r\overline{n},\ldots,\text{ON }\overline{ns}}(r,s) & = & \ldots &
\end{array}
$$

6.5 Implementing the staged parsing algorithm

The `flap` library generates code for the fused grammar using MetaOCaml's staging facilities together with Yallop and Kiselyov's [2019] *letrec insertion* library for creating the indexed mutually-recursive functions produced by the staged parsing algorithm (Section 6.4).

There are three key differences between the pseudocode algorithm in Figure 12 and `flap`'s implementation. First, while the pseudocode presents a recognizer that either consumes input or fails, `flap` supports *semantic actions* — i.e. constructing and returning ASTs or other values when parsing succeeds — as described in Section 2.3.

Second, while the input to the pseudocode is a linked list of characters, `flap` operates on OCaml's more efficient flat array representation of strings, using indexes to keep track of string positions as parsing proceeds. Relatedly, `flap` also optimizes the test for the end of input by taking advantage of the fact that OCaml's strings are null-terminated, to ease interoperability with C. This representation allows the check for end of input to be incorporated into the branch on the next character in the generated code: a null character `'\000'` indicates a *possible* end of input, which can subsequently be confirmed by checking the string length.

Third, while the pseudocode generates a case in each branch for each possible character in the input, `flap` generates a much smaller number of cases by grouping characters with equivalent behaviour into classes. Branching on these character classes rather than treating characters individually leads to a substantial reduction in code size. Owens et al. [2009] describe the construction of these classes in described in detail.

Here is an excerpt of the code generated by `flap` for the s-expression parser:

```
and parse5 r i len s = match s.[i] with
  | ' '|'\n' → parse6 r (i + 1) len s
  | '('      → parse9 r (i + 1) len s
  | 'a'..'z' → parse3 r (i + 1) len s
  | '\000'   → if i = len then [] else failwith "unexpected"
  | _        → []
```

This excerpt shows the code generated for a single indexed function $\mathcal{S}_{F_n,k}$. There are four arguments, representing the beginning of the current token `r` (to support backtracking in the lookahead transition), the current index `i`, the input length `len`, and the input string `s`.

The subscripts 5, 6, etc. attached to the `parse` functions correspond to the indexes F_n, k in the pseudocode algorithm; the letrec insertion library assigns a fresh subscript to each distinct index.

The character range pattern `'a'..'z'` illustrates the character class optimization described above. Without that optimization, each of the characters from `'a'` to `'z'` would have a separate case in the **`match`** expression.

The check `i = len` determines whether the character `'\000'` represents the end of input or a null character in the input string.

The value `[]` corresponds to a semantic action: it is the empty list returned when an empty sequence of s-expressions is parsed. It appears twice in the generated code, since (as Figure 12

shows), parsing for a particular nonterminal can end in two ways: when it encounters the end of input, and when it encounters a non-matching character.

7 EVALUATION

This section evaluates the performance of `flap`, and shows that lexer-parser fusion drastically improves performance. Many parser combinator libraries suffer from poor performance, but the experiments described here show that combinator parsing does not need to be slow.

Three key techniques account for `flap`'s speed. First, Krishnaswami and Yallop's type system ensures that the time taken for parsing is linear in the length of the input, a substantial advantage over libraries that require backtracking. Second, staging eliminates the overhead arising from parsing abstractions, generating parsing code that is specialized to a particular grammar. Finally, lexer-parser fusion eliminates the overhead arising from defining lexers and parsers separately: in particular, it avoids the materialization of tokens, and eliminates all branching except for approximately one branch on each character in the input.

Krishnaswami and Yallop showed that the first two of these techniques can be used to build a parser combinator library that outperforms code generated by `ocamlyacc`. We focus here on the question of how much additional performance benefit arises from lexer-parser fusion.

7.1 Benchmarks

We build on the benchmark suite published by Krishnaswami and Yallop [2019], adding implementations of each benchmark for `flap`, for the parser generator `menhir` [Pottier and Régis-Gianas [n.d.]], and for the ParTS deterministic parsing library [Casinghino and Roux 2020], and extending the suite with an additional benchmark for parsing CSV files.

For each benchmark we compare up to six implementations. We make no comparison with unstaged parser combinators, which Krishnaswami and Yallop found to be significantly slower than their staged implementation, and between 4.5 and 125 times slower than ocamlyacc.

(a) an implementation generated by `ocamllex` and `ocamlyacc`
(b) an implementation generated by `ocamllex` and `menhir` in table-generation mode
(c) an implementation generated by `ocamllex` and `menhir` in code-generation (tableless) mode
(d) an implementation created using `flap`
(e) the staged parser implementation from Krishnaswami and Yallop [2019]
(f) an implementation using ParTS, where one is available

(a)–(c) use identically structured grammars and lexers in each benchmark, since `menhir` accepts `ocamlyacc` files as input. (d)–(f) also use identically structured grammars in each benchmark, since they all use the standard parser combinator interface (Section 2.3). However, (d)–(f) use differently-structured lexers: (e) and (f), taken respectively from Krishnaswami and Yallop [2019] and Casinghino and Roux [2020], reuse the deterministic parser combinators for lexing, while `flap` uses the more conventional lexing interface described in Section 2.2.

The benchmarks are as follows:

(1) (pgn) Chess game descriptions in Portable Game Notation format. The semantic actions extract the result of each game. The input is a corpus of 6759 Grand Master games.
(2) (ppm) Image files in Netpbm format. The semantic actions validate the non-syntactic constraints of the format, such as colour range and pixel count.
(3) (sexp) S-expressions with alphanumeric atoms. The semantic actions of the parser count the number of atoms in each s-expression.
(4) (csv) A parser for the Comma-Separated Value format. The grammar conforms quite closely to RFC 4180 [Shafranovich 2005], but makes the terminating CRLF mandatory and does not treat

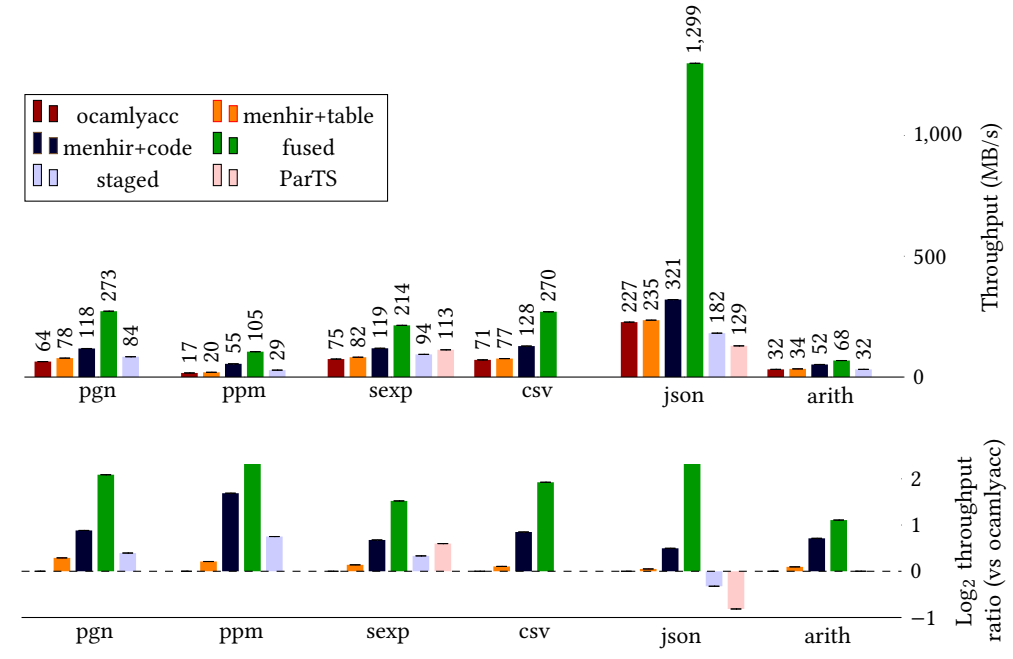


Fig. 13. Parser throughput: ocamlyacc, menhir, `flap`, staged combinators and ParTS

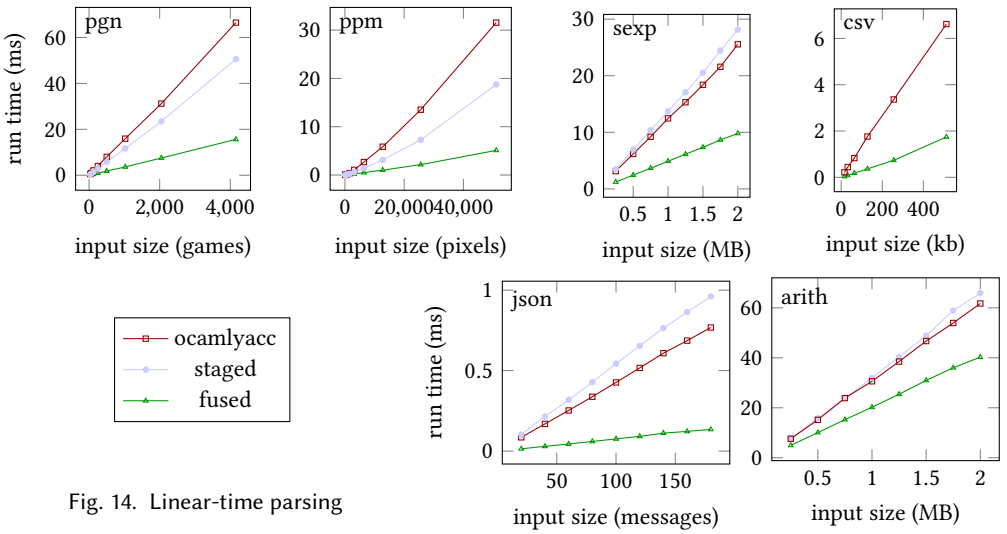


Fig. 14. Linear-time parsing

headers specially. The semantic actions check that each row contains the same number of fields. There is no implementation using Krishnaswami and Yallop's combinators for this benchmark, because more than a single character of lookahead is needed to distinguish escaped (i.e. repeated) double-quotes "" from unescaped quotes ", and so the lexer cannot be implemented with typed context-free expressions without substantial changes to its structure. The lexer interface used in `flap` (Section 2.2) does not suffer from this limitation.

	Input		Normalized		Fused	Output
Grammar	Lexer rules	Context-free exps	Nonterms	Productions	Productions	Functions
pgn	13	95	38	53	91	206
ppm	6	10	5	6	16	55
sexp	4	11	3	6	9	11
csv	3	14	5	7	7	20
json	12	42	9	33	42	97
arith	14	143	28	55	83	209

Table 1. Sizes of inputs, intermediate representations, and generated code

Benchmark	Compilation time (ms)
sexp	0.331
pgn	212
ppm	3.60
json	28.5
csv	0.499
arith	460

Table 2. Compilation time (type-checking, normalization, fusion, code generation)

(5) (json) A parser for JavaScript Object Notation (JSON). The semantic actions count the number of objects represented in the input. Following Krishnaswami and Yallop, we use the simple JSON grammar given by Jonnalagedda et al. [2014].
(6) (arith) A miniature programming language with constructs for arithmetic, comparison, let-binding and branching. The semantic actions evaluate the parsed expression.

7.2 Running time

Figure 13 shows the absolute and relative throughput of the six implementations using the six benchmark grammars. For the benchmarks that are taken from [Krishnaswami and Yallop 2019] we use the test corpora from the same source. For the CSV benchmark we have generated a set of files of various sizes and dimensions, using a random variety of textual and numeric data.

The benchmarks were compiled with BER MetaOCaml N111 with flambda optimizations enabled and run on a single Intel i9-12900K core with 8GB memory running Ubuntu Linux, using the `Core_bench` micro-benchmarking library [Hardin and James 2013].

As the graph shows, our experiments confirm the results reported by Krishnaswami and Yallop: the staged implementation of typed context-free expressions generally outperforms `ocamlyacc`. The addition of lexer-parser fusion makes `flap` considerably faster than both typed CFEs and `ocamlyacc`, reaching around 1.3GB/s (a little under 2.5 cycles per byte) on the `json` benchmark.

Linear-time parsing. Finally, as Figure 14 illustrates, on Krishnaswami and Yallop's benchmark suite, parsers built with `flap`, like parsers built with Krishnaswami and Yallop's combinators, execute in time linear in the length of their input.

7.3 Code size

A second important measure of usefulness for parsing: if parsing tools are to be usable in practice, it is essential that they do not generate unreasonably large code.

There are several reasons to be apprehensive about the size of code generated by `flap`. First, conversion to Greibach Normal Form is well known to substantially increase the size of grammars; for example, in the procedure given by Blum and Koch [1999] the result of converting a grammar

G has size $O(|G|^3)$. Second, the fusion process is inherently duplicative, repeatedly copying the lexer rules into the various grammar productions. Finally, experience in the multi-stage programming community shows that it is easy to inadvertently generate extremely large programs, since antiquotation makes it easy to duplicate terms.

However, measurements largely dispel these concerns. Table 1 gives the size of the representations of the benchmark parsers at various stages in `flap`'s pipeline. The leftmost pair of columns of figures shows the size of the input parsers, measured as the number of lexer rules (including both **Return** and **Skip** rules) and the number of context-free expression nodes, as described in Section 2. The pair of columns to the right shows the number of nonterminals and productions after the grammar is converted to Deterministic GNF using the procedure in Section 4. As the figures show, the normalization algorithm for typed context-free expressions does not produce the drastic increases in size that occurs in the more general form of conversion to GNF. The next column to the right shows the size of the grammar after fusion (Section 5). Fusion does not alter the number of nonterminals, but it can add productions: for example, the **Skip** rules in the s-expression lexer add additional productions to each nonterminal. Finally, the rightmost column shows the number of function bindings in the code generated by `flap`. Comparing this generated function count with the number of context-free expressions in the input reveals a fairly unalarming relationship: with one exception (pgn), the ratio between the two barely exceeds 2.

Sharing. The entries for *pgn* and *arith* hint at opportunities for further improvement. In both cases, the number of context-free expressions that make up the grammar (95 and 143) is surprisingly high, since both languages are fairly simple. Inspecting the implementations of the grammars reveals the cause: in several places, the combinators that construct the grammar duplicate subexpressions. For example, here is the implementation of a Kleene plus operator used in *pgn*:

```
let oneormore e = (e >>> star e) ...
```

Normalization turns these two occurrences of `e` into multiple entries in the normalized form, and ultimately to multiple functions in the generated code.

The core problem is that the parser combinator interface (Section 2.3) provides no way to express sharing of subgrammars. Since duplication of this sort is common, it is likely that extending `flap` with facilities to express and maintain sharing could substantially reduce generated code size.

7.4 Compilation time

A final measure of practicality is the time taken to perform the fusion transformation. Slow compilation times can have a significant effect on usability; as Nielsen [1993] notes, software that takes more than one second to respond can cause a user to lose concentration, harming interactivity.

Table 2 shows the compilation time for the six benchmark grammars. In each case, the total time taken to type-check and normalize the grammar, fuse the grammar and lexer and generate optimized code is less than half a second.

8 RELATED WORK

Deterministic Greibach Normal Form. There are several longstanding results related to deterministic variants of Greibach Normal Form. For example, Geller et al. [1976] show that every strict deterministic language can be given a strict deterministic grammar in Greibach Normal Form, and Nijholt [1979] gives a translation into Greibach Normal Form that preserves strict deterministicness. The distinctive contributions of this paper are the new normal form that is well suited to fusion, and the compositional normalization procedure from typed context-free expressions, allowing deterministic GNF to be used in the implementation of parser combinators.

Combining lexers and parsers. The work most closely related to ours, by Casinghino and Roux [2020] investigates the application of traditional stream fusion techniques to parser combinators in the ParTS system. We have included their two published benchmarks in the evaluation of Section 7 and found that, as they report, when the flambda suite of compiler optimizations is applied to their code, its performance is similar to the results achieved by Krishnaswami and Yallop [2019]. A significant difference between their work and ours is that they approach fusion as a traditional optimization problem, in which transformations are applied to code that satisfies certain heuristics, and are not applied in more complex cases. In contrast, we treat lexer-parser fusion as a sequence of total transformations that is guaranteed to convert every input (i.e. every parser) into a form that enjoys pleasant performance properties. More concretely, Figure 13 shows significant performance differences between ParTS and `flap`, with ParTS achieving one half and a tenth of the throughputs of `flap` on the `sexp` and `json` benchmarks.

Another line of work, on *Scannerless GLR parsing* [Economopoulos et al. 2009; van den Brand et al. 2002], also aims to eliminate the boundary between lexers and parsers, both in the interface and the implementation. The principal aim is to provide a principled way to handle lexical ambiguity, in contrast to our focus on performance.

Context-aware scanning, introduced by Van Wyk and Schwerdfeger [2007] is another variant on the parser-scanner interface focused on disambiguation; it passes contextual information from the parser to the scanner about the set of valid tokens at a particular point, in a similar way to the lexer specialization in Section 3.3 of this paper. However, Van Wyk and Schwerdfeger's framework goes further, and allows the automatic selection of a lexer (not just a subset of lexing rules) based on the parsing context.

Fusion. The notion of fusion, in the sense of merging computations to eliminate intermediate structures, has been applied in several domains, including query engines [Shaikhha et al. 2018], GPU kernels [Filipovic et al. 2015] and tree traversals [Sakka et al. 2019].

Perhaps the most widespread is stream fusion, which appears to have originated with Wadler's deforestation [Wadler 1990], and has since been successfully applied as both a traditional compiler optimization [Coutts et al. 2007] and as a staged library [Kiselyov et al. 2017] that provides guarantees similar to those we give here for parsers.

Parser optimization. Finally, in contrast to the constant-time speedups resulting from lexer-parser fusion, we note an intriguing piece of work by Klyuchnikov [2010] that applies two-level-supercompilation to parser optimization, leading to asymptotic improvements.

9 FUTURE WORK

There are a number of promising avenues for future work. First, extending `flap`'s rather minimal lexer and parser interfaces to support common needs such as left-recursive grammars, lexers and parsers with multiple entry points, mechanisms for maintaining state during parsing, and more expressive lexer semantic action could make the library substantially more usable in practice.

Second, building on the proofs of normalization correctness in Section 4 to cover the whole of `flap`, we plan to formally establish that the code generated by Section 5 faithfully represents the semantics of the combinators in Section 2.

Third, applying the ideas in this paper to more powerful parsing algorithms such as LR(1) and LALR(1), and incorporating them into traditional standalone parser generator (rather than a staged library) could make lexer-parser fusion available to many more software developers.

Finally, it may be that fusion can be extended to longer pipelines than the lexer-parser interface that we investigate here. Might it be possible to fuse together (e.g.) decompression, unicode decoding, lexing and parsing into a single computation that does not materialize intermediate values?

REFERENCES

Norbert Blum and Robert Koch. 1999. Greibach Normal Form Transformation Revisited. *Information and Computation* 150, 1 (1999), 112–118. https://doi.org/10.1006/inco.1998.2772

Anders Bondorf. 1992. Improving Binding Times Without Explicit CPS-conversion. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, USA) *(LFP '92)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/141471.141483

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

Chris Casinghino and Cody Roux. 2020. ParTS: Final Report. HR001120C0016 - Final Report.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 315–326. https://doi.org/10.1145/1291151.1291199

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-Expansion Does The Trick. *ACM Trans. Program. Lang. Syst.* 18, 6 (1996), 730–751. https://doi.org/10.1145/236114.236119

Rowan Davies and Frank Pfenning. 1996. A Modal Analysis of Staged Computation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 258–270. https://doi.org/10.1145/237721.237788

Giorgios Economopoulos, Paul Klint, and Jurgen J. Vinju. 2009. Faster Scannerless GLR Parsing. In *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5501)*, Oege de Moor and Michael I. Schwartzbach (Eds.). Springer, 126–141. https://doi.org/10.1007/978-3-642-00722-4_10

Jiri Filipovic, Matus Madzin, Jan Fousek, and Ludek Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *J. Supercomput.* 71, 10 (2015), 3934–3957. https://doi.org/10.1007/s11227-015-1483-z

Matthew M. Geller, Michael A. Harrison, and Ivan M. Havel. 1976. Normal forms of deterministic grammars. *Discret. Math.* 16, 4 (1976), 313–321. https://doi.org/10.1016/S0012-365X(76)80004-0

Sheila A. Greibach. 1965. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM* 12, 1 (Jan. 1965), 42–52. https://doi.org/10.1145/321250.321254

Christopher S. Hardin and Roshan P. James. 2013. Core_bench: Micro-Benchmarking for OCaml. OCaml Workshop.

Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 637–653. https://doi.org/10.1145/2660193.2660241

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *FLOPS 2014 (LNCS, Vol. 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 285–299. https://doi.org/10.1145/3009837

Ilya Klyuchnikov. 2010. Towards effective two-level supercompilation. Preprint 81. Keldysh Institute of Applied Mathematics, Moscow.

Neelakantan R. Krishnaswami and Jeremy Yallop. 2019. A typed, algebraic approach to parsing, See [McKinley and Fisher 2019], 379–393. https://doi.org/10.1145/3314221.3314625

Kathryn S. McKinley and Kathleen Fisher (Eds.). 2019. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM. https://doi.org/10.1145/3314221

Jakob Nielsen. 1993. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Kristian Nielsen and Morten Heine Sørensen. 1995. Call-By-Name CPS-Translation As a Binding-Time Improvement. In *Proceedings of the Second International Symposium on Static Analysis (SAS '95)*. Springer-Verlag, London, UK, UK, 296–313. http://dl.acm.org/citation.cfm?id=647163.717677

Anton Nijholt. 1979. Strict Deterministic Grammars and Greibach Normal Form. *J. Inf. Process. Cybern.* 15, 8/9 (1979), 395–401.

Scott Owens, John H. Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *J. Funct. Program.* 19, 2 (2009), 173–190. https://doi.org/10.1017/S0956796808007090

François Pottier and Yann Régis-Gianas. [n.d.]. *The Menhir parser generator*. http://gallium.inria.fr/~fpottier/menhir/.

Alban Reynaud, Gabriel Scherer, and Jeremy Yallop. 2021. A practical mode system for recursive definitions. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434326

Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. 2019. Sound, fine-grained traversal fusion for heterogeneous trees, See [McKinley and Fisher 2019], 830–844. https://doi.org/10.1145/3314221.3314626

Yakov Shafranovich. 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180. https://doi.org/10.17487/RFC4180

Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2018. Push versus pull-based loop fusion in query engines. *J. Funct. Program.* 28 (2018), e10. https://doi.org/10.1017/S0956796818000102

Walid Taha. 1999. *Multi-Stage Programming: Its Theory and Applications.* Technical Report.

Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. 2002. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 143–158. https://doi.org/10.1007/3-540-45937-5_12

Eric R. Van Wyk and August C. Schwerdfeger. 2007. Context-aware Scanning for Parsing Extensible Languages. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering* (Salzburg, Austria) *(GPCE '07)*. ACM, New York, NY, USA, 63–72. https://doi.org/10.1145/1289971.1289983

Philip Wadler. 1985. How to Replace Failure by a List of Successes. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture* (Nancy, France). Springer-Verlag, Berlin, Heidelberg, 113–128.

Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

Jeremy Yallop and Oleg Kiselyov. 2019. Generating Mutually Recursive Definitions. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Cascais, Portugal) *(PEPM 2019)*. ACM, New York, NY, USA, 75–81. https://doi.org/10.1145/3294032.3294078

A COMPLETE DERIVATION

This section presents the complete derivation for normalizing

$$g = \mu\, \text{sexp} .(\textsc{lpar} \cdot (\mu\, \text{sexps} . \epsilon \vee \text{sexp} \cdot \text{sexps}) \cdot \textsc{rpar}) \vee \textsc{atom}$$

We automatically remove unreachable productions in the result.

$$\frac{\mathcal{N}[\![\text{sexp}]\!] = n_2 \Rightarrow \{ n_2 \to \text{sexp} \} \qquad \mathcal{N}[\![\text{sexps}]\!] = n_3 \Rightarrow \{ n_3 \to \text{sexps} \}}{\mathcal{N}[\![\text{sexp} \cdot \text{sexps}]\!] = n_4 \Rightarrow \{ n_4 \to \text{sexp}\, n_3, n_3 \to \text{sexps} \}}$$

$$\frac{\mathcal{N}[\![\epsilon]\!] = n_1 \Rightarrow \{ n_1 \to \epsilon \} \qquad \mathcal{N}[\![\text{sexp} \cdot \text{sexps}]\!] = n_4 \Rightarrow \{ n_4 \to \text{sexp}\, n_3, n_3 \to \text{sexps} \}}{\mathcal{N}[\![\epsilon \vee \text{sexp} \cdot \text{sexps}]\!] = n_5 \Rightarrow \{ n_5 \to \epsilon, n_5 \to \text{sexp}\, n_3, n_3 \to \text{sexps} \}}$$

$$\frac{\mathcal{N}[\![\epsilon \vee \text{sexp} \cdot \text{sexps}]\!] = n_5 \Rightarrow \{ n_5 \to \epsilon, n_5 \to \text{sexp}\, n_3, n_3 \to \text{sexps} \}}{\mathcal{N}[\![\mu\, \text{sexps} . \epsilon \vee \text{sexp} \cdot \text{sexps}]\!] = \text{sexps} \Rightarrow \{ \text{sexps} \to \epsilon, \text{sexps} \to \text{sexp}\, n_3, n_3 \to \epsilon, n_3 \to \text{sexp}\, n_3 \}}$$

$$\frac{\mathcal{N}[\![\textsc{lpar}]\!] = n_6 \Rightarrow \{ n_6 \to \textsc{lpar} \} \qquad \vdots}{\mathcal{N}[\![\textsc{lpar} \cdot (\mu\, \text{sexps} . \epsilon \vee \text{sexp} \cdot \text{sexps})]\!] = n_7 \Rightarrow \{ n_7 \to \textsc{lpar}\, \text{sexps}, \text{sexps} \to \epsilon, \text{sexps} \to \text{sexp}\, n_3, n_3 \to \epsilon, n_3 \to \text{sexp}\, n_3 \}}$$

$$\frac{\vdots \qquad \mathcal{N}[\![\textsc{rpar}]\!] = \text{rpar} \Rightarrow \{ \text{rpar} \to \textsc{rpar} \}}{\mathcal{N}[\![\textsc{lpar} \cdot (\mu\, \text{sexps} . \epsilon \vee \text{sexp} \cdot \text{sexps}) \cdot \textsc{rpar}]\!] = n_8 \Rightarrow \{ n_8 \to \textsc{lpar}\, \text{sexps}\, \text{rpar}, \text{sexps} \to \epsilon, \text{sexps} \to \text{sexp}\, n_3, n_3 \to \epsilon, n_3 \to \text{sexp}\, n_3, \text{rpar} \to \textsc{rpar} \}}$$

$$\mathcal{N}[\![\textsc{atom}]\!] = n_9 \Rightarrow \{ n_9 \to \textsc{atom} \}$$

$$\mathcal{N}[\![(\textsc{lpar} \cdot (\mu\, \text{sexps} . \epsilon \vee \text{sexp} \cdot \text{sexps}) \cdot \textsc{rpar}) \vee \textsc{atom}]\!] = n_{10} \Rightarrow \{ n_{10} \to \textsc{lpar}\, \text{sexps}\, \text{rpar}, n_{10} \to \textsc{atom}, \text{sexps} \to \epsilon, \text{sexps} \to \text{sexp}\, n_3, n_3 \to \epsilon, n_3 \to \text{sexp}\, n_3, \text{rpar} \to \textsc{rpar} \}$$

$$\mathcal{N}[\![g]\!] = \text{sexp} \Rightarrow \{ \text{sexp} \to \textsc{lpar}\, \text{sexps}\, \text{rpar}, \text{sexp} \to \textsc{atom}, \text{sexps} \to \epsilon, \text{sexps} \to \textsc{lpar}\, \text{sexps}\, \text{rpar}\, n_3, \text{sexps} \to \textsc{atom}\, n_3, n_3 \to \epsilon, n_3 \to \textsc{lpar}\, \text{sexps}\, \text{rpar}\, n_3, n_3 \to \textsc{atom}\, n_3, \text{rpar} \to \textsc{rpar} \}$$

Comparing the simplified derivation in Section 4.1 with the complete derivation, we note the following simplification: first, we omit the derivation of tokens; second, when normalizing sexps we produce a nonterminal n_3 with a production $n_3 \to$ sexps. That means n_3 is equivalent to sexps. However, this n_3 is retained in the final result, making the final grammar a big larger. It's easy to check that the grammar is equivalent to the one given in the paper.

It is easy to consider an optimization process that gets rid of n_3 in the middle of the derivation. For example, for the result for n_5, instead of

$$n_5 \Rightarrow n_5 \to \epsilon, n_5 \to \text{sexp}\, n_3, n_3 \to \text{sexps}$$

We can have

$$n_5 \Rightarrow n_5 \to \epsilon, n_5 \to \text{sexp sexps}$$

Then the normalization result would be exactly the same as the one in the paper.

B DETERMINISTIC PARSING

Theorem 4.1 (Deterministic Parsing). *If G is a DGNF grammar, then for any expansion $G \vdash n \rightsquigarrow w$, there is a unique derivation for such expansion.*

Proof. By straightforward induction on $G \vdash n \rightsquigarrow w$. □

C WELL-TYPED NORMALIZATION

This section presents well-typed normalization, which shows how normalization captures the type information, and then proves its properties that are important for later proofs.

First, we note that during normalization (Figure 6), we create one fresh nonterminal exactly for one context-free expression. Therefore, we can attach to each nonterminal its type information. That is, instead of n, we have n_τ, where τ indicates the type of n. We also write α_τ where τ is the type of α as in $\mu\alpha : \tau.\ g$.

Refining the normalization, we have:

$\mathcal{N}[\![\, g \,]\!]$ returns $n_\tau \Rightarrow G$, with a grammar G and the start nonterminal n of type τ (with n fresh)

(*epsilon*)	$\mathcal{N}[\![\, \epsilon \,]\!]$	=	$n_{\tau_\epsilon} \Rightarrow \{\, n_{\tau_\epsilon} \to \epsilon \,\}$
(*token*)	$\mathcal{N}[\![\, t \,]\!]$	=	$n_{\tau_c} \Rightarrow \{\, n_{\tau_c} \to t \,\}$
(*bot*)	$\mathcal{N}[\![\, \bot \,]\!]$	=	$n_{\tau_\bot} \Rightarrow \emptyset$
(*seq*)	$\mathcal{N}[\![\, g_1 \cdot g_2 \,]\!]$	=	$n_{\tau_1 \cdot \tau_2} \Rightarrow \{n_{\tau_1 \cdot \tau_2} \to N_1\ n_{2\tau_2} \mid n_{1\tau_1} \to N_1 \in G_1\} \cup G_1 \cup G_2$ where $\mathcal{N}[\![\, g_1 \,]\!] = n_{1\tau_1} \Rightarrow G_1 \wedge \mathcal{N}[\![\, g_2 \,]\!] = n_{2\tau_2} \Rightarrow G_2$
(*alt*)	$\mathcal{N}[\![\, g_1 \vee g_2 \,]\!]$	=	$n_{\tau_1 \vee \tau_2} \Rightarrow \{n_{\tau_1 \vee \tau_2} \to N_1 \mid n_{1\tau_1} \to N_1 \in G_1\} \cup \{n_{\tau_1 \vee \tau_2} \to N_2 \mid n_{2\tau_2} \to N_2 \in G_2\} \cup G_1 \cup G_2$ where $\mathcal{N}[\![\, g_1 \,]\!] = n_{1\tau_1} \Rightarrow G_1 \wedge \mathcal{N}[\![\, g_2 \,]\!] = n_{2\tau_2} \Rightarrow G_2$
(*fix*)	$\mathcal{N}[\![\, \mu\alpha : \tau.\ g \,]\!]$	=	$\alpha_\tau \Rightarrow \{\alpha_\tau \to N \mid n_\tau \to N \in G\} \cup \{n'_{\tau'} \to N\ \overline{n}' \mid n'_{\tau'} \to \alpha_\tau\ \overline{n}' \in G \wedge n_\tau \to N \in G\} \cup G\backslash_{n'_{\tau'} \to \alpha_\tau\ \overline{n}'}$ where $\mathcal{N}[\![\, g \,]\!] = n_\tau \Rightarrow G$ $G\backslash_{n'_{\tau'} \to \alpha_\tau\ \overline{n}'}$ is G with all $n'_{\tau'} \to \alpha_\tau\ \overline{n}'$ removed for any n', τ' and $\overline{n}'$
(*var*)	$\mathcal{N}[\![\, \alpha_\tau \,]\!]$	=	$n_\tau \Rightarrow \{\, n_\tau \to \alpha_\tau \,\}$

We also add to typing that

$$\frac{}{\Gamma; \Delta \vdash n_\tau : \tau}$$

With that, we can type-check any N according to the typing rules, by treating t as constants, n_τ as nonterminal of type τ, and lists as sequences (e.g., $n_{1\tau_1}\ n_{2\tau_2}$ as $n_{1\tau_1} \cdot n_{2\tau_2}$).

Now we can prove properties about the well-typed normalization. While those lemmas are proved in the typed normalization, they naturally hold for the untyped normalization as the two are the same process.

Lemma C.1. *Given $\Gamma; \Delta \vdash g : \tau$, and $\mathcal{N}[\![\, g \,]\!]$ returns $n_{\tau'} \Rightarrow G$, then $\tau = \tau'$.*

Proof. By a straightforward induction on $\Gamma; \Delta \vdash g : \tau$. □

Lemma C.2. *Given $\Gamma; \Delta \vdash g : \tau'$, and $\mathcal{N}[\![\, g \,]\!]$ returns $\_ \Rightarrow G$, then for any $n_\tau \in G$, if $N_1, ..., N_i$ are all productions of n. we have*

- *$\tau = \tau_1 \vee \tau_2 \vee \cdots \vee \tau_i$, where*
- *$(n_\tau \to N_1 \in G \wedge \Gamma; \Delta \vdash N_1 : \tau_1)$ and $(n_\tau \to N_2 \in G \wedge \Gamma; \Delta \vdash N_2 : \tau_2)$ and $\cdots$ and $(n_\tau \to N_i \in G \wedge \Gamma; \Delta \vdash N_i : \tau_i)$; and*

- $\tau_1 \# \tau_2 \cdots \# \tau_i$, *i.e., all* $\tau_1, \tau_2, \cdots, \tau_i$ *are apart from each other.*

Proof. By induction on $\Gamma; \Delta \vdash g : \tau$.

- The cases for ϵ, t, $\bot$ and α follow trivially.
- The case for $g_1 \cdot g_2$.
  $\mathcal{N}[\![g_1 \cdot g_2]\!] = n_{\tau_1 \cdot \tau_2} \Rightarrow \{n_{\tau_1 \cdot \tau_2} \to N_1\ n_{2\tau_2} \mid n_{1\tau_1} \to N_1 \in G_1\} \cup G_1 \cup G_2$
  $\mathcal{N}[\![g_1]\!] = n_{1\tau_1} \Rightarrow G_1 \wedge \mathcal{N}[\![g_2]\!] = n_{2\tau_2} \Rightarrow G_2$
  By I.H., we know that for each N_1, we have $\Gamma; \Delta \vdash N_1 : \tau_1'$ for some τ_1', and τ_1 is the $\vee$ of all τ_1', and all τ_1' is apart (#) from each other.
  According to well-typedness, we know that $\tau_1 \circledast \tau_2$, which says that $\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset$, and $\neg\tau_1.\textsc{Null}$.
  Since τ_1 is the $\vee$ of all τ_1', we know $\tau_1'.\textsc{Null} = \text{false}$, and $\tau_1'.\textsc{FLast} \cap \tau_2.\textsc{First} = \emptyset$, and thus $\tau_1' \circledast \tau_2$.
  So $\Gamma; \Delta \vdash N_1\ n_{2\tau_2} : \tau_1' \cdot \tau_2$
  Moreover, since $\tau_1'.\textsc{Null} = \text{false}$, we have $\tau_1' \cdot \tau_2.\textsc{First} = \tau_1'.\textsc{First}$ and $\tau_1' \cdot \tau_2.\textsc{Null} = \text{false}$.
  Given that all τ_1' apart from each other, we can derive that all $\tau_1' \cdot \tau_2$ apart from each other.
- The case for
  $\mathcal{N}[\![g_1 \vee g_2]\!] = n_{\tau_1 \vee \tau_2} \Rightarrow \{n_{\tau_1 \vee \tau_2} \to N_1 \mid n_{1\tau_1} \to N_1 \in G_1\} \cup \{n_{\tau_1 \vee \tau_2} \to N_2 \mid n_{2\tau_2} \to N_2 \in G_2\}$
  $\cup\ G_1 \cup G_2$
  $\mathcal{N}[\![g_1]\!] = n_{1\tau_1} \Rightarrow G_1 \wedge \mathcal{N}[\![g_2]\!] = n_{2\tau_2} \Rightarrow G_2$
  By I.H., we know that for each N_1, we have $\Gamma; \Delta \vdash N_1 : \tau_1'$ for some τ_1', and τ_1 is the $\vee$ of all τ_1', and all τ_1' is apart (#) from each other. Moreover, for each N_2, we have $\Gamma; \Delta \vdash N_2 : \tau_2'$ for some τ_2', and τ_2 is the $\vee$ of all τ_2', and all τ_2' is apart (#) from each other.
  It's easy to see that $\tau_1 \vee \tau_2$ is the $\vee$ of all τ_1' and τ_2'.
  According to well-typedness, we know that $\tau_1 \# \tau_2$. That is $\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset$, and $\neg(\tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null})$. From the former, we can derive that $\tau_1'.\textsc{First} \cap \tau_2'.\textsc{First} = \emptyset$. From the latter, we know that at least one of τ_1 and τ_2 has $\textsc{Null} = \text{false}$, so at least one of τ_1' and τ_2' has $\textsc{Null} = \text{false}$. With that, we have $\tau_1' \# \tau_2'$. Thus, all τ_1' and τ_2' apart from each other.
- The case for $\mu\alpha : \tau.\ g$
  $\mathcal{N}[\![\mu\alpha : \tau.\ g]\!] = \alpha_\tau \Rightarrow \{\alpha_\tau \to N \mid n_\tau \to N \in G\} \cup \{n'_{\tau'} \to N\ \overline{n}' \mid n'_{\tau'} \to \alpha_\tau\ \overline{n}' \in G \wedge n_\tau \to N \in G\}$
  $\cup\ G \backslash_{n'_{\tau'} \to \alpha_\tau\ \overline{n}'}$
  $\mathcal{N}[\![g]\!] = n_\tau \Rightarrow G$
  $G \backslash_{n'_{\tau'} \to \alpha_\tau\ \overline{n}'}$ is G with all $n'_{\tau'} \to \alpha_\tau\ \overline{n}'$ removed for any n', τ' and $\overline{n}'$
  By I.H., we know that for each N, we have $\Gamma; \Delta \vdash N : \tau''$ for some τ'', and τ is the $\vee$ of all τ'', and all τ'' is apart (#) from each other.
  The goal for α_τ follows from n_τ. The remaining is to show that the goal holds for each $n'_{\tau'}$ that has a production that starts with α_τ. Essentially what happens is that one production $n'_{\tau'} \to \alpha_\tau\ \overline{n}'$ is replaced by multiple productions $n'_{\tau'} \to N\ \overline{n}'$ for each $n_\tau \to N \in G$ where $\Gamma; \Delta \vdash N : \tau''$.
  First, we need to show that $N\ \overline{n}'$ is well-typed. We already know each individual terminal or nonterminal in $N\ \overline{n}'$ is well-typed, so the only requirement is the $\circledast$ condition during type-checking. Given that $\alpha_\tau\ \overline{n}'$ is well-typed, we know that $\tau.\textsc{Null} = \text{false}$, so $\tau''.\textsc{Null} = \text{false}$. Moreover, $\tau''.\textsc{FLast} \subseteq \tau.\textsc{FLast}$. With that, and the fact that $\alpha_\tau\ \overline{n}'$ is well-typed, we can derive that the $\circledast$ condition is always satisfied when type-checking $N\ \overline{n}'$. Therefore, $N\ \overline{n}'$ is well-typed.
  Because τ is the $\vee$ of all τ'', it's easy to show that the type of $\alpha_\tau\ \overline{n}'$ is the $\vee$ of the types of all $N\ \overline{n}'$. Therefore, the type of n' is the same as before. Also, all types of the productions of n' are still apart with each other.

□

D NORMALIZATION IS WELL-DEFINED (PROOF)

Lemma 4.2 (Productions of Null). *Given* $\Gamma;\Delta \vdash g : \tau$ *and* $\mathcal{N}[\![g]\!]$ *returns* $n \Rightarrow G$, $\tau.\textsc{Null} = \text{true}$ *if and only if (1)* $n \to \epsilon \in G$*; or (2)* $n \to \alpha \in G$ *where* $(\alpha : \tau') \in \Gamma$ *and* $\tau'.\textsc{Null} = \text{true}$.

Proof. **Left to right** According to Lemma C.2, we must have one $n_\tau \to N \in G$, where $\Gamma;\Delta \vdash N : \tau$, and $\tau.\textsc{Null} = \text{true}$. We case analyze the shape of N:

- If $N = \epsilon$, then we have proved (1).
- If $N = t\ \overline{n}$, then it's impossible that $\tau.\textsc{Null} = \text{true}$.
- If $N = \alpha\ \overline{n}$. Since $\alpha\ \overline{n}$ is well-typed, if $\overline{n}$ is not empty, then the type must have $\textsc{Null} = \text{false}$. Therefore $\overline{n}$ must be empty, and α has its type $\textsc{Null} = \text{true}$. So we have proved (2).

Right to left Following Lemma C.2, the type τ is the $\vee$ of all types. If either $n \to \epsilon$ of α has type $\textsc{Null} = \text{true}$, we know that $\tau.\textsc{Null} = \text{true}$.

□

Theorem 4.3 (Well-definedness). *If* $\Gamma;\Delta \vdash g : \tau$*, then* $\mathcal{N}[\![g]\!]$ *returns* $n \Rightarrow G$ *for some* G *and* n.

Proof. By induction on g. Most cases are straightforward. The only interesting cases are when $g = g_1 \cdot g_2$ or $g = \mu\alpha.\ g'$.

- $g = g_1 \cdot g_2$. We have:
  $\mathcal{N}[\![g_1 \cdot g_2]\!] = n \Rightarrow \{n \to N_1\ n_2 \mid n_1 \to N_1 \in G_1\} \cup G_1 \cup G_2$
  $\mathcal{N}[\![g_1]\!] = n_1 \Rightarrow G_1 \wedge \mathcal{N}[\![g_2]\!] = n_2 \Rightarrow G_2$
  As $g_1 \cdot g_2$ is well-typed, we know that the type of g_1 has $\textsc{Null} = \text{false}$. By Lemma 4.2, N_1 is not ϵ, ensuring that $N_1\ n_2$ is a valid form.
- $g = \mu\alpha.\ g'$. We have:
  $\Gamma;\Delta, \alpha : \tau \vdash g : \tau$
  $\mathcal{N}[\![\mu\alpha.\ g]\!] = \alpha \Rightarrow \{\alpha \to N \mid n \to N \in G\} \cup \{n' \to N\ \overline{n}' \mid n' \to \alpha\ \overline{n}' \in G \wedge n \to N \in G\} \cup G\backslash_{n' \to \alpha\ \overline{n}'}$
  $\mathcal{N}[\![g]\!] = n \Rightarrow G$
  $G\backslash_{n' \to \alpha\ \overline{n}'}$ is G with all $n' \to \alpha\ \overline{n}'$ removed for any n' and $\overline{n}'$
  We need to show that $N\ \overline{n}'$ is valid, requiring either N to not be ϵ, or $\overline{n}'$ to be empty. Since $\alpha\ \overline{n}'$ is well-typed (Lemma C.2), we know that either $\overline{n}'$ is empty, or α must have $\textsc{Null} = \text{false}$. In the first case we are done. In the second case, following Lemma 4.2, we know N cannot be ϵ.

□

E NORMALIZATION RETURNS DGNF GRAMMARS (PROOF)

E.1 Normalizing closed expressions produces no $\alpha\ \overline{n}$ form

Lemma 4.4 (Internal normal form). *Given* $\Gamma;\Delta \vdash g : \tau$ *and* $\mathcal{N}[\![g]\!]$ *returns* $n \Rightarrow G$,

- *if* $(n \to \alpha\ \overline{n}) \in G$*, then we have* $\alpha \in \text{dom}(\Gamma)$*;*
- *if* $(n' \to \alpha\ \overline{n}) \in G$ *for any* n'*, then we have* $\alpha \in \text{fv}(g)$*, and thus* $\alpha \in \text{dom}(\Gamma, \Delta)$.

Proof. **Part 1** By induction on $\Gamma;\Delta \vdash g : \tau$, most cases are straightforward. We discuss the following three cases:

- $g = \alpha$. As g is well-typed, it must be $\alpha \in \text{dom}(\Gamma)$. The goal follows directly.
- $g = g_1 \cdot g_2$. The goal follows by the I.H. on g_1.
- $g = \mu\alpha.\ g'$. As the well-typedness of g' adds α to Δ, the goal follows directly by the I.H. on g'.

Part 2 By induction on $\Gamma;\Delta \vdash g : \tau$. The only interesting case is when $g = \mu\alpha.\ g'$. We have

$\Gamma; \Delta, \alpha : \tau \vdash g : \tau$
$\mathcal{N}[\![\, \mu\alpha.\ g \,]\!] = \alpha \Rightarrow \{\alpha \to N \mid n \to N \in G\}_{(1)} \cup \{n' \to N\ \overline{n}' \mid n' \to \alpha\ \overline{n}' \in G_{(2)} \wedge n \to N \in G\} \cup G\backslash_{n' \to \alpha\ \overline{n}'}{}_{(3)}$
$\mathcal{N}[\![\, g \,]\!] = n \Rightarrow G$
$G\backslash_{n' \to \alpha\ \overline{n}'}$ is G with all $n' \to \alpha\ \overline{n}'$ removed for any n' and $\overline{n}'$
By I.H., we know that for all $(n'' \to beta\ \overline{n}) \in G\backslash_{n' \to \alpha\ \overline{n}'}$, $\beta \in \mathrm{fv}(g)$.
For N, if it is $beta\ \overline{n}$, then either $\beta \in \mathrm{fv}(\mu\alpha.\ g)$, or $\beta = \alpha$. By Part 1, we know that $\beta \in \mathrm{dom}(\Gamma)$, so $\beta \neq \alpha$. So it can only be $\beta \in \mathrm{fv}(\mu\alpha.\ g)$. And the goal follows.

□

Corollary 4.5 (Normal form). *Given $\bullet;\bullet \vdash g : \tau$ and $\mathcal{N}[\![\, g \,]\!]$ returns $\_ \Rightarrow G$, then for all $(n \to N) \in G$, N is ϵ or $t\ \overline{n}$ for some t and $\overline{n}$.*

Proof. Follows directly from Lemma 4.4. □

E.2 A nonterminal's non-ϵ productions start with distinct terminals.

Lemma 4.6 (Terminals in First). *Given $\Gamma; \Delta \vdash g : \tau$ and $\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow G$, we have $t \in \tau.\textsc{First}$ if and only if (1) $(n \to t\ \overline{n}) \in G$; or (2) $(n \to \alpha\ \overline{n}) \in G$ where $(\alpha : \tau') \in \Gamma$ and $t \in \tau.\textsc{First}$.*

Proof. **Left to right** According to Lemma C.2, we must have one $n_\tau \to N \in G$, where $\Gamma; \Delta \vdash N : \tau$, and $t \in \tau.\textsc{First}$. We case analyze the shape of N:

- If $N = \epsilon$, then it's impossible.
- If $N = t\ \overline{n}$, then we have proved (1).
- If $N = \alpha\ \overline{n}$. Since $\alpha\ \overline{n}$ is well-typed, the First of the type of $\alpha\ \overline{n}$ is equivalent to the First of the type of α. So we have proved (2).

Right to left Following Lemma C.2, the type τ is the $\vee$ of all types. If either $n \to t\ \overline{n}$ of α has type $t \in \textsc{First}$, we know that $t \in \tau.\textsc{First}$.

□

Lemma E.1 (Productions with distinct terminals). *If $\Gamma; \Delta \vdash g : \tau$, and $\mathcal{N}[\![\, g \,]\!]$ returns $\_ \Rightarrow G$, then for any two productions $(n \to t_1\ \overline{n}_1) \in G$ and $(n \to t_2\ \overline{n}_2) \in G$, we have $t_1 \neq t_2$.*

Proof. Suppose there are $n \to t\ \overline{n}_1$ and $n \to t\ \overline{n}_2$.
By Lemma C.2, we know that the types of $t\ \overline{n}_1$ and $t\ \overline{n}_2$ must be apart. Therefore they have disjoint First.
By Lemma 4.6, we know that both $t\ \overline{n}_1$ and $t\ \overline{n}_2$ have $t \in \textsc{First}$. However, since their types have disjoint First, this is impossible. So contradiction.

□

E.3 The ϵ-production may only be used when other productions do not apply.

We defined the notion of containment of types as follows. The key of the definition is rule ST-BASE, which says that a grammar g_1 is a subtype grammar of g_2, if g_1 is of type τ_1, g_2 is of type τ_2, and $\tau_1 = \tau_2 \vee \tau$ for some τ. Notably, we have $\Gamma; \Delta \vdash g \lesssim g$ for any well-typed grammar $\Gamma; \Delta \vdash g : \tau$, as we have $\tau = \tau \vee \{\textsc{Null} = \mathrm{false};\ \textsc{First} = \emptyset;\ \textsc{FLast} = \emptyset\}$.

$\boxed{\Gamma;\Delta \vdash g_1 \lesssim g_2}$ *(containment of types)*

ST-BASE
$$\frac{\Gamma;\Delta \vdash g_1 : \tau_1 \qquad \Gamma;\Delta \vdash g_2 : \tau_2 \qquad \tau_1 = \tau_2 \vee \tau}{\Gamma;\Delta \vdash g_1 \lesssim g_2}$$

ST-TRANS
$$\frac{\Gamma;\Delta \vdash g_1 \lesssim g_2 \qquad \Gamma;\Delta \vdash g_2 \lesssim g_3}{\Gamma;\Delta \vdash g_1 \lesssim g_3}$$

ST-CON
$$\frac{\Gamma;\Delta \vdash g_1 \lesssim g_1' \qquad \Gamma;\Delta \vdash g_2 \lesssim g_2'}{\Gamma;\Delta \vdash g_1 \cdot g_2 \lesssim g_1' \cdot g_2'}$$

ST-UNION
$$\frac{\Gamma;\Delta \vdash g_1 \lesssim g_1' \qquad \Gamma;\Delta \vdash g_2 \lesssim g_2'}{\Gamma;\Delta \vdash g_1 \vee g_2 \lesssim g_1' \vee g_2'}$$

Lemma E.2. *If* $\Gamma;\Delta \vdash g_1 : \tau_1$, *and* $\Gamma;\Delta \vdash g_1 \lesssim g_2$, *then* $\Gamma;\Delta \vdash g_2 : \tau_2$, *and* $\tau_1 = \tau_2 \vee \tau$ *for some* τ.

Proof. By induction on $\Gamma;\Delta \vdash g_1 \lesssim g_2$.

- rule ST-BASE follows trivially.
- Case
ST-TRANS
$$\frac{\Gamma;\Delta \vdash g_1 \lesssim g_2 \qquad \Gamma;\Delta \vdash g_2 \lesssim g_3}{\Gamma;\Delta \vdash g_1 \lesssim g_3}$$
We have g_1 of type τ_1.
By I.H., g_2 of type τ_2, and $\tau_1 = \tau_2 \vee \tau$.
By the second I.H., g_3 of type τ_3, and $\tau_2 = \tau_3 \vee \tau'$.
Therefore, $\tau_1 = \tau_3 \vee (\tau \vee \tau')$.
- Case
ST-CON
$$\frac{\Gamma;\Delta \vdash g_1 \lesssim g_1' \qquad \Gamma;\Delta \vdash g_2 \lesssim g_2'}{\Gamma;\Delta \vdash g_1 \cdot g_2 \lesssim g_1' \cdot g_2'}$$
We have $g_1 \cdot g_2$ of type $\tau_1 \cdot \tau_2$ with g_1 of type τ_1 and g_2 of type τ_2 and $\tau_1 \circledast \tau_2$.
By I.H., g_1' of type τ_1', and $\tau_1 = \tau_1' \vee \tau$.
By the second I.H., g_2' of type τ_2', and $\tau_2 = \tau_2' \vee \tau'$.
Now we want to show $g_1' \cdot g_2'$ is of type $\tau_1' \cdot \tau_2'$. For that, we need to prove $\tau_1' \circledast \tau_2'$.
That means we need to prove $\tau_1'.\textsc{First} \cap \tau_2'.\textsc{First} = \emptyset \wedge \neg\tau_1'.\textsc{Null}$
We already know $\tau_1 \circledast \tau_2$, which means $\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset \wedge \neg\tau_1.\textsc{Null}$
Since $\tau_1 = \tau_1' \vee \tau$ and $\tau_2 = \tau_2' \vee \tau'$, we can derive $\tau_1'.\textsc{First} \cap \tau_2'.\textsc{First} = \emptyset \wedge \neg\tau_1'.\textsc{Null}$
Therefore, $\tau_1' \circledast \tau_2'$, and $g_1' \cdot g_2'$ is of type $\tau_1' \cdot \tau_2'$.
Now the goal is to relate $\tau_1 \cdot \tau_2$ with $\tau_1' \cdot \tau_2'$.
$$\tau_1 \cdot \tau_2 = \begin{cases} \textsc{Null} & = & \tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null} \\ \textsc{First} & = & \tau_1.\textsc{First} \ \cup \tau_1.\textsc{Null} \,?\, \tau_2.\textsc{First} \\ \textsc{FLast} & = & \tau_2.\textsc{FLast} \ \cup \tau_2.\textsc{Null} \,?\, (\tau_2.\textsc{First} \cup \tau_1.\textsc{FLast}) \end{cases}$$
given $\neg\tau_1.\textsc{Null}$
$$\tau_1 \cdot \tau_2 = \begin{cases} \textsc{Null} & = & \text{false} \\ \textsc{First} & = & \tau_1.\textsc{First} \\ \textsc{FLast} & = & \tau_2.\textsc{FLast} \ \cup \tau_2.\textsc{Null} \,?\, (\tau_2.\textsc{First} \cup \tau_1.\textsc{FLast}) \end{cases}$$
Similarly,
$$\tau_1' \cdot \tau_2' = \begin{cases} \textsc{Null} & = & \text{false} \\ \textsc{First} & = & \tau_1'.\textsc{First} \\ \textsc{FLast} & = & \tau_2'.\textsc{FLast} \ \cup \tau_2'.\textsc{Null} \,?\, (\tau_2'.\textsc{First} \cup \tau_1'.\textsc{FLast}) \end{cases}$$
We have $\tau_1 = \tau_1' \vee \tau$ and $\tau_2 = \tau_2' \vee \tau'$. Therefore, with $\neg\tau_2.\textsc{Null}$ implying $\neg\tau_2'.\textsc{Null}$,
$$\tau_1 \cdot \tau_2 = (\tau_1' \cdot \tau_2') \vee \begin{cases} \textsc{Null} & = & \text{false} \\ \textsc{First} & = & \tau.\textsc{First} \\ \textsc{FLast} & = & \tau'.\textsc{FLast} \ \cup \tau_2.\textsc{Null} \,?\, (\tau_2.\textsc{First} \cup \tau_1.\textsc{FLast}) \end{cases}$$

- Case

$$\textsc{st-union}\ \frac{\Gamma;\Delta \vdash g_1 \lesssim g_1' \qquad \Gamma;\Delta \vdash g_2 \lesssim g_2'}{\Gamma;\Delta \vdash g_1 \vee g_2 \lesssim g_1' \vee g_2'}$$

We have $g_1 \vee g_2$ of type $\tau_1 \vee \tau_2$ with g_1 of type τ_1 and g_2 of type τ_2 and $\tau_1 \# \tau_2$.
By I.H., g_1' of type τ_1', and $\tau_1 = \tau_1' \vee \tau$.
By the second I.H., g_2' of type τ_2', and $\tau_2 = \tau_2' \vee \tau'$.
Now we want to show $g_1' \vee g_2'$ is of type $\tau_1' \vee \tau_2'$. For that, we need to prove $\tau_1' \# \tau_2'$.
That means we need to prove $(\tau_1'.\textsc{First} \cap \tau_2'.\textsc{First} = \emptyset) \wedge \neg(\tau_1'.\textsc{Null} \wedge \tau_2'.\textsc{Null})$
We already know $\tau_1 \# \tau_2$, which means $(\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset) \wedge \neg(\tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null})$
Since $\tau_1 = \tau_1' \vee \tau$ and $\tau_2 = \tau_2' \vee \tau'$, we can derive $(\tau_1'.\textsc{First} \cap \tau_2'.\textsc{First} = \emptyset) \wedge \neg(\tau_1'.\textsc{Null} \wedge \tau_2'.\textsc{Null})$
Therefore, $\tau_1' \# \tau_2'$, and $g_1' \vee g_2'$ is of type $\tau_1' \vee \tau_2'$.
Finally, we have $\tau_1 \vee \tau_2 = (\tau_1' \vee \tau_2') \vee (\tau \vee \tau')$.

□

Lemma E.3 (Expansion preserves typing). *Given* $\Gamma;\Delta \vdash g : \tau$, $\mathcal{N}[\![g]\!]$ *returns* $\_ \Rightarrow G$, *if* $G \vdash n_\tau \rightsquigarrow \overline{t}\, n' \, \overline{n}$, *then* $\Gamma;\Delta \vdash \overline{t}\, n' \, \overline{n} : \tau_1$, *and* $\tau = \tau_1 \vee \tau'$ *for some* τ'.

Proof. By induction on $G \vdash n_\tau \rightsquigarrow \overline{t}\, n' \, \overline{n}$.

- In the base case, $G \vdash n_\tau \rightsquigarrow n_\tau$. The goal follows trivially.
- In the inductive case, we have $G \vdash n_\tau \rightsquigarrow \overline{t}\, n' \, \overline{n}$, $n' \longrightarrow N \in G$ and so $G \vdash n \rightsquigarrow \overline{t}\, N \, \overline{n}$,
  By I.H., we have $\Gamma;\Delta \vdash \overline{t}\, n' \, \overline{n} : \tau_1$, and $\tau = \tau_1 \vee \tau'$.
  According to Lemma C.2, we know that $\Gamma;\Delta \vdash n' \lesssim N$ by rule st-base.
  Therefore, $\Gamma;\Delta \vdash \overline{t}\, n' \, \overline{n} \lesssim \overline{t}\, N \, \overline{n}$ by rule st-con.
  By Lemma E.2, $\Gamma;\Delta \vdash \overline{t}\, N \, \overline{n} : \tau_2$, and $\tau_1 = \tau_2 \vee \tau''$.
  Therefore, $\tau = \tau_2 \vee (\tau' \vee \tau'')$.

□

Lemma E.4 (Guarded ϵ-production). *Given* $\Gamma;\Delta \vdash g : \tau$, $\mathcal{N}[\![g]\!]$ *returns* $n \Rightarrow G$, *and* $G \vdash n \rightsquigarrow^* \cdots n_1 n_2 \cdots$, *if* $(n_1 \longrightarrow \epsilon) \in G$, *then either* $(n_1 \longrightarrow t\ \overline{n}_1) \notin G$ *or* $(n_2 \longrightarrow t\ \overline{n}_2) \notin G$ *for any* $t, \overline{n}_1, \overline{n}_2$.

Proof. We have:

$\cdots n_1 n_2 \cdots$ is well-typed	By Lemma E.3
The type of $\cdots n_1$ is $\tau \cdot \tau_1$, the type of n_1 is τ_1, and the type of n_2 is τ_2	Suppose
$\tau \cdot \tau_1 \circledast \tau_2$	By typing
$\tau \cdot \tau_1.\textsc{FLast} \cap \tau_2.\textsc{First} = \emptyset$	By $\circledast$
$\tau \cdot \tau_1.\textsc{FLast} = \tau_1.\textsc{FLast} \cup \tau_1.\textsc{Null}\ ?\ (\tau_1.\textsc{First} \cup \tau.\textsc{FLast})$	By definition
$n_1 \longrightarrow \epsilon \in G$	Given
$n_1.\textsc{Null} = \text{true}$	Lemma 4.2
$\tau \cdot \tau_1.\textsc{FLast} = \tau_1.\textsc{FLast} \cup (\tau_1.\textsc{First} \cup \tau.\textsc{FLast})$	Follows
$\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset$	Follows
$n_1 \longrightarrow t\ \overline{n}_1 \in G \wedge n_2 \longrightarrow t\ \overline{n}_2 \in G$	Assume
$t \in \tau_1.\textsc{First} \wedge t \in \tau_2.\textsc{First}$	Lemma 4.6

Contradiction with $\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset$ | □

E.4 Final result

Theorem 4.7 ($\mathcal{N}[\![\, g \,]\!]$ produces DGNF). *If* $\bullet; \bullet \vdash g : \tau$ *and* $\mathcal{N}[\![\, g \,]\!]$ *returns* $\_ \Rightarrow G$, *then* G *is DGNF.*

Proof. Follows from Corollary 4.5, Lemma E.1, and Lemma E.4. □

F SOUNDNESS (PROOF)

F.1 An alternative normalization

To make proofs easier, we consider the definition $\boldsymbol{\mathcal{N}}$, which has the same definition as $\mathcal{N}$ except for the case of $\mu\alpha.\, g$, where we do not substitute α:

$$\boldsymbol{\mathcal{N}}(\mu\alpha : \tau.\, g) \quad = \quad \alpha \Rightarrow \{\alpha \to N \mid n \to N \in G\} \cup G \quad \text{where} \quad \boldsymbol{\mathcal{N}}(g) = n \Rightarrow G$$

While $\boldsymbol{\mathcal{N}}$ does not return a DGNF grammar, it is easy to see that $\mathcal{N}$ and $\boldsymbol{\mathcal{N}}$ defines the same language:

Lemma F.1. *If* $\mathcal{N}[\![\, g \,]\!]$ *return* $n_1 \Rightarrow G_1$, *and* $\boldsymbol{\mathcal{N}}(g)$ *return* $n_2 \Rightarrow G_2$, *then for all* w, $G_1 \vdash n_1 \rightsquigarrow^* w$ *if and only if* $G_2 \vdash n_2 \rightsquigarrow^* w$.

Proof. By straightforward induction on g. □

F.2 Subexpression

The subexpression relation essentially defines a subset relation between the grammars denoted by context-free expressions.

$\boxed{g_1 \sqsubseteq g_2}$ *(Subexpression)*

$$\textsc{sg-refl}\ \frac{}{g \sqsubseteq g} \qquad \textsc{sg-trans}\ \frac{g_1 \sqsubseteq g_2 \qquad g_2 \sqsubseteq g_3}{g_1 \sqsubseteq g_3} \qquad \textsc{sg-con-l}\ \frac{}{g_1 \sqsubseteq g_1 \cdot g_2} \qquad \textsc{sg-con-r}\ \frac{}{g_2 \sqsubseteq g_1 \cdot g_2} \qquad \textsc{sg-union-l}\ \frac{}{g_1 \sqsubseteq g_1 \vee g_2}$$

$$\textsc{sg-union-r}\ \frac{}{g_2 \sqsubseteq g_1 \vee g_2} \qquad \textsc{sg-mu}\ \frac{}{g \sqsubseteq \mu\alpha.\, g}$$

We can show that what subexpression means in terms of the alternative normalization.

Lemma F.2. *If* $g_1 \sqsubseteq g_2$, *and* $\boldsymbol{\mathcal{N}}(g_1)$ *returns* $n_1 \Rightarrow G_1$, *and* $\boldsymbol{\mathcal{N}}(g_2)$ *returns* $n_2 \Rightarrow G_2$, *then for all* $n \in \mathsf{dom}(G_1)$, $(n \to N) \in G_1$ *if and only if* $(n \to N) \in G_2$.

Proof. By straightforward induction on $g_1 \sqsubseteq g_2$. □

F.3 Proof of soundness

In the following lemma statement, we denote a natural number as $\mathbb{n}$, and the length of a word w as $|w|$. The relations $\gamma \vDash \Gamma$ and $\delta \vDash \Delta$ mean that γ and δ give interpretations (i.e., languages L) of variables in Γ and Δ respectively.

$$\frac{}{\bullet \vDash \bullet} \qquad \frac{\delta \vDash \Delta \quad \mathsf{L} \vDash \tau}{\delta, \mathsf{L}/\alpha \vDash \Delta, \alpha : \tau}$$

$$\mathsf{L} \vDash \tau \triangleq \textsc{Null}(LL) \Rightarrow \tau.\textsc{Null} \wedge \textsc{First}(\mathsf{L}) \subseteq \tau.\textsc{First} \wedge \textsc{FLast}(\mathsf{L}) \subseteq \tau.\textsc{FLast}$$

Lemma F.3. *Given* $\Gamma; \Delta \vdash g : \tau$, *and* $\gamma \vDash \Gamma$, *and* $\delta \vDash \Delta$, *and* $\boldsymbol{\mathcal{N}}(g)$ *returns* $n \Rightarrow G$, *if*

(1) $g \sqsubseteq g'$, *where* $\bullet; \bullet \vdash g' : \tau'$ *and* $\boldsymbol{\mathcal{N}}(g')$ *returns* $n' \Rightarrow G'$; *and*
(2) $\forall \alpha \in \mathrm{dom}(\gamma), \forall |w_1| \leq \mathbb{n}, w_1 \in \gamma(\alpha)$ *if and only if* $G' \vdash \alpha \rightsquigarrow^* w_1$; *and*
(3) $\forall \alpha \in \mathrm{dom}(\delta), \forall |w_2| < \mathbb{n}, w_2 \in \delta(\alpha)$ *if and only if* $G' \vdash \alpha \rightsquigarrow^* w_2$,

then $\forall w \leq \mathbb{n}, w \in [\![g]\!]_{(\gamma,\delta)}$ *if and only if and* $G' \vdash n \rightsquigarrow^* w$.

Proof. By first induction on $\mathbb{n}$. The base case of 0 is trivial. In the inductive case, we have that the lemma holds for $|w| < \mathbb{n}$, and we want to prove it for $|w| \leq \mathbb{n}$.

Now we perform induction on g.

- The cases for $g = t$, $g = \epsilon$, and $g = \bot$ are straightforward.
- $g = \alpha$. Then $\boldsymbol{\mathcal{N}}(\alpha) = n \Rightarrow n \to \alpha$. By Lemma F.2, we know $(n \to \alpha) \in G'$, and there is no other production for n in G'.
  Since g is well-typed, it must be $\alpha \in \mathrm{dom}(\Gamma)$, and thus $\alpha \in \mathrm{dom}(\gamma)$. Then $[\![g]\!]_{(\gamma,\delta)} = \gamma(\alpha)$.
  As given, we know that $\forall |w| \leq \mathbb{n}, w \in \gamma(\alpha)$ if and only if $G' \vdash \alpha \rightsquigarrow^* w$.
  Since we know $(n \to \alpha) \in G'$, we have $\forall |w| \leq \mathbb{n}, w \in \gamma(\alpha)$ if and only if $G' \vdash n \rightsquigarrow^* w$.
- $g = g_1 \vee g_2$. Then $[\![g_1 \vee g_2]\!]_{(\gamma,\delta)} = [\![g_1]\!]_{(\gamma,\delta)} \cup [\![g_2]\!]_{(\gamma,\delta)}$
  We have
  $\{n \to N_1 \mid n_1 \to N_1 \in G_1\} \cup \{n \to N_2 \mid n_2 \to N_2 \in G_2\} \cup G_1 \cup G_2$
  $\boldsymbol{\mathcal{N}}(g_1) = n_1 \Rightarrow G_1$
  $\boldsymbol{\mathcal{N}}(g_2) = n_2 \Rightarrow G_2$
  The goal follows from I.H. on g_1 and g_2.
- $g = g_1 \cdot g_2$. Then $[\![g_1 \cdot g_2]\!]_{(\gamma,\delta)} = \{w_1 \cdot w_2 \mid w_1 \in [\![g_1]\!]_{(\gamma,\delta)} \wedge w_2 \in [\![g_2]\!]_{(\gamma,\delta)}\}$.
  According to $\boldsymbol{\mathcal{N}}$, we have
  $\{n \to N_1\ n_2 \mid n_1 \to N_1 \in G_1\} \cup G_1 \cup G_2$
  $\boldsymbol{\mathcal{N}}(g_1) = n_1 \Rightarrow G_1$
  $\boldsymbol{\mathcal{N}}(g_2) = n_2 \Rightarrow G_2$
  According to typing, we have
  $\Gamma; \Delta \vdash g_1 : \tau_1$
  $\Gamma, \Delta; \bullet \vdash g_2 : \tau_2$
  By I.H. on g_1, we have
  $\forall |w_1| \leq \mathbb{n}, w_1 \in [\![g_1]\!]_{(\gamma,\delta)}$ if and only if $G' \vdash n_1 \rightsquigarrow^* w_1$.
  By I.H. on g_2, we have the following. Here we use $<$ instead of $\leq$ as its typing context Γ, Δ includes Δ that only has interpretations for $|w_2| < \mathbb{n}$.
  $\forall |w_2| < \mathbb{n}, w_2 \in [\![g_2]\!]_{(\gamma,\delta)}$ if and only if $G' \vdash n_2 \rightsquigarrow^* w_2$.
  We first prove the conclusion from left to right. Given $w \leq \mathbb{n}$, and $w \in [\![g_1 \cdot g_2]\!]_{(\gamma,\delta)}$, it must be $w = w_1 \cdot w_2$ and $w_1 \in [\![g_1]\!]_{(\gamma,\delta)}$ and $w_2 \in [\![g_2]\!]_{(\gamma,\delta)}$. As $g_1 \cdot g_2$ is well-typed, we know $\tau_1.\textsc{Null} = \mathrm{false}$, so w_1 cannot be empty, and thus w_2 must have length $< \mathbb{n}$. So following I.H., and that n represents the same language as $n_1\ n_2$, we have $G' \vdash n \rightsquigarrow^* w_1 \cdot w_2$.
  Now we move to the conclusion from right to left. Given $G' \vdash n \rightsquigarrow^* w$, it must be $w = w_1 \cdot w_2$, and $G' \vdash n_1 \rightsquigarrow^* w_1$, and $G' \vdash n_2 \rightsquigarrow^* w_2$. As $g_1 \cdot g_2$ is well-typed, we know that $\tau_1.\textsc{Null} = \mathrm{false}$, so by Lemma 4.2, w_1 cannot be empty, and thus w_2 must have length $< \mathbb{n}$. So following I.H., we have $w_1 \in [\![g_1]\!]_{(\gamma,\delta)}$, and $w_2 \in [\![g_2]\!]_{(\gamma,\delta)}$, and thus $w \in [\![g_1 \cdot g_2]\!]_{(\gamma,\delta)}$.

- $g = \mu\alpha.\ g_1$. Then $[\![\mu\alpha.\ g_1]\!]_{(\gamma,\delta)} = [\![g_1]\!]_{(\gamma,\delta,[\![\mu\alpha.\ g_1]\!]_{(\gamma,\delta)}/\alpha)}$.
  We have
  $\boldsymbol{\mathcal{N}}(\mu\alpha : \tau.\ g_1) = \alpha \Rightarrow \{\alpha \to N \mid n \to N \in G\} \cup G$
  $\boldsymbol{\mathcal{N}}(g_1) = n \Rightarrow G$
  According to typing, we have $\Gamma; \Delta, \alpha : \tau \vdash g_1 : \tau$.
  According to the I.H. on $\mathbb{n}$, we have
  $\forall w' < \mathbb{n}, w' \in [\![\mu\alpha.\ g_1]\!]_{(\gamma,\delta)}$ if and only if and $G' \vdash \alpha \rightsquigarrow^* w'$.
  We have $(\gamma, \delta, [\![\mu\alpha.\ g_1]\!]_{(\gamma,\delta)}/\alpha)(\alpha) = [\![\mu\alpha.\ g_1]\!]_{(\gamma,\delta)}$.
  That means we have
  $\forall \beta \in \mathrm{dom}\,(\delta, [\![\mu\alpha.\ g_1]\!]_{(\gamma,\delta)}/\alpha)$,
  $\forall |w'| < \mathbb{n}, w' \in (\delta, [\![\mu\alpha.\ g_1]\!]_{(\gamma,\delta)}/\alpha)(\beta)$ if and only if $G' \vdash beta \rightsquigarrow^* w'$.
  Now by I.H. on g_1,
  $\forall w \le \mathbb{n}, w \in [\![g_1]\!]_{(\gamma,\delta,[\![\mu\alpha.\ g_1]\!]_{(\gamma,\delta)}/\alpha)}$ if and only if $G' \vdash n \rightsquigarrow^* w$
  equivalent to
  $\forall w \le \mathbb{n}, w \in [\![\mu\alpha.\ g_1]\!]_{(\gamma,\delta)}$ if and only if $G' \vdash n \rightsquigarrow^* w$.
  We have $\alpha \to N \in \boldsymbol{\mathcal{N}}(\mu\alpha.\ g_1)$, where $n \to N \in \boldsymbol{\mathcal{N}}(\mu\alpha.\ g_1)$. By Lemma F.2, we have $\alpha \to N \in G'$ and there is no other productions for α.
  Therefore,
  $\forall w \le \mathbb{n}, w \in [\![\mu\alpha.\ g_1]\!]_{(\gamma,\delta)}$ if and only if $G' \vdash \alpha \rightsquigarrow w$.

□

Theorem 4.8 (Soundness). *Given $\bullet;\bullet \vdash g : \tau$ and $\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow G$, we have $w \in [\![g]\!]_\bullet$ if and only if $G \vdash n \rightsquigarrow w$ for any w.*

Proof. Follows by Lemma F.3, making use of Lemma F.1. □