Making syntaxes LL1 through transformations

Bachelor project report

Noé De Santo

The goal of this Bachelor project was to transform syntaxes into equivalent LL1 syntaxes. This was done in an effort to allow efficient parsing algorithms/libraries to accept a broader set of syntaxes, as well as allowing users of said libraries to write their syntaxes in a readable and natural way, instead of focusing on making it LL1.

Syntaxes are different, yet equivalent, to formal grammars, with the advantage of being value-aware. More precisely, the transformations were implemented upon the ScaLL1on library [2], and the theory is based upon the associated formalism [1].

1 Definitions, notations & recalls

1.1 Lists/Sequences

The type of a list whose elements are of type T will be noted $\langle T \rangle$.

1.2 Tuples

Tuples will be noted using $[\dots]$ (instead of the usual $(\dots)$).

This is done in order to make function application on tuples more readable, as in $\varphi([a, b, c])$ instead of $\varphi((a, b, c))$.

To lighten some definitions and notations, we will also add two properties to tuples:

1. We will note $\tilde{\star}$ the point-wise application of $\star$ over elements of two tuples of same size.

   More formally, let $[t_1, \dots, t_n] : [T_1, \dots, T_n]$ and $[s_1, \dots, s_n] : [S_1, \dots, S_n]$ be two n-tuples.

   Then $\forall \star$ such that $\star : (T_i, S_i) \to V_i \quad \forall i \in 1, \dots, n$ we define

   $$\tilde{\star} : ([T_1, \dots, T_n], [S_1, \dots, S_n]) \to [V_1, \dots, V_n]$$

   as

   $$[t_1, \dots, t_n] \,\tilde{\star}\, [s_1, \dots, s_n] := [t_1 \star s_1, \dots, t_n \star s_n]$$

2. Let $[t_1, \dots, t_n] : [T, \dots, T]$ be a tuple whose elements all have the same type. Then we will consider that

   $$[t_1, \dots, t_n] : \langle T \rangle$$

   (i.e. the tuple is a sequence of T).

As those two properties are not usual, we will warn before using them.

1.3 Formal grammars

For a formal grammar $[\mathcal{N}, \mathcal{T}, R, S]$ (Non-terminals, Terminals, Rules, Start symbol), we define $\Sigma = (\mathcal{N} \cup \mathcal{T})$, the set of its symbols.

We will denote the elements of $\mathcal{N}$ by upper-case Latin letters, the elements of $\mathcal{T}$ by lower-case Latin letters and the elements of Σ^* by lower-case Greek letters.

1.4 Syntaxes

Definition 1 — **Fundamental sets** (from [2, Sections 3.1, 3.2])

- $\mathcal{T}$ is the set of all possible types;
- `Token` is the set of all tokens, and `Token`* is the set of all token sequences;
- $\mathcal{K}$ is the set of all kinds; each token t has exactly one kind assigned to it: `getKind`$(t) \in \mathcal{K}$;
- $\mathcal{S}_T$ is the set of all syntaxes that associate values of type $T \in \mathcal{T}$ to token sequences.

Definition 2 — Semantics of syntaxes (from [1, Page 54])

$$\text{MElem}\,\frac{k = \texttt{getKind}(t)}{elem_k \vdash \langle t \rangle \leadsto t} \qquad \text{MEps}\,\frac{}{\varepsilon_v \vdash \langle \rangle \leadsto v}$$

$$\text{MDisL}\,\frac{s_1 \vdash ts \leadsto v}{s_1 \vee s_2 \vdash ts \leadsto v} \qquad \text{MDisR}\,\frac{s_2 \vdash ts \leadsto v}{s_1 \vee s_2 \vdash ts \leadsto v}$$

$$\text{MSeq}\,\frac{s_1 \vdash ts_1 \leadsto v_1 \qquad s_2 \vdash ts_2 \leadsto v_2}{s_1 \cdot s_2 \vdash ts_1 \mathbin{++} ts_2 \leadsto [v_1, v_2]}$$

$$\text{MMap}\,\frac{s \vdash ts \leadsto v}{f \circledcirc s \vdash ts \leadsto f(v)} \qquad \text{MVar}\,\frac{s = \texttt{getDef}(x) \qquad s \vdash ts \leadsto v}{var_x \vdash ts \leadsto v}$$

Definition 3 — Many combinator

The $\texttt{many} : \mathcal{S}_T \to \mathcal{S}_{\langle T \rangle}$ combinator takes a syntax s as an argument, and returns another syntax which matches any sequence of sequences matched by s, and associates to it a sequence made of the values s associated to each of those sequences.

The sequences it matches (and the values it assigns to them) can be defined inductively as follows:

$$\frac{}{\texttt{many}(s) \vdash \langle \rangle \leadsto \langle \rangle} \qquad \frac{s \vdash ts_1 \leadsto v \qquad \texttt{many}(s) \vdash ts_2 \leadsto ls}{\texttt{many}(s) \vdash ts_1 \mathbin{++} ts_2 \leadsto v :: ls}$$

Note on the environment In the original thesis [1], an environment Γ is used to map identifiers to syntaxes. For simplicity, we will use one single global environment in this report instead. Additionally, we will sometimes need to define new mappings in the environment. We will then make the assumption that the identifier of any new mapping is not yet associated to anything (otherwise, simply change the identifier of the newly defined variable, as the exact identifier is irrelevant), and note new definitions (e.g. to map the identifier x to body v) as follows:

$$\texttt{getDef}(x) := v$$

1.5 Transformations

Definition 4 — Syntax equivalence

Let $T \in \mathcal{T}$ $s_1, s_2 \in \mathcal{S}_T$ be two syntaxes. Those syntaxes are said to be equivalent if

$$s_1 \vdash ts \leadsto v \iff s_2 \vdash ts \leadsto v \quad \forall ts \in \texttt{Token}^*,\ \forall v \in T$$

We will denote this relation by $s_1 \equiv s_2$

Note: As mentioned above, we will use one single global environment Γ. As such, $e_1 \equiv e_2$ in this report is equivalent to $e_1 \equiv_\Gamma e_2$ in [1].

Definition 5 — Equivalence preserving transformation

Let $T \in \mathcal{T}$ and $f \in \mathcal{S}_T \to \mathcal{S}_T$ be a partial function. f is said to be (an) equivalence preserving (transformation) if

$$s \equiv f(s) \quad \forall s \in \mathcal{S}_T \text{ where } f \text{ is defined}$$

We will see later that some transformations cannot be applied to all syntaxes. This is why equivalence preserving transformations are defined as partial functions. They can then be undefined on the syntaxes they cannot be applied to.

Lemma 1: Basic equivalence preserving transformations. (from [1, Page 62])

$$\begin{array}{rcl|l}
\bot \vee e & \equiv & e & \text{Left-unit of disjunction} \\
e \vee \bot & \equiv & e & \text{Right-unit of disjunction} \\
e_1 \vee (e_2 \vee e_3) & \equiv & (e_1 \vee e_2) \vee e_3 & \text{Associativity of disjunction} \\
(e_1 \cdot e) \vee (e_2 \cdot e) & \equiv & (e_1 \vee e_2) \cdot e & \text{Right-factoring} \\
f \circledcirc (e_1 \vee e_2) & \equiv & (f \circledcirc e_1) \vee (f \circledcirc e_2) & \text{Distributy of map over disjunctions}
\end{array}$$

We will now introduce the concept of syntax splitter. A splitter takes a syntax, and returns a list of syntaxes, such that their disjunction results in a syntax equivalent to the original one. This concept might appear a bit weird at first, but will be useful to define some equivalence preserving transformations later on.

Definition 6 — Syntax splitter

Let $T \in \mathcal{T}$ and $f \in \mathcal{S}_T \to \langle \mathcal{S}_T \rangle$ be a function. f is said to be a syntax splitter if

$$\left(\bigvee_{s' \in f(s)} s' \right) \equiv s$$

2 Left factorization for grammars

We will now define (informally) left factorization on formal grammars. We will go through multiple definitions, each one adding some useful properties for our usage.

The reason why we don't simply introduce the final definition is because we find it is the best way to explain why such a definition was reached, and what the shortcomings solved by this definition were.

It is interesting to note that most references we found use the first definition we will introduce, and their definition is as informal as ours (e.g a book on parsing [4, Page 252], a paper on "syntaxes improvement" [3, Page 33]), which was an additional incentive to build our definition upon it.

"Spontaneous idea" remove the specified prefix α from all alternatives, and create a new non-terminal with α as a prefix.

$$A ::= \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \quad \xrightarrow{\texttt{leftFactorize}(\alpha,A)} \quad \begin{array}{ll} A & ::= \alpha A_f \\ A_f & ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{array}$$

Generalized The previous definition is a bit restricting. In particular, it only allows to factorize non-terminals where all alternatives have a common prefix. This leads to another definition which lifts this limitation, where the factorization is applied to all the alternatives which allow it, and the other alternatives are left as they are.

$$A ::= \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m \quad \xrightarrow{\texttt{leftFactorize}(\alpha,A)} \quad \begin{array}{ll} A & ::= \alpha A_f \mid A_a \\ A_f & ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \\ A_a & ::= \gamma_1 \mid \ldots \mid \gamma_m \end{array}$$

where $\forall \gamma_i \neg \exists \delta_i \in \Sigma^* : \alpha\delta_i = \gamma_i$ (i.e. α is not a prefix of any γ_i).

We call A_f the resulting "follow" and A_a the resulting "alternative".

Why not recursive? Consider the following grammar (and its factorization)

$$\begin{array}{ll} A & ::= \alpha\beta \mid B\gamma \\ B & ::= \alpha\delta \end{array} \quad \xrightarrow{\texttt{leftFactorize}(\alpha,A)} \quad \begin{array}{ll} A & ::= \alpha A_f \mid A_a \\ A_f & ::= \beta \\ A_a & ::= B\gamma \\ B & ::= \alpha\delta \end{array}$$

The fact that the factorization was not applied recursively to B might or might not seem natural, but in the context of LL1 conflicts, applying it recursively would be beneficial, as the resulting grammar would not longer contain a first/first conflict anymore.

$$\begin{array}{ll} A & ::= \alpha\beta \mid B\gamma \\ B & ::= \alpha\delta \end{array} \quad \xrightarrow{\texttt{leftFactorize}(\alpha,A)} \quad \begin{array}{ll} A & ::= \alpha A_f \\ A_f & ::= \beta \mid B_f\gamma \\ B_f & ::= \delta \end{array}$$

This final version also behaves more similarly to the definition which will be used for syntaxes.

3 Intermission: Left factor out

Before defining left factorization for syntaxes, we will first introduce left factor out, which is a slightly different operation, which will be needed to define left-factoring on syntaxes.

3.1 For grammars

The goal here is to remove the prefix from the grammar, instead of just "emphasizing it" as in left-factoring. As such, instead of generating an equivalent non-terminal, only the resulting follow and alternative are generated.

$$A ::= \alpha\beta_1|\alpha\beta_2|\dots|\alpha\beta_n|\gamma_1|\dots|\gamma_m \qquad \xrightarrow{\texttt{leftFactorOut}(\alpha,A)} \qquad \begin{array}{ll} A_f & ::= \beta_1|\beta_2|\dots|\beta_n \\ A_a & ::= \gamma_1|\dots|\gamma_m \end{array}$$

3.2 For syntaxes

One of the main differences between syntaxes and grammars is that the results of syntaxes have a fixed and known type. This implies that the left factor out operation, when seen as a (partial) function, should also have a fixed (yet generic) type.

It might be tempting to "chop out" some part of the original type, e.g., we might want

$$\texttt{leftFactorOut} : \left(\mathcal{S}_{\texttt{Token}}, \mathcal{S}_{[\texttt{Token},[\texttt{Token},\texttt{Token}]]}\right) \to \mathcal{S}_{[\texttt{Token},\texttt{Token}]}$$

so that

$$\texttt{leftFactorOut}(\mathit{elem}_a, \mathit{elem}_a \cdot (\mathit{elem}_b \cdot \mathit{elem}_c)) = \mathit{elem}_b \cdot \mathit{elem}_c$$

This solution might seem nice and quite natural, but those two qualifiers become unsuited when considering slightly more complicated examples, such as the two following (assume $f : (\texttt{Token}, \texttt{Token}) \to T$ for some $T \in \mathcal{T}$)

$$\texttt{leftFactorOut}(\mathit{elem}_a, (\mathit{elem}_a \cdot \mathit{elem}_b) \cdot \mathit{elem}_c) \qquad \texttt{leftFactorOut}(\mathit{elem}_a, f \circledcirc (\mathit{elem}_a \cdot \mathit{elem}_b))$$

The problem in both cases is simply that we don't have the "nice type schema" we had before, which had the form

$$\texttt{leftFactorOut} : \left(\mathcal{S}_P, \mathcal{S}_{[P,F]}\right) \to \mathcal{S}_F \qquad \text{for some } P, F \in \mathcal{T}$$

In the first example, we could break the type further down to get a similar result, but this solution is clearly not applicable to the second one, as there is no relation whatsoever between `Token` and T.

The best (and in fact only) solution we found to this issue is to make `leftFactorOut` return a function which needs the "chopped out" part in order to return a complete value:

$$\texttt{leftFactorOut} : (\mathcal{S}_P, \mathcal{S}_S) \to \mathcal{S}_{(P \to S)}$$

This idea allows the following definition of `leftFactorOut`

Definition 7 — Left factor out

`leftFactorOut` $\in (\mathcal{S}_P, \mathcal{S}_S) \to \mathcal{S}_{P \to S}$ is defined as follows:

$$\begin{aligned} \texttt{leftFactorOut}(t, t) &:= \varepsilon_{x \to x} \\ \texttt{leftFactorOut}(t, s_1 \vee s_2) &:= \texttt{leftFactorOut}(t, s_1) \vee \texttt{leftFactorOut}(t, s_2) \\ \texttt{leftFactorOut}(t, s_1 \cdot s_2) &:= \sigma_l \circledcirc (\texttt{leftFactorOut}(t, s_1) \cdot s_2) \\ \texttt{leftFactorOut}(t, f \circledcirc s) &:= (x \to f \circ x) \circledcirc \texttt{leftFactorOut}(t, s) \\ \texttt{leftFactorOut}(t, \mathit{var}_x) &:= \mathit{var}_{x_{\text{lf}}} \end{aligned}$$

where $\sigma_l([l, r]) := x \to [l(x), r]$ and $\texttt{getDef}(x_{\text{lf}}) := \texttt{leftFactorOut}(t, \texttt{getDef}(x))$

When `leftFactorOut` is called with arguments which do not fall in any of the above cases, it is undefined.

3.3 Handling failure

This definition has the same shortcoming as the first definition of left factorization on grammars: it assumes that the factorization succeeds on all alternatives (in fact, the factorization might even fail on all alternatives). Here are two examples showing this:

$$\texttt{leftFactorOut}(\mathit{elem}_a, \mathit{elem}_b) = ??? \qquad \texttt{leftFactorOut}(\mathit{elem}_a, \mathit{elem}_a \vee \mathit{elem}_b) = ???$$

A simple solution to handle the possibility of a failing factorization is to define it like this instead

$$\texttt{leftFactorOut} : (\mathcal{S}_P, \mathcal{S}_S) \to [\mathcal{S}_{(P \to S)}, \mathcal{S}_S]$$

where the first value returned is the successful factorization as mentioned before, and the second value is the part where factorization was impossible.

$$\texttt{leftFactorOut}(\mathit{elem}_a, \underbrace{\mathit{elem}_b}_{\text{fails}}) = [\bot, \mathit{elem}_b] \qquad \texttt{leftFactorOut}(\mathit{elem}_a, \underbrace{\mathit{elem}_a}_{\text{succeeds}} \vee \underbrace{\mathit{elem}_b}_{\text{fails}}) = [\varepsilon_{x \to x}, \mathit{elem}_b]$$

Failure handling is necessary to make the operation usable in practice. We will nevertheless ignore it in the proof section of this document, as this greatly increases the number of cases to consider in the subsequent proofs, without introducing any food for thought.

Definition 8 — Left factor out (with failure handling)

This definition makes use of the additional properties of tuples defined in section 1.2.

$\texttt{leftFactorOut} \in (\mathcal{S}_P, \mathcal{S}_S) \to [\mathcal{S}_{(P \to S)}, \mathcal{S}_S]$ is defined as follows:

$$\begin{aligned}
\texttt{leftFactorOut}(t, t) &:= [\varepsilon_{x \to x}, \bot] \\
\texttt{leftFactorOut}(t, \mathit{elem}_k) &:= [\bot, \mathit{elem}_k] \\
\texttt{leftFactorOut}(t, \varepsilon_v) &:= [\bot, \varepsilon_v] \\
\texttt{leftFactorOut}(t, s_1 \vee s_2) &:= \texttt{leftFactorOut}(t, s_1) \mathbin{\tilde{\vee}} \texttt{leftFactorOut}(t, s_2) \\
\texttt{leftFactorOut}(t, s_1 \cdot s_2) &:= [\sigma_l, x \to x] \mathbin{\tilde{\circledcirc}} (\texttt{leftFactorOut}(t, s_1) \mathbin{\tilde{\cdot}} [s_2, s_2]) \\
\texttt{leftFactorOut}(t, f \circledcirc s) &:= [(x \to f \circ x), f] \mathbin{\tilde{\circledcirc}} \texttt{leftFactorOut}(t, s) \\
\texttt{leftFactorOut}(t, \mathit{var}_x) &:= [\mathit{var}_{x_{\mathrm{lf}}}, \mathit{var}_{x_{\mathrm{la}}}]
\end{aligned}$$

where $\sigma_l([l, r]) := x \to [l(x), r]$ and $[\texttt{getDef}(x_{\mathrm{lf}}), \texttt{getDef}(x_{\mathrm{la}})] := \texttt{leftFactorOut}(t, \texttt{getDef}(x))$

3.4 Non-recursive version

We will also define a version of `leftFactorOut` which does not visit the variables' definition, called `leftFactorSymbolOut`, which will also be useful later on.

Definition 9 — Left factor symbol out

$\texttt{leftFactorSymbolOut} \in (\mathcal{S}_P, \mathcal{S}_S) \to [\mathcal{S}_{(P \to S)}, \mathcal{S}_S]$ is defined as follows:

$$\begin{aligned}
\texttt{leftFactorSymbolOut}(t, t) &:= [\varepsilon_{x \to x}, \bot] \\
\texttt{leftFactorSymbolOut}(t, \mathit{elem}_k) &:= [\bot, \mathit{elem}_k] \\
\texttt{leftFactorSymbolOut}(t, \varepsilon_v) &:= [\bot, \varepsilon_v] \\
\texttt{leftFactorSymbolOut}(t, s_1 \vee s_2) &:= \texttt{leftFactorSymbolOut}(t, s_1) \vee \texttt{leftFactorSymbolOut}(t, s_2) \\
\texttt{leftFactorSymbolOut}(t, s_1 \cdot s_2) &:= [\sigma_l, x \to x] \circledcirc (\texttt{leftFactorSymbolOut}(t, s_1) \cdot [s_2, s_2]) \\
\texttt{leftFactorSymbolOut}(t, f \circledcirc s) &:= [(x \to f \circ x), f] \circledcirc \texttt{leftFactorSymbolOut}(t, s) \\
\texttt{leftFactorSymbolOut}(t, \mathit{var}_x) &:= [\bot, \mathit{var}_x]
\end{aligned}$$

where $\sigma_l([l, r]) := x \to [l(x), r]$.

3.5 "Strength" of the factorization

Consider the syntax $s := \varepsilon_v \cdot \mathit{elem}_k$. When left factor out (with failure handling) is applied, the result is

$$\texttt{leftFactorOut}(\mathit{elem}_k, s) = [\bot, \varepsilon_v \cdot \mathit{elem}_k]$$

As we will use left-factoring as a conflict solving tool, the fact that elem_k was not factored out is problematic. It is interesting to note that this issue is usually not addressed when considering grammars, as ε can only appear alone in formal grammars (and, as such, cannot prefix (non-)terminals), and the factorization is usually not applied recursively.

This tends to show that, in the context of syntaxes, there is seemingly not one `leftFactorOut` definition for the sequence case, but multiple possible definitions with different behaviors when ε is the first element of the sequence. In our case, we would like elem_k to be factored even if it is prefixed by multiple ε.

An alternative definition would then be

$$\texttt{leftFactorOut}(t, s_1 \cdot s_2) := \begin{cases} \sigma_r \circledcirc (s_1 \cdot \texttt{leftFactorOut}(t, s_2)) & \text{if } s_1 \text{ is null (i.e. only matches } \langle\rangle) \\ \sigma_l \circledcirc (\texttt{leftFactorOut}(t, s_1) \cdot s_2) & \text{otherwise} \end{cases}$$

(recall that $\sigma_l([l, r]) := x \to [l(x), r]$ and $\sigma_r([l, r]) := x \to [l, r(x)]$).

But this would still “fail” on $s = (\mathit{elem}_l \vee \varepsilon_v) \cdot \mathit{elem}_k$

$$\texttt{leftFactorOut}(\mathit{elem}_k, s) = [\mathit{elem}_k \cdot \mathit{elem}_k, \varepsilon_v \cdot \mathit{elem}_k]$$

The “strongest” definition I came up with is the following

$$\begin{aligned} \texttt{leftFactorOut}(t, s_1 \cdot s_2) := {} & \sigma_l \circledcirc (\texttt{leftFactorOut}(t, \texttt{notNullPart}(s_1)) \cdot s_2) \\ & \vee \sigma_r \circledcirc (\texttt{nullPart}(s_1) \cdot \texttt{leftFactorOut}(t, s_2)) \end{aligned}$$

For simplicity, the proofs will use the original (weakest & without failure handling) definition.

4 Left factorization for Syntaxes

Now that `leftFactorOut` is defined, `leftFactorize` can easily be defined.

Definition 10 — Left factorization

Let $\varphi \in [T, (T \to S)] \to S$ defined by $\varphi([x, f]) := f(x)$. Then

$$\texttt{leftFactorize}(t, s) := \varphi \circledcirc (t \cdot \texttt{leftFactorOut}(t, s))$$

When $\texttt{leftFactorOut}(t, s)$ is undefined, then so is $\texttt{leftFactorize}(t, s)$ (recall that a transformation is a partial function!) and we say that the factorization does not succeed.

We also give the definition of `leftFactorize` when `leftFactorOut` handles failure.

Definition 11 — Left factorization (with failure handling)

Let $\varphi \in [T, (T \to S)] \to S$ defined by $\varphi([x, f]) := f(x)$. Then

$$\begin{aligned} & [s_f, s_a] := \texttt{leftFactorOut}(t, s) \\ & \texttt{leftFactorize}(t, s) := (\varphi \circledcirc (t \cdot s_f)) \vee s_a \end{aligned}$$

4.1 Correctness

Lemma 2: `leftFactorOut`'s soundness. Let $s \in \mathcal{S}_W$ and $t \in \mathcal{S}_V$.If

$$t \vdash ts_1 \rightsquigarrow v \qquad \textbf{and} \qquad \texttt{leftFactorOut}(t, s) \vdash ts_2 \rightsquigarrow f$$

where $v : V$, then $f : V \to W$ and $s \vdash ts_1 \mathbin{++} ts_2 \rightsquigarrow f(v)$

Proof. (We will only consider the case where the factorization succeeds).

We will prove this by induction on the derivation of $\texttt{leftFactorOut}(t, s) \vdash ts_2 \rightsquigarrow f$.

- *Case* `MElem`: $\texttt{leftFactorOut}(t, s)$ is bound to be elem_k.

  By definition of `leftFactorOut`, such an expression is never returned, and as such, this case cannot happen.

- *Case* `MEps`: $\texttt{leftFactorOut}(t, s)$ is bound to be ε_f for some f.

  By definition of `leftFactorOut`, the only inputs resulting in such an expression are the ones verifying $t = s$, and as such $f = x \to x$, $ts_2 = \langle\rangle$.

  Then, for any $v : t \vdash ts_1 \rightsquigarrow v$, we have

  $$s \vdash ts_1 \mathbin{++} \langle\rangle \rightsquigarrow v = (x \to x)(v) = f(v)$$

- *Case* `MDisL`: $\texttt{leftFactorOut}(t, s)$ is bound to be $s'_1 \vee s'_2$.

  By definition of `leftFactorOut`, the only inputs resulting in such an expression are the ones where $s = s_1 \vee s_2$, and as such $s'_1 = \texttt{leftFactorOut}(t, s_1)$, $s'_2 = \texttt{leftFactorOut}(t, s_2)$.

  By induction hypothesis, if

  $$t \vdash ts_1 \rightsquigarrow v \qquad \texttt{leftFactorOut}(t, s_1) \vdash ts_2 \rightsquigarrow f$$

  then

  $$s_1 \vdash ts_1 \mathbin{++} ts_2 \rightsquigarrow f(v)$$

  By applying the inductive predicates, we get

  $$\text{MDisL}\frac{\texttt{leftFactorOut}(t, s_1) \vdash ts_2 \rightsquigarrow f}{\texttt{leftFactorOut}(t, s_1 \vee s_2) \vdash ts_2 \rightsquigarrow f}$$

  because $\texttt{leftFactorOut}(t, s_1 \vee s_2) = \texttt{leftFactorOut}(t, s_1) \vee \texttt{leftFactorOut}(t, s_2)$.

  And we can directly verify that

  $$\text{MDisL}\frac{s_1 \vdash ts_1 \mathbin{++} ts_2 \rightsquigarrow f(v)}{s \vdash ts_1 \mathbin{++} ts_2 \rightsquigarrow f(v)}$$

- *Case* `MDisR`: Similar to MDisL.
- *Case* `MSeq`: $\texttt{leftFactorOut}(t, s)$ is bound to be $s_1 \cdot s_2$.

  By definition of `leftFactorOut`, such an expression is never returned, and as such, this case cannot happen.

- *Case* `MMap`: $\texttt{leftFactorOut}(t, s)$ is bound to be $\varphi \circledcirc s'$.

  The definition of `leftFactorOut` then allows two different possibilities depending on s

  - If $s = s_1 \cdot s_2$, then $\varphi = \sigma_l$ and $s' = \texttt{leftFactorOut}(t, s_1) \cdot s_2$.

    By induction hypothesis, if

    $$t \vdash ts_1 \rightsquigarrow v \qquad \texttt{leftFactorOut}(t, s_1) \vdash ts_2 \rightsquigarrow f$$

    then

    $$s_1 \vdash ts_1 \mathbin{++} ts_2 \rightsquigarrow f(v)$$

    By applying the inductive predicates, we get

    $$\text{MMap}\frac{\text{MSeq}\dfrac{\texttt{leftFactorOut}(t, s_1) \vdash ts_2 \rightsquigarrow f \qquad s_2 \vdash ts_3 \rightsquigarrow u}{\texttt{leftFactorOut}(t, s_1) \cdot s_2 \vdash ts_2 \mathbin{++} ts_3 \rightsquigarrow [f, u]}}{\texttt{leftFactorOut}(t, s) \vdash ts_2 \mathbin{++} ts_3 \rightsquigarrow x \to [f(x), u]}$$

    And we can directly verify that

    $$\text{MSeq}\frac{s_1 \vdash ts_1 \mathbin{++} ts_2 \rightsquigarrow f(v) \qquad s_2 \vdash ts_3 \rightsquigarrow u}{s \vdash ts_1 \mathbin{++} ts_2 \mathbin{++} ts_3 \rightsquigarrow [f(v), u] = (x \to [f(x), u])(v)}$$

  - If $s = g \circledcirc r$, then $\varphi = x \to g \circ x$ and $s' = \texttt{leftFactorOut}(t, r)$.

    By induction hypothesis, if

    $$t \vdash ts_1 \rightsquigarrow v \qquad \texttt{leftFactorOut}(t, r) \vdash ts_2 \rightsquigarrow f$$

then

$$r \vdash ts_1 +\!\!+\; ts_2 \leadsto f(v)$$

By applying the inductive predicates, we get

$$\text{MMap}\,\frac{\texttt{leftFactorOut}(t, r) \vdash ts_2 \leadsto f}{\varphi \circ \texttt{leftFactorOut}(t, r) \vdash ts_2 \leadsto (x \to g \circ x)(f) = g \circ f}$$

And we can directly verify that

$$\text{MMap}\,\frac{r \vdash ts_1 +\!\!+\; ts_2 \leadsto f(v)}{s \vdash ts_1 +\!\!+\; ts_2 \leadsto g(f(v)) = (g \circ f)(v)}$$

- *Case* `MVar`: $\texttt{leftFactorOut}(t, s)$ is bound to be $var_{x_{lf}}$.

  By definition of `leftFactorOut`, the only inputs resulting in such an expression are the ones verifying $s = var_x$, and as such $\texttt{getDef}(x_{lf}) = \texttt{leftFactorOut}(t, \texttt{getDef}(x))$.

  Let $s' = \texttt{getDef}(x)$.

  By inductive hypothesis, if

  $$t \vdash ts_1 \leadsto v \qquad \texttt{leftFactorOut}(t, s') \vdash ts_2 \leadsto f$$

  then

  $$s' \vdash ts_1 +\!\!+\; ts_2 \leadsto f(v)$$

  By applying the inductive predicates, we get

  $$\text{MVar}\,\frac{\texttt{leftFactorOut}(t, s') = \texttt{getDef}(x_{lf}) \qquad \texttt{leftFactorOut}(t, s') \vdash ts_2 \leadsto f}{var_{x_{lf}} \vdash ts_2 \leadsto f}$$

  And we can directly verify that

  $$\text{MVar}\,\frac{s' = \texttt{getDef}(x) \qquad s' \vdash ts_1 +\!\!+\; ts_2 \leadsto f(v)}{s \vdash ts_1 +\!\!+\; ts_2 \leadsto f(v)}$$

□

Lemma 3: `leftFactorOut`'s completeness. Let $s \in \mathcal{S}_W$ and $t \in \mathcal{S}_V$.If

$$s \vdash ts \leadsto w \qquad\qquad \textbf{and} \qquad\qquad \texttt{leftFactorOut}(t, s) \text{ succeeeds}$$

then $\exists ts_1, ts_2 \in \texttt{Token}^*$

$$ts = ts_1 +\!\!+\; ts_2 \quad \textbf{and} \quad t \vdash ts_1 \leadsto v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s) \vdash ts_2 \leadsto f \quad \textbf{and} \quad f(v) = w$$

Proof. We will first consider the case where $s = t$, and the proceed by structural induction on the derivation of $s \vdash ts \leadsto w$ for the remaining cases.

1. As $s = t$, we have $\texttt{leftFactorOut}(t, s) = \varepsilon_{x \to x}$.

   Then, $\forall ts \in \texttt{Token}^* : t \vdash ts \leadsto v$, we indeed have

   $$ts = ts +\!\!+\; \langle\rangle \quad \textbf{and} \quad t \vdash ts \leadsto v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s) \vdash \langle\rangle \leadsto x \to x \quad \textbf{and} \quad (x \to x)(v) = v$$

2. Induction on $s \vdash ts \leadsto w$:

   - *Case* `MElem`: The syntax s is bound to be $elem_k$ for some kind k, and as such $W = \texttt{Token}$.

     Since the factorization succeeded, $s = t$ (the factorization would fail otherwise), and as such this case was already handled previously.

   - *Case* `MEps`: The syntax s is bound to be ε_w for some $w : W$.

     Since the factorization succeeded, $s = t$ (the factorization would fail otherwise), and as such this case was already handled previously.

- *Case* `MDisL`: The syntax s is bound to be $s_1 \vee s_2$.

  From the inductive predicate, we abduct that $s_1 \vdash ts \rightsquigarrow w$.

  By induction hypothesis, $\exists ts_1, ts_2 \in \texttt{Token}^*$

  $$ts = ts_1 \mathbin{++} ts_2 \quad \textbf{and} \quad t \vdash ts_1 \rightsquigarrow v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s_1) \vdash ts_2 \rightsquigarrow f \quad \textbf{and} \quad f(v) = l$$

  Using the inductive predicates

  $$\text{MDisL}\,\frac{\texttt{leftFactorOut}(t, s_1) \vdash ts_2 \rightsquigarrow f}{\texttt{leftFactorOut}(t, s_1) \vee \texttt{leftFactorOut}(t, s_2) \vdash ts_2 \rightsquigarrow f}$$

  By noting that $\texttt{leftFactorOut}(t, s_1) \vee \texttt{leftFactorOut}(t, s_2) = \texttt{leftFactorOut}(t, s)$, we conclude that

  $$ts = ts_1 \mathbin{++} ts_2 \quad \textbf{and} \quad t \vdash ts_1 \rightsquigarrow v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s) \vdash ts_2 \rightsquigarrow f \quad \textbf{and} \quad f(v) = w$$

- *Case* `MDisR`: Similar to MDisL.

- *Case* `MSeq`: The syntax s is bound to be $s_1 \cdot s_2$.

  The value w is bound to be $[l, r]$, with $s_1 \vdash ts' \rightsquigarrow l$ and $s_2 \vdash ts_3 \rightsquigarrow r$.

  By induction hypothesis, $\exists ts_1, ts_2 \in \texttt{Token}^*$

  $$ts' = ts_1 \mathbin{++} ts_2 \quad \textbf{and} \quad t \vdash ts_1 \rightsquigarrow v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s_1) \vdash ts_2 \rightsquigarrow f \quad \textbf{and} \quad f(v) = l$$

  Using the inductive predicates

  $$\text{MMap}\,\frac{\text{MSeq}\,\dfrac{\texttt{leftFactorOut}(t, s_1) \vdash ts_2 \rightsquigarrow f \qquad s_2 \vdash ts_3 \rightsquigarrow r}{\texttt{leftFactorOut}(t, s_1) \cdot s_2 \vdash ts_2 \mathbin{++} ts_3 \rightsquigarrow [f, r]}}{\sigma_l \circledcirc (\texttt{leftFactorOut}(t, s_1) \cdot s_2) \vdash ts_2 \mathbin{++} ts_3 \rightsquigarrow \sigma_l([f, r])}$$

  By noting that $\sigma_l \circledcirc (\texttt{leftFactorOut}(t, s_1) \cdot s_2) = \texttt{leftFactorOut}(t, s)$, we conclude that

  $$\begin{aligned} & ts = ts_1 \mathbin{++} (ts_2 \mathbin{++} ts_3) && \textbf{and} \\ & t \vdash ts_1 \rightsquigarrow v && \textbf{and} \\ & \texttt{leftFactorOut}(t, s) \vdash ts_2 \mathbin{++} ts_3 \rightsquigarrow x \to [f(x), r] && \textbf{and} \\ & (x \to [f(x), r])(v) = [f(v), r] = [l, r] \end{aligned}$$

- *Case* `MMap`: The syntax s is bound to be $g \circledcirc s'$ for some $s' \in \mathcal{S}_U$ and $g \in U \to W$.

  The value w is bound to be $g(u)$ where $s' \vdash ts \rightsquigarrow u$.

  By induction hypothesis, $\exists ts_1, ts_2 \in \texttt{Token}^*$

  $$ts = ts_1 \mathbin{++} ts_2 \quad \textbf{and} \quad t \vdash ts_1 \rightsquigarrow v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s') \vdash ts_2 \rightsquigarrow f \quad \textbf{and} \quad f(v) = u$$

  Using the inductive predicates

  $$\text{MMap}\,\frac{\texttt{leftFactorOut}(t, s') \vdash ts_2 \rightsquigarrow f}{(x \to g \circ x) \circledcirc \texttt{leftFactorOut}(t, s') \vdash ts_1 \mathbin{++} ts_2 \rightsquigarrow (x \to g \circ x)(f) = g \circ f}$$

  By noting that $(g \circ f)(v) = g(f(v)) = g(u) = w$ and $(x \to g \circ x) \circledcirc \texttt{leftFactorOut}(t, s') = \texttt{leftFactorOut}(t, s)$, we conclude that

  $$ts = ts_1 \mathbin{++} ts_2 \quad \textbf{and} \quad t \vdash ts_1 \rightsquigarrow v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s) \vdash ts_2 \rightsquigarrow g \circ f \quad \textbf{and} \quad (g \circ f)(v) = w$$

- *Case* `MVar`: The syntax s is bound to be var_x for some identifier x.

  Let $s' := \texttt{getDef}(x)$. By definition

  $$s' \vdash ts \rightsquigarrow w \qquad \textbf{and} \qquad \texttt{leftFactorOut}(t, s') \text{ succeeeds}$$

as such, by induction hypothesis $\exists ts_1, ts_2 \in \texttt{Token}^*$:

$$ts = ts_1 \mathbin{++} ts_2 \quad \textbf{and} \quad t \vdash ts_1 \rightsquigarrow v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s') \vdash ts_2 \rightsquigarrow f \quad \textbf{and} \quad f(v) = w$$

Applying the inductive predicates

$$\text{MVar}\frac{\texttt{leftFactorOut}(t, s') := \texttt{getDef}(x_{lf}) \qquad \texttt{leftFactorOut}(t, s') \vdash ts_2 \rightsquigarrow f}{var_{x_{lf}} \vdash ts_2 \rightsquigarrow f}$$

By noting that $var_{x_{lf}} = \texttt{leftFactorOut}(t, s)$, we conclude that

$$ts = ts_1 \mathbin{++} ts_2 \quad \textbf{and} \quad t \vdash ts_1 \rightsquigarrow v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s) \vdash ts_2 \rightsquigarrow f \quad \textbf{and} \quad f(v) = w$$

□

Theorem 4. Left factorization is equivalence preserving, i.e. $s \equiv \texttt{leftFactorize}(t, s)$

Proof. We have to prove that

$$s \vdash ts \rightsquigarrow w \iff \texttt{leftFactorize}(t, s) \vdash ts \rightsquigarrow w$$

and will only consider the case where the factorization succeeds.

First direction: $\implies$

Let ts such that

$$s \vdash ts \rightsquigarrow w$$

By lemma 3, we have

$$ts = ts_1 \mathbin{++} ts_2 \quad \textbf{and} \quad t \vdash ts_1 \rightsquigarrow v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s) \vdash ts_2 \rightsquigarrow f \quad \textbf{and} \quad f(v) = w$$

By applying the inductive predicates

$$\text{MMap}\frac{\text{MSeq}\dfrac{t \vdash ts_1 \rightsquigarrow v \qquad \texttt{leftFactorOut}(t, s) \vdash ts_2 \rightsquigarrow f}{t \cdot \texttt{leftFactorOut}(t, s) \vdash ts_1 \mathbin{++} ts_2 \rightsquigarrow [v, f]}}{\texttt{leftFactorize}(t, s) \vdash ts_1 \mathbin{++} ts_2 \rightsquigarrow \varphi([v, f]) = f(v) = w}$$

So we conclude that

$$s \vdash ts \rightsquigarrow w \implies \texttt{leftFactorize}(t, s) \vdash ts \rightsquigarrow w$$

Second direction: $\impliedby$

Let ts such that

$$\texttt{leftFactorize}(t, s) \vdash ts \rightsquigarrow w$$

By definition of `leftFactorize`, there must $\exists ts_1, ts_2$:

$$ts = ts_1 \mathbin{++} ts_2 \quad \textbf{and} \quad t \vdash ts_1 \rightsquigarrow v \quad \textbf{and} \quad \texttt{leftFactorOut}(t, s) \vdash ts_2 \rightsquigarrow f \quad \textbf{and} \quad f(v) = w$$

By applying lemma 2, we get

$$s \vdash ts_1 \mathbin{++} ts_2 \rightsquigarrow f(v) = w$$

So we can conclude

$$\texttt{leftFactorize}(t, s) \vdash ts \rightsquigarrow w \implies s \vdash ts \rightsquigarrow w$$

□

5 Substitution

Substitution replaces all occurrences of a syntax by another provided syntax. This transformation is, in itself, not of any use, but will be really useful to replace all occurrences of a syntax containing conflicts by another one where the conflicts were solved.

Definition 12 — Substitute

$\texttt{substitute} : (\mathcal{S}_T, \mathcal{S}_T, \mathcal{S}_S) \rightarrow \mathcal{S}_S$ is defined as follows:

$$\begin{aligned}
\texttt{substitute}(t, s, t) &:= s \\
\texttt{substitute}(t, s, elem_k) &:= elem_k \\
\texttt{substitute}(t, s, \varepsilon_v) &:= \varepsilon_v \\
\texttt{substitute}(t, s, s_1 \cdot s_2) &:= \texttt{substitute}(t, s, s_1) \cdot \texttt{substitute}(t, s, s_2) \\
\texttt{substitute}(t, s, s_1 \vee s_2) &:= \texttt{substitute}(t, s, s_1) \vee \texttt{substitute}(t, s, s_2) \\
\texttt{substitute}(t, s, f \circledcirc s') &:= f \circledcirc \texttt{substitute}(t, s, s') \\
\texttt{substitute}(t, s, var_x) &:= var_{x_s}
\end{aligned}$$

where $\texttt{getDef}(x_s) := \texttt{substitute}(t, s, \texttt{getDef}(x))$

Theorem 5: (Unproven).

$$s \equiv t \implies \texttt{substitute}(t, s, r) \equiv r$$

6 Split by prefix

Definition 13 — Split by prefix

This definition makes use of the additional properties of tuples defined in section 1.2.

$\texttt{splitByPrefix} : (\mathcal{S}_P, \mathcal{S}_S) \rightarrow [\mathcal{S}_S, \mathcal{S}_S]$ is defined as follows:

$$\begin{aligned}
\texttt{splitByPrefix}(t, t) &:= [t, \bot] \\
\texttt{splitByPrefix}(t, elem_k) &:= [\bot, elem_k] \\
\texttt{splitByPrefix}(t, \varepsilon_v) &:= [\bot, \varepsilon_v] \\
\texttt{splitByPrefix}(t, s_1 \cdot s_2) &:= \texttt{splitByPrefix}(t, s_1) \mathbin{\tilde{\cdot}} [s_2, s_2] \\
\texttt{splitByPrefix}(t, s_1 \vee s_2) &:= \texttt{splitByPrefix}(t, s_1) \mathbin{\tilde{\vee}} \texttt{splitByPrefix}(t, s_2) \\
\texttt{splitByPrefix}(t, f \circledcirc s') &:= [f, f] \mathbin{\tilde{\circledcirc}} \texttt{splitByPrefix}(t, s') \\
\texttt{splitByPrefix}(t, var_x) &:= [\bot, var_x]
\end{aligned}$$

Theorem 6. $\texttt{splitByPrefix}$ is a syntax splitter, i.e.

$$\begin{aligned}
&[s_p, s_n] := \texttt{splitByPrefix}(t, s) \\
&s_p \vee s_n \equiv s
\end{aligned}$$

Proof. We will first consider the case where $s = t$, and the proceed by structural induction on the derivation of $s \vdash ts \rightsquigarrow v$ for the remaining cases.

1. As $s = t$, we have $\texttt{splitByPrefix}(t, t) = [t, \bot]$. Then, we directly have $t \vee \bot \overset{lem1}{\equiv} t = s$
2. Induction on $s \vdash ts \rightsquigarrow v$:
   - *Case* `MElem`: The syntax s is bound to be $elem_k$ for some kind k.

     We have $\texttt{splitByPrefix}(t, elem_k) = [\bot, elem_k]$, and directly get $\bot \vee elem_k \overset{lem1}{\equiv} elem_k = s$
   - *Case* `MEps`: The syntax s is bound to be ε_v for some value v.

     We have $\texttt{splitByPrefix}(t, \varepsilon_v) = [\bot, \varepsilon_v]$, and directly get $\bot \vee \varepsilon_v \overset{lem1}{\equiv} \varepsilon_v = s$

- *Case* `MDisL` and `MDisR`: The syntax s is bound to be $s_1 \vee s_2$.

  Let $[s_p, s_n] := \texttt{splitByPrefix}(t, s)$, $[s_{p1}, s_{n1}] := \texttt{splitByPrefix}(t, s_1)$ and $[s_{p2}, s_{n2}] := \texttt{splitByPrefix}(t, s_2)$. By definition, $s_p = s_{p1} \vee s_{p2}$ and $s_n = s_{n1} \vee s_{n2}$. By induction hypothesis, $s_{p1} \vee s_{n1} \equiv s_1$ and $s_{p2} \vee s_{n2} \equiv s_2$.

  We can then verify that

  $$s = s_1 \vee s_2 \equiv (s_{p1} \vee s_{n1}) \vee (s_{p2} \vee s_{n2}) \overset{lem1}{\equiv} (s_{p1} \vee s_{p2}) \vee (s_{n1} \vee s_{n2}) = s_p \vee s_n$$

- *Case* `MSeq`: The syntax s is bound to be $s_1 \cdot s_2$.

  Let $[s_p, s_n] := \texttt{splitByPrefix}(t, s)$ and $[s'_p, s'_n] := \texttt{splitByPrefix}(t, s_1)$. By definition, $s_p = s'_p \cdot s_2$ and $s_n = s'_n \cdot s_2$. By induction hypothesis, $s'_p \vee s'_n \equiv s_1$.

  We can then verify that

  $$s = s_1 \cdot s_2 \equiv (s'_p \vee s'_n) \cdot s_2 \overset{lem1}{\equiv} (s'_p \cdot s_2) \vee (s'_n \cdot s_2) = s_p \vee s_n$$

- *Case* `MMap`: The syntax s is bound to be $f \circledcirc s'$.

  Let $[s_p, s_n] := \texttt{splitByPrefix}(t, s)$ and $[s'_p, s'_n] := \texttt{splitByPrefix}(t, s')$. By definition, $s_p = f \circledcirc s'_p$ and $s_n = f \circledcirc s'_n$. By induction hypothesis, $s'_p \vee s'_n \equiv s'$.

  We can then verify that

  $$s = f \circledcirc s' \equiv f \circledcirc (s'_p \vee s'_n) \overset{lem1}{\equiv} (f \circledcirc s'_p) \vee (f \circledcirc s'_n) = s_p \vee s_n$$

- *Case* `MVar`: We have $\texttt{splitByPrefix}(t, var_x) = [\bot, var_x]$, and directly get $\bot \vee var_x \overset{lem1}{\equiv} var_x = s$

□

7 Left recursion elimination

7.1 First recursions

To define left recursion, we will need the concept of first recursions of a syntax.

Definition 14 The set of first recursions of a syntax s is defined by the following inductive predicate

$$\frac{x \in \textsc{firstRec}(e_1)}{x \in \textsc{firstRec}(e_1 \vee e_2)} \qquad \frac{x \in \textsc{firstRec}(e_2)}{x \in \textsc{firstRec}(e_1 \vee e_2)}$$

$$\frac{x \in \textsc{firstRec}(e_1)}{x \in \textsc{firstRec}(e_1 \cdot e_2)} \qquad \frac{\textsc{nullable}(e_1) \quad x \in \textsc{firstRec}(e_2)}{x \in \textsc{firstRec}(e_1 \cdot e_2)}$$

$$\frac{x \in \textsc{firstRec}(e)}{x \in \textsc{firstRec}(f \circledcirc e)} \qquad \frac{}{x \in \textsc{firstRec}(var_x)}$$

Definition 15 A variable var_x is said to be direct left recursive if

$$x \in \textsc{firstRec}(\texttt{getDef}(x))$$

Definition 16 A variable var_x is said to be left recursive if there exists a sequence of variables var_{x_1}, var_{x_2}, …, var_{x_n}, where $var_x = var_{x_1}$, such that

$$x_1 \in \textsc{firstRec}(\texttt{getDef}(x_n)) \qquad x_{i+1} \in \textsc{firstRec}(\texttt{getDef}(x_i)) \quad \forall i \in 1, \ldots, n$$

For a grammar $(\mathcal{N}, \mathcal{T}, R, S)$ where $\mathcal{N} = \{A_1, \ldots, A_n\}$.

```
for (i = 1 **to** n) {
    for (j = 1 **to** i–1) {
        for (r **in** R **matching** $A_i ::= A_j\alpha$) {
            remove r from $\mathcal{N}$
            for (_ **in** R **matching** $A_j ::= \beta$) {
                add $A_i ::= \beta\alpha$ **to** $\mathcal{N}$
            }
        }
    }
    eliminate $A_i$'s direct left recursion(s)
}
```

Fig. 1: Paull's algorithm, adapted from [7, Figure 1].

Note The thesis [1, Page 68] gives another definition of left-recursive syntaxes (which we only found after making our own definition). We kept our own because the original one doesn't easily allow the definition of direct left recursion.

Left recursions are problematic for two reasons:

1. Left recursive syntaxes always contain first/first conflicts (or are unproductive);
2. Left-factoring (as defined before) might loop infinitely on such syntaxes.

As such, left recursion elimination is a useful transformation.

7.2 Paull's algorithm

An algorithm to remove left recursion is Paull's algorithm, which is quite well-known due to its usage in the standard transformation into Greibach Normal Form [5, Section 7.7].

Its pseudo-code is given in fig. 1. The algorithm operates by repeating two steps:

1. Inline some non-terminals;
2. Remove all direct left recursions.

7.3 Adaptation to syntax

7.3.1 Inlining

Inlining non-terminals (also called unfolding, as in [8]) is the grammar equivalent to replacing a variable by its body, which can easily be performed using `substitute`:

$$\texttt{inline}(x, s) := \texttt{substitute}(var_x, \texttt{getDef}(x), s)$$

Theorem 7. Variables inlining is equivalence preserving, i.e. $s \equiv \texttt{inline}(x, s)$

Proof. By definition, $var_x \equiv \texttt{getDef}(x)$. Then, using this in theorem 5 directly yields

$$\texttt{inline}(x, s) \overset{\text{def}}{=} \texttt{substitute}(var_x, \texttt{getDef}(x), s) \overset{\text{th5}}{\equiv} s$$

□

7.3.2 Removing direct left recursion

A procedure for eliminating direct left recursion is given in [5, Exercise 7.1.11c]

$$A ::= A\beta_1 \mid \ldots \mid A\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m \xrightarrow{\texttt{directLeftRecElimination}(A)} \begin{array}{ll} A & ::= \gamma_1 \mid \ldots \mid \gamma_m \\ A & ::= \gamma_1 A' \mid \ldots \mid \gamma_m A' \\ A' & ::= \beta_1 A' \mid \beta_1 \mid \ldots \mid \beta_n A' \mid \beta_n \end{array}$$

where $\forall\gamma_i \neg\exists\delta : \gamma_i = A\delta$ (i.e. A is not a prefix of any γ_i).

This can then be re-arranged into

$$A ::= A\beta_1 \mid \ldots \mid A\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m \quad \xrightarrow{\texttt{directLeftRecElimination}(A)} \quad \begin{array}{ll} A & ::= A_p A_f \\ A_p & ::= \gamma_1 \mid \ldots \mid \gamma_m \\ A_f & ::= \varepsilon \mid \beta_1 A_f \mid \ldots \mid \beta_n A_f \end{array}$$

Which can be adapted for syntaxes as follows:

$$\begin{aligned} &[s_f, s_a] := \texttt{leftFactorSymbolOut}(var_x, s) \\ &\texttt{directLeftRecElimination}(x) := \theta \circledcirc (s_a \cdot \texttt{many}(s_f)) \end{aligned}$$

where $\theta([v, \langle f_1, f_2, \ldots, f_n\rangle]) := f_n(\ldots f_2(f_1(v)))$

8 Implementation

8.1 Transformations

All the transformations discussed before[1] (and some extra ones) were implemented in Scala, on top of the ScaLL1on library[2]. The transformations were implemented without changing anything to the library: the files can simply added by the user without changing the core library.

8.2 Some extra issues

Doing a concrete implementation of the transformations introduces a few extra points which must be discussed:

8.2.1 Syntaxes cannot simply be re-written

When discussing the transformations, we were only interested in “how to perform the transformation”, and not really on “how to use it”.

In particular, one cannot simply apply a transformation to a syntax and hope that the old syntax (which most likely contained conflicts) disappears from all variables or the root syntax (even more since ScaLL1on mostly uses immutable data structures).

This is the reason why the `substitute` transformation is a must-have: it allows, once a syntax is transformed, to replace all occurrences of the original one with the new one. One might have noticed that `substitute` as defined in definition 12 isn't a perfect fit, as it is not applied recursively on the variables it contains. This will be further discussed in section 8.2.2.

This problem reached its peak when implementing Paull's algorithm. The algorithm assumes that rules can easily be removed/added (fig. 1, lines 4,6), which is, as mentioned earlier, not the case with immutable syntaxes. This required many tweaks to the algorithm, so much in fact, that the implemented algorithm doesn't even remotely look like Paull's anymore, and is probably way less efficient.

8.2.2 Transformation mode

When introducing the different transformations in this report, we always introduced the version which was most useful at that point (`leftFactorize` maximized conflict elimination, `substitute` was suited to eliminate left recursions). In the actual implementation, some transformations take an extra argument (e.g. `leftFactorize`, see fig. 2) which specifies whether recursive blocks (which are ScaLL1on's equivalent to variables) should be further explored or seen as leafs of the syntax

In fact, $\texttt{leftFactorSymbolOut}(t, s)$ was not implemented as a transformation on its own. Since it is only `leftFactorOut` without recursive application, it simply became $\texttt{leftFactorOut}(t, s, \texttt{false})$.

8.3 A little extra: Syntax Zipper

Huet's Zipper [6] is the data structure on which ScaLL1on relies to parse efficiently, but its use is purely internal to the parser; no method to navigate or modify a syntax using zippers is provided to the user.

[1] This sentence is a bit misleading: the transformations were in fact first implemented in Scala, and then formalized in the present report.

[2] `https://github.com/Ef55/scallion/tree/factorization`

```
def leftFactorize[A, L](leftFactor: Syntax[L], s: Syntax[A], enterRecs: Boolean = true): Syntax[A]
```

Fig. 2: Signature of `leftFactorize`.

Since transformations need to modify the syntax tree quite often, which isn't easy nor efficient with immutable trees, having a Zipper to perform such operations more efficiently seemed like a good idea. This motivated the implementation of a Zipper on syntaxes[3]. Sadly, this idea came quite late during this project, so the zipper is not used as much as it could in the different transformations.

8.4 Actual results

8.4.1 Applying the transformations automatically

Now that all transformations were introduced, we can now quickly discuss how to solve conflicts automatically. Note that the following procedures do not really have any theoretical ground, they just happen to solve some conflicts. Also, it is perfectly possible that these procedures fail at solving some conflicts. In fact, we sadly found some cases where the procedures (as implemented in practice) crashed or looped infinitely, which is quite the let down.

Nullable/Nullable conflicts These conflicts are the easiest: they cannot be fixed! In fact, their presence implies that the underlying grammar is ambiguous, so fixing them automatically is most likely impossible.

First/First conflicts The first step is to eliminate left recursions (if there are any). As mentioned before, left recursions generate first/first conflicts, and left factorization might loop infinitely on left recursive syntaxes, so eliminating these recursions is the first logical step.

Then, left factorization is applied to all remainig first/first conflicts. Note that left-factoring might generate left recursions, and as such, left recursion elimination must be applied after each left factorization.

First/Follow conflicts Those are the most difficult conflicts to solve. First, note that such conflicts are always caused by two things:

1. There must be a disjunction which "bootstraped" the should-not-follow set (SDisFN or SDisNF in [1, Figure 4.1]). We will call it the source of the conflict;
2. There must be a sequence where the left-child's should-not-follow set and the right-child's first set are not disjoined (this is where ScaLL1on detects the conflict). We will call it the root of the conflict[4].

The procedure we came up with is the following.

- Right factor out the source from the left-child. Note that this is always possible, as, what comes after the source must be nullable (otherwise, the should-not-follow set of the source would not have influenced the root left-child's should-not-follow set).
- Create a new sequence from the source and the root's right-child. This new syntax has a special kind of first/follow conflict(s) which can be solved by left factorization.
- Sequence the root's left-child after factorization with the syntax from previous step, and apply a transformation ($[f, [l, r]] \to [f(l), r]$) to make it equivalent to the original one.

This procedure seem bewildering (the actual implementation might make it easier to understand: fig. 3), but is in fact well-typed and we verified on some examples (like the one in section 8.4.2) that conflicts were indeed solved.

[3] In fact, the implemented Zipper is generic and could be used on any binary-tree-like structure.

[4] Those names (source, root) are the ones used by ScaLL1on.

```
def solveFollowConflict[A](syntax: Syntax[A], conflict: FollowConflict): Syntax[A] = {
    val tokens = conflict.ambiguities
    (conflict.source, conflict.root) match {
        case ( src: Syntax[tS], root@Sequence(l: Syntax[tL], r: Syntax[tR]) ) => {
            val (factorized: Syntax[tS => tL], alternative: Syntax[tS]) = rightForceOut(src, l)
            val solved: Syntax[tS ~ tR] = tokens.foldLeft(src ~ r)( (s, t) => leftFactorizeKind(t, s))
            val newSyntax: Syntax[tL ~ tR] = (alternative ~ r) |
                (factorized ~ solved).map{ case f ~ (l ~ r) => scallion.~(f(l), r) }
            eliminate(syntax, root, newSyntax)
        }
    }
}
```

The `rightForceOut` function used is special variant of right factor out, which we did not discuss in this report.

Fig. 3: Actual implementation of the procedure solving First/Follow conflicts.

```
// The kinds
val identifier = accept(word)
val tagPrefix = elem("<")
val tagSuffix = elem(">")
val slash = elem("/")

// The different tags
lazy val tag: Syntax[Tag] = recursive{ tagPair | emptyTag }
val openingTag = (tagPrefix ⇝ identifier ⇜ tagSuffix).map{case id => OpeningTag(id)}
val closingTag = (tagPrefix ⇝ slash ⇝ identifier ⇜ tagSuffix).map(id => ClosingTag(id))
val emptyTag = (tagPrefix ⇝ identifier ⇜ slash ⇜ tagSuffix).map{case id => EmptyTag(id)}.up[Tag]
val tagPair = (openingTag ~ opt(tag) ~ closingTag).map{ case op ~ ct ~ cl => TagPair(op, ct, cl)}.up[Tag]

val grammar = tag
```

Fig. 4: Syntax for tags, in ScaLL1on's notation.

8.4.2 An example

A little example was built to showcase the conflict solving procedures. It was inspired by markup languages, and more precisely HTML and XML, which use tags to structure their data. The syntax to define these tags is quite easy to write, but is clearly not LL1, and writing it in an LL1 way is quite unnatural. These facts make it a perfect candidate to be written in a non-LL1 fashion, and then transformed before being made into a parser.

The toy syntax (fig. 4) is quite simple. It allows to define tag pairs (e.g. `<pair> </pair>`), empty tags (e.g. `<empty/>`) and to nest them (e.g. `<outer> <inner/> </outer>`).

This example was implemented in Scala, and was successfully transformed into an LL1 parser. It can be found on our repository[5].

[5] https://github.com/Ef55/scallion/blob/factorization/example/tags/FirstFollow.scala

References

[1] Romain Edelmann. “Efficient Parsing with Derivatives and Zippers”. PhD thesis. Lausanne, CH: École polytechnique fédérale de Lausanne, 2021.

[2] Romain Edelmann, Jad Hamza, and Viktor Kunčak. “Zippy LL(1) Parsing with Derivatives”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 1036–1051. ISBN: 9781450376136. DOI: 10.1145/3385412.3385992. URL: https://doi.org/10.1145/3385412.3385992.

[3] J. M. Foster. “A syntax improving program”. In: *The Computer Journal* 11.1 (Jan. 1968), pp. 31–34. ISSN: 0010-4620. DOI: 10.1093/comjnl/11.1.31. eprint: https://academic.oup.com/comjnl/article-pdf/11/1/31/1172766/110031.pdf. URL: https://doi.org/10.1093/comjnl/11.1.31.

[4] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques. A Practical Guide*. 2nd ed. New York: Springer-Verlag, 2008. ISBN: 978-1-4419-1901-4.

[5] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Intoduction to Automata Theory, Languages, and Computation*. 3rd ed. Pearson, 2006. ISBN: 978-0321455369.

[6] Gérard Huet. “The zipper”. In: *Journal of functional programming* 7.5 (1997), pp. 549–554.

[7] Robert C Moore. “Removing left recursion from context-free grammars”. In: *1st Meeting of the North American Chapter of the Association for Computational Linguistics*. 2000.

[8] Peter Pepper. *LR Parsing = Grammar Transformation + LL Parsing — Making LR Parsing More Understandable And More Efficient*. 1999.