Closure Properties of General Grammars — Formally Verified

Martin Dvorak
Institute of Science and Technology Austria, Austria

Jasmin Blanchette
Ludwig-Maximilians-Universität München, Germany

Abstract

We formalized general (i.e., type-0) grammars using the Lean 3 proof assistant. We defined basic notions of rewrite rules and of words derived by a grammar, and used grammars to show closure of the class of type-0 languages under four operations: union, reversal, concatenation, and the Kleene star. The literature mostly focuses on Turing machine arguments, which are possibly more difficult to formalize. For the Kleene star, we could not follow the literature and came up with our own grammar-based construction.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Lean, type-0 grammars, recursively enumerable languages, Kleene star

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.11

Supplementary Material https://github.com/madvorak/grammars/tree/publish

Funding *Jasmin Blanchette*: This research has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

Acknowledgements We thank Vladimir Kolmogorov for making this collaboration possible. We thank Václav Končický for discussing ideas about the Kleene star construction. We thank Patrick Johnson, Floris van Doorn, and Damiano Testa for their small yet very valuable contributions to our code. We thank Eric Wieser for simplifying one of our proofs. We thank Mark Summerfield for suggesting textual improvements. We thank the anonymous reviewers for very helpful comments. Finally, we thank the Lean community for helping us with various technical issues and answering many questions.

1 Introduction

The notion of formal languages lies at the heart of computer science. There are several formalisms that recognize formal languages, including Turing machines and formal grammars. In particular, both Turing machines and general grammars (also called type-0 grammars or unrestricted grammars) characterize the same class of languages, namely, the recursively enumerable or type-0 languages.

There has been work on formalizing Turing machines in proof assistants [7, 2, 26, 6, 15, 3]. General grammars are an interesting alternative because they are easier to define than Turing machines, and some proofs about general grammars are much easier than the proofs of similar properties of Turing machines.

We therefore chose general grammars as the basis for our Lean 3 [9] library of results about recursively enumerable or type-0 languages. The definition of grammars consists of several layers of concepts (Section 2):

- the type of symbols is the disjoint union of terminals and nonterminals;
- rewrite rules are pairs of the form $u \to v$, where u and v are strings over symbols and u contains at least one nonterminal [1];

© Martin Dvorak and Jasmin Blanchette;
licensed under Creative Commons License CC-BY 4.0

- a grammar is a tuple consisting of a type of terminals, a type of nonterminals, an initial symbol S, and a set of rewrite rules;
- application of a rewrite rule $u \to v$ to a string $\alpha u \beta$ is written $\alpha u \beta \Rightarrow \alpha v \beta$;
- the derivation relation $\Rightarrow^*$ is the reflexive transitive closure of the $\Rightarrow$ relation;
- a grammar derives a word w if $S \Rightarrow^* w$;
- the language generated by a grammar is the set of words derived by it;
- a language is type 0 if there exists a grammar that generates it.

We formalized four closure properties of type-0 languages.

The first such property we present is closure of type-0 languages under union (Section 3). We followed the standard construction for context-free grammars, which incidentally works for general grammars as well.

The second closure property we formalized is closure under reversal (Section 4). This was straightforward.

The third closure property we formalized is closure under concatenation (Section 5). The main difficulty was to avoid matching strings on the boundary of the concatenation. This issue does not arise with context-free grammars because only single symbols are matched and these are tidily located on either side of the boundary.

The fourth and last closure property we formalized is closure under the Kleene star (Section 6). This was the most difficult part of our work. Because the literature mostly focuses on Turing machine arguments, we needed to invent our own construction. We first developed a detailed proof sketch and then formalized it. The sketch is included in this paper.

One closure property we did not formalize is closure under intersection. The reason is that we are not aware of any elegant construction based on grammars only. Recall that type-0 languages are not closed under complement, as witnessed by the halting problem [18].

Our development is freely available online.[1] It consists of about 12 500 lines of spaciously formatted Lean code. It uses the Lean 3 mathematical library `mathlib` [22].[2] We also benefited from the metaprogramming framework [10], which allowed us to easily develop small-scale automation that helped make some proofs less verbose.

Although Lean is based on dependent type theory [21], our code uses only nondependent types for data. We still found dependent type theory useful for bound-checked indexing of lists using the function `list.nth_le` (which takes a list, an index, and a proof that the index is within bounds as arguments). We did not attempt to make our development constructive.

2 Definitions

2.1 Grammars

As outlined in the introduction, the definition of grammars consists of several layers of declarations.

Symbols are essentially defined as a sum type of terminals `T` and nonterminals `N`. However, we want to refer to terminals and nonterminals by name (using `symbol.terminal` and `symbol.nonterminal` instead of `sum.inl` and `sum.inr`), so we define symbols as an inductive type:

[1] https://github.com/madvorak/grammars/tree/publish
[2] https://github.com/leanprover-community/mathlib/tree/7ed4f2cec2

```
inductive symbol (T : Type) (N : Type)
| terminal    : T → symbol
| nonterminal : N → symbol
```

We do not require `T` and `N` to be finite. As a result, we do not need to copy the typeclass instances `[fintype T]` and `[fintype N]` alongside our type parameters (which would appear in almost every lemma statement). Instead, later we work in terms of a list of rewrite rules, which is finite by definition and from which we could infer that only a finite set of terminals and a finite set of nonterminals can occur.

The left-hand side u of a rewrite rule $u \to v$ consists of three parts (an arbitrary string α, a nonterminal A, and another arbitrary string β, such that $u = \alpha A \beta$):

```
structure grule (T : Type) (N : Type) :=
(input_L : list (symbol T N))
(input_N : N)
(input_R : list (symbol T N))
(output_string : list (symbol T N))
```

An advantage of this representation is that we do not need to carry the proposition "the left-hand side contains a nonterminal" around. A disadvantage is that we subsequently need to concatenate more terms.

A definition of a general grammar follows. Notice that only the type argument `T` is part of its type:

```
structure grammar (T : Type) :=
(nt : Type)
(initial : nt)
(rules : list (grule T nt))
```

Later we can use the dot notation to access individual fields. For example, if `g` is a term of the type `grammar T`, we can write `g.nt` to access the type of its nonterminals. By writing `(g.rules.nth_le 0 _).output_string` we obtain the right-hand side of the first rewrite rule in `g`. The underscore, when not inferred automatically, must be replaced by a term of the type `0 < g.rules.length`, which is a proof that the list `g.rules` is not empty.

The next line adds an implicit type argument `T` to all declarations that come after:

```
variables {T : Type}
```

The following definition captures the application $\Rightarrow$ of a rewrite rule:

```
def grammar_transforms (g : grammar T)
    (w₁ w₂ : list (symbol T g.nt)) :
  Prop :=
∃ r : grule T g.nt,
  r ∈ g.rules   ∧
  ∃ u v : list (symbol T g.nt),
    w₁ = u ++ r.input_L ++ [symbol.nonterminal r.input_N]
           ++ r.input_R ++ v    ∧
    w₂ = u ++ r.output_string ++ v
```

The operator `++` concatenates two lists. We can view `grammar_transforms` as a function that takes a grammar `g` over the terminal type `T` and outputs a binary relation over strings of the type that `g` works internally with.

The part `r.input_L ++ [symbol.nonterminal r.input_N] ++ r.input_R` represents the left-hand side of the rewrite rule `r`. Note that the terms `r.input_L` and `r.input_N` cannot be concatenated directly, since they have different types. The term `r.input_N` must first be wrapped in `symbol.nonterminal` to go from the type `g.nt` to the type `symbol T g.nt` and then surrounded by `[]` to become a (singleton) list.

The derivation relation $\Rightarrow^*$ is defined from `grammar_transforms` using the reflexive transitive closure:

```
def grammar_derives (g : grammar T) :
  list (symbol T g.nt) → list (symbol T g.nt) → Prop :=
relation.refl_trans_gen (grammar_transforms g)
```

Consequently, proofs about derivations will use structural induction.

The predicate “to be a word generated by the grammar `g`” is defined as the special case of the relation `grammar_derives g` where the left-hand side is fixed to be the singleton list made of the initial symbol of `g` and the right-hand side is required to consist of terminal symbols only:

```
def grammar_generates (g : grammar T) (w : list T) : Prop :=
grammar_derives g [symbol.nonterminal g.initial]
  (list.map symbol.terminal w)
```

2.2 Languages

In our entire project, we work with the following definition of languages provided by `mathlib` in the `computability` package:

```
def language (α : Type*) := set (list α)
```

The type argument α is instantiated by our terminal type `T` in all places where we work with languages. We do not mind restricting `T` to be `Type` since we are not interested in languages over types from `Type 1` and higher universes.

The language of the grammar `g` is defined as the set of all `w` that satisfy the predicate `grammar_generates g w` declared above:

```
def grammar_language (g : grammar T) : language T :=
set_of (grammar_generates g)
```

Note that the type parameter `T` is preserved, but `g.nt` does not matter in the description of what words are generated. It corresponds to our intuition that the type of terminals is a part of the interface, but the type of nonterminals is an implementation matter.

This is the first time that our custom types meet the standard `mathlib` type `language`, which is already connected to many useful types, such as the type of regular expressions.

Finally, we define the class of type-0 languages:

```
def is_T0 (L : language T) : Prop :=
∃ g : grammar T, grammar_language g = L
```

All top-level theorems about type-0 languages are expressed in terms of the `is_T0` predicate.

Note that the type system distinguishes between a list of terminals and a list of symbols that happen to be terminals. Languages are defined as sets of the former, whereas derivations in the grammar work with the latter.

In a similar way, we define `CF_grammar`, `CF_transforms`, `CF_derives`, `CF_generates`, `CF_language`, and the `is_CF` predicate for the formal definition of context-free languages.

The theorem `CF_subclass_T0` connects the context-free languages to the type-0 languages. Type-0 languages remain the main focus of our work.

2.3 Operations

The operations under which we prove closure are defined below.

Union is defined in `mathlib` as follows:

```
protected def set.union (s₁ s₂ : set α) : set α :=
{a | a ∈ s₁ ∨ a ∈ s₂}
```

The following declaration in `mathlib` states that the union of languages is denoted by writing the `+` operator between two terms of the `language` type:

```
instance : language.has_add (language α) := ⟨set.union⟩
```

We define the reversal of a language as follows:

```
def reverse_lang (L : language T) : language T :=
λ w : list T, w.reverse ∈ L
```

We do not declare any syntactic sugar for reversal.

Concatenation is defined using the following general `mathlib` definition:

```
def set.image2 (f : α → β → γ) (s : set α) (t : set β) : set γ :=
{c | ∃ a b, a ∈ s ∧ b ∈ t ∧ f a b = c}
```

The next `mathlib` declaration states that concatenation of languages is denoted by writing the `*` operator between two terms of the `language` type:

```
instance : language.has_mul (language α) := ⟨set.image2 (++)⟩
```

The Kleene star of a language is defined in `mathlib` as follows:

```
def language.star (l : language α) : language α :=
{x | ∃ S : list (list α), x = S.join ∧ ∀ y ∈ S, y ∈ l}
```

We do not declare any syntactic sugar for the Kleene star.

3 Closure under Union

In this section, we prove the following theorem:

```
theorem T0_of_T0_u_T0 (L₁ : language T) (L₂ : language T) :
  is_T0 L₁ ∧ is_T0 L₂ → is_T0 (L₁ + L₂)
```

The proof of closure of type-0 languages under union consists of three main ingredients:
(1) a construction of a new grammar `g` from any two given grammars g_1 and g_2;
(2) a proof that any word generated by g_1 or g_2 can also be generated by `g`;
(3) a proof that any word generated by `g` can be equally generated by g_1 or g_2.
Proofs of the other closure properties are organized analogously. We describe the proof of closure under union in more detail; it allows us to outline the main ideas of proving closure properties formally in a simple setting. Since (3) is usually much more difficult than (2), we refer to (2) as the "easy direction" and to (3) as the "hard direction".

The proof of the closure of type-0 languages under union follows the standard construction, which usually states only (1) explicitly, and leaves (2) and (3) to the reader. We begin (1) by defining a new type of nonterminals. The nonterminals of `g` consist of

- the nonterminals of $\texttt{g}_1$ including a mark indicating their origin;
- the nonterminals of $\texttt{g}_2$ including a mark indicating their origin;
- one new distinguished nonterminal.

The Lean type `option (g₁.nt ⊕ g₂.nt)` encodes this disjoint union. If `m` is a nonterminal of type $\texttt{g}_1$`.nt`, its corresponding nonterminal of type `g.nt` is `some (sum.inl m)`. If `n` is a nonterminal of type $\texttt{g}_2$`.nt`, its corresponding nonterminal of type `g.nt` is `some (sum.inr n)`. The new distinguished nonterminal is called `none` and becomes the initial symbol of `g`. The rewrite rules of `g` consist of

- the rewrite rules of $\texttt{g}_1$ with all nonterminals mapped to the larger nonterminal type;
- the rewrite rules of $\texttt{g}_2$ with all nonterminals mapped to the larger nonterminal type;
- two additional rules that rewrite the initial symbol of `g` to the initial symbol of $\texttt{g}_1$ or $\texttt{g}_2$.

To reduce the amount of repeated code in the proof, we developed lemmas that allow us to "lift" a grammar with a certain type of nonterminals to a grammar with a larger type of nonterminals while preserving what the grammar derives. Under certain conditions, we can also "sink" the larger grammar to the original grammar and preserve its derivations.

These lemmas operate on a structure called `lifted_grammar` that consists of the following fields:

- a smaller grammar $\texttt{g}_0$ that represents either $\texttt{g}_1$ or $\texttt{g}_2$ in case of the proof for union;
- a larger grammar `g` with the same type of terminals;
- a function `lift_nt` from $\texttt{g}_0$`.nt` to `g.nt`;
- a partial function `sink_nt` from `g.nt` to $\texttt{g}_0$`.nt`;
- a proposition `lift_inj` that guarantees that `lift_nt` is injective;
- a proposition `sink_inj` that guarantees that `sink_nt` is injective on inputs for which $\texttt{g}_0$ has a corresponding nonterminal;
- a proposition `lift_nt_sink` that guarantees that `sink_nt` is essentially an inverse of `lift_nt`;
- a proposition `corresponding_rules` that guarantees that `g` has a rewrite rule for each rewrite rule $\texttt{g}_0$ has (with different type but the same behavior);
- a proposition `preimage_of_rules` that guarantees that $\texttt{g}_0$ has a rewrite rule for each rewrite rule of `g` whose nonterminal has a preimage on the $\texttt{g}_0$ side.

Thanks to this structure, we can abstract from the specifics of how the larger grammar is constructed in concrete proofs and care only about the properties that are required to follow analogous derivations.

To illustrate how we work with this abstraction, we review the proof of the following lemma:

```
private lemma lift_tran {lg : lifted_grammar T}
    {w₁ w₂ : list (symbol T lg.g₀.nt)}
    (hyp : grammar_transforms lg.g₀ w₁ w₂) :
  grammar_transforms lg.g
    (lift_string lg.lift_nt w₁)
    (lift_string lg.lift_nt w₂)
```

We need to show that if $\texttt{g}_0$ has a rewrite rule that transforms $\texttt{w}_1$ to $\texttt{w}_2$, then `g` has a rewrite rule that transforms `lift_string lg.lift_nt w₁` to `lift_string lg.lift_nt w₂`.

We start by deconstructing `hyp` according to the `grammar_transforms` definition. To go from g_0 to `g`, we first "lift" the rewrite rule (i.e., translate its nonterminals in all fields) that g_0 used. We call `corresponding_rules` to show that `g` has such a rule. Then we use the function `lift_string` to lift `u` and `v`, which are the parts of the string w_1 that were not matched by the rule. We are then left with the proof obligation

```
lift_string lg.lift_nt w₁ =
lift_string lg.lift_nt u ++ (lift_rule lg.lift_nt r).input_L
  ++ [symbol.nonterminal (lift_rule lg.lift_nt r).input_N]
  ++ (lift_rule lg.lift_nt r).input_R ++ lift_string lg.lift_nt v
```

where

```
w₁ =
u ++ r.input_L ++ [symbol.nonterminal r.input_N] ++ r.input_R ++ v
```

and with the proof obligation

```
lift_string lg.lift_nt w₂ =
lift_string lg.lift_nt u ++ (lift_rule lg.lift_nt r).output_string
  ++ lift_string lg.lift_nt v
```

where

```
w₂ = u ++ r.output_string ++ v
```

These two obligations originate from the two identities in the definition `grammar_transforms` from Section 2. Essentially, we discharge them using the distributivity of `lift_string` over the `++` operation.

The abstraction provided by `lifted_grammar` takes care of the vast majority of our proof of the closure of type-0 languages under union. It remains to separately analyze what was the first step of the derivation that `g` did in the hard direction. We need to exclude all rules that are inherited from g_1 and g_2 and perform a case analysis on the two special rules.

The two additional rules of `g` are context-free. Therefore, if g_1 and g_2 have context-free rules only, then all rules of `g` are context-free as well. As a consequence, our result about type-0 languages can easily be reused to prove the closure of context-free languages under union:

```
theorem CF_of_CF_u_CF (L₁ : language T) (L₂ : language T) :
  is_CF L₁ ∧ is_CF L₂  → is_CF (L₁ + L₂)
```

Not much Lean code needs to be duplicated to obtain the result about context-free grammars. We need to write the construction of `g` again and the main result again. The remaining parts are achieved by reusing lemmas from the proof for general grammars. The main overhead is proving

```
private lemma union_grammar_eq_union_CF_grammar
    {g₁ g₂ : CF_grammar T} :
  union_grammar (grammar_of_cfg g₁) (grammar_of_cfg g₂) =
  grammar_of_cfg (union_CF_grammar g₁ g₂)
```

Even though the statement might look complicated, the proof has only five lines, making it one of the shortest tactic-based proofs in our project.

4 Closure under Reversal

In this section, we prove the following theorem:

```
theorem T0_of_reverse_T0 (L : language T) :
  is_T0 L → is_T0 (reverse_lang L)
```

The proof is very easy. Simply speaking, everything gets reversed. We start with the rewrite rules:

```
private def reversal_grule {N : Type} (r : grule T N) : grule T N :=
grule.mk r.input_R.reverse r.input_N r.input_L.reverse
  r.output_string.reverse
```

The constructor `grule.mk` takes arguments in the same order as they are written in the definition:

- its `input_L` is instantiated by `r.input_R.reverse`;
- its `input_N` is instantiated by `r.input_N`;
- its `input_R` is instantiated by `r.input_L.reverse`;
- its `output_string` is instantiated by `r.output_string.reverse`.

The new grammar is constructed as follows:

```
private def reversal_grammar (g : grammar T) : grammar T :=
grammar.mk g.nt g.initial (list.map reversal_grule g.rules)
```

The rest is essentially a repeated application of lemma `list.reverse_append_append`, which is just a repeated application of lemma `list.reverse_append`, which states that reversing two concatenated lists is equivalent to reversing both parts and concatenating them in the opposite order, and lemma `list.reverse_reverse`, which states that `list.reverse` is a dual operation.

5 Closure under Concatenation

In this section, we prove the following theorem:

```
theorem T0_of_T0_c_T0 (L₁ : language T) (L₂ : language T) :
  is_T0 L₁ ∧ is_T0 L₂ → is_T0 (L₁ * L₂)
```

Because the proof is highly technical, we only outline the main idea here.

We first review the classical construction for context-free grammars. Let $L_1 \subseteq T^*$ be a language generated by a grammar $G_1 = (N_1, T, P_1, S_1)$. Let $L_2 \subseteq T^*$ be a language generated by a grammar $G_2 = (N_2, T, P_2, S_2)$. Without loss of generality, the sets N_1 and N_2 are disjoint. We create a new initial symbol S that appears only in the rule $S \to S_1 S_2$. The new grammar is $(N_1 \cup N_2 \cup \{S\},\ T,\ P_1 \cup P_2 \cup \{S \to S_1 S_2\},\ S)$. This construction works for context-free grammars because S_1 gives rise to a word from L_1 and, independently, S_2 gives rise to a word from L_2.

For general grammars, the construction above does not work, as the following counter-example over $T = \{a, b\}$ illustrates. Let the rule sets be $P_1 = \{S_1 \to S_1 a,\ S_1 \to \epsilon\}$ and $P_2 = \{S_2 \to S_2 a,\ S_2 \to \epsilon,\ a S_2 \to b\}$. We obtain $L_1 = L_2 = \{a^n \mid n \in \mathbb{N}_0\}$ and so $L_1 L_2$ is $\{a^n \mid n \in \mathbb{N}_0\}$ as well. We can now derive $S \Rightarrow S_1 S_2 \Rightarrow S_1 a S_2 \Rightarrow S_1 b \Rightarrow b \notin L_1 L_2$ and obtain a contradiction.

We need to avoid matching strings that span across the boundary of the concatenation. Since the nonterminal sets are disjoint, the issue arises only with terminals in the left-hand side of rules, which are not present in context-free grammars. We provide a solution below.

Let $\texttt{g}_1$ and $\texttt{g}_2$ generate $\texttt{L}_1$ and $\texttt{L}_2$ respectively. The nonterminals of our new grammar `g` consist of

- the nonterminals of $\texttt{g}_1$ including a mark indicating their origin;
- the nonterminals of $\texttt{g}_2$ including a mark indicating their origin;
- a proxy nonterminal for every terminal from `T` marked for use by $\texttt{g}_1$ only;
- a proxy nonterminal for every terminal from `T` marked for use by $\texttt{g}_2$ only;
- one new distinguished nonterminal.

The new nonterminal type is encoded by the Lean type $\texttt{option (g}_1\texttt{.nt} \oplus \texttt{g}_2\texttt{.nt)} \oplus \texttt{(T} \oplus \texttt{T)}$. The new distinguished nonterminal becomes the initial symbol of `g`.

In this way, we ensure that the nonterminals used by `g` to simulate $\texttt{g}_1$ are disjoint from the nonterminals used by `g` to simulate $\texttt{g}_2$. There are still real terminals used by both grammars, but `g` never has these terminals on the left-hand side of a rule, since the rewrite rules of `g` consist of

- the rewrite rules of $\texttt{g}_1$ with all nonterminals mapped to the new nonterminal type and all terminals replaced by proxy nonterminals of the first kind;
- the rewrite rules of $\texttt{g}_2$ with all nonterminals mapped to the new nonterminal type and all terminals replaced by proxy nonterminals of the second kind;
- for every terminal from `T`, a rule that rewrites the proxy nonterminal of the first kind to the corresponding terminal and a rule that rewrites the proxy nonterminal of the second kind to the corresponding terminal;
- a special rule that rewrites `g.initial` to a two-symbol string $\texttt{[g}_1\texttt{.initial, g}_2\texttt{.initial]}$ wrapped to use the new nonterminal type.

Using this construction, we ensure that all rules of `g` avoid matching strings on the boundary of the concatenation.

Proving that `g` generates a superset of $\texttt{L}_1 \texttt{ * L}_2$ is easy because we can apply the rewrite rules in the following order, regardless of the languages:

(1) use the special rule to obtain $\texttt{[g}_1\texttt{.initial, g}_2\texttt{.initial]}$ with the necessary wrapping;
(2) generate the string of proxy nonterminals corresponding to the word from $\texttt{L}_1$ while $\texttt{g}_2\texttt{.initial}$ remains unchanged;
(3) replace all proxy nonterminals of the first kind by the corresponding terminals, which results in deriving a word from $\texttt{L}_1$ followed by $\texttt{g}_2\texttt{.initial}$ as the last symbol;
(4) generate the string of proxy nonterminals corresponding to the word from $\texttt{L}_2$ while the first part of the string remains unchanged;
(5) replace all proxy nonterminals of the second kind by the corresponding terminals, which results in deriving a word from $\texttt{L}_2$ that follows the word from $\texttt{L}_1$ obtained before.

Step (1) is trivial. Steps (2) and (4) are done by following the derivations by $\texttt{g}_1$ and $\texttt{g}_2$, respectively. Steps (3) and (5) are straightforward proofs by induction.

Proving that `g` generates a subset of $\texttt{L}_1 \texttt{ * L}_2$ is much harder because we do not know in which order the rules of `g` are applied. We had to come up with an invariant that relates intermediate strings derived by `g` to strings that can be derived by $\texttt{g}_1$ and $\texttt{g}_2$ from their respective initial symbols.

Very roughly speaking, we prove that there are strings `x` and `y` for every string `w` that `g` can derive, such that the grammar $\texttt{g}_1$ can derive `x`, the grammar $\texttt{g}_2$ can derive `y`, and `x ++ y` corresponds to `w`. As usual, we employ structural induction. Looking at the last rule `g` used, we update `x` or `y` or neither. In particular, we want to point out the following declarations in the formalization:

- function `nst` provides the new symbol type which `g` operates with;
- functions `wrap_symbol`$_1$ and `wrap_symbol`$_2$ convert symbols for use by `g`;
- relation `corresponding_strings` built on top of relation `corresponding_symbols` is used to define how the strings `x` and `y` are precisely related to `w` after each step by `g`;
- lemma `induction_step_for_lifted_rule_from_g`$_1$ characterizes the `x` update;
- lemma `induction_step_for_lifted_rule_from_g`$_2$ characterizes the `y` update;
- lemma `big_induction` states the invariant for proving the hard direction;
- lemma `in_concatenated_of_in_big` puts the proof of the hard direction together.

Note that the added rules have only one symbol on the left-hand side. Therefore, if the two original grammars are context-free, our constructed grammar is also context-free. We thereby obtain, as a bonus, a proof that context-free languages are closed under concatenation. It is implemented in a similar fashion to the proof that context-free languages are closed under union.

6 Closure under Kleene Star

In this section, we prove the following theorem:

```
theorem T0_of_star_T0 (L : language T) :
  is_T0 L → is_T0 L.star
```

This is usually demonstrated by a hand-waving argument about a two-tape nondeterministic Turing machine. The language to be iterated is given by a single-tape (nondeterministic) Turing machine. The new machine scans the input on the first tape, copying it onto the second tape as it progresses, and nondeterministically chooses where the first word ends. Next, the original machine is simulated on the second tape. If the simulated machine accepts the word on the second tape, the process is repeated with the current position of the first head instead of returning to the beginning of the input. Finally, when the first head reaches the end of the input, the second tape contains a suffix of the first tape. The original machine is simulated once more on the second tape. If it accepts, the new machine accepts.

Unfortunately, we did not find any proof based on grammars in the literature. Therefore, we had to invent our own construction. In Section 6.1, we present the construction and the idea underlying its correctness using traditional mathematical notation. In Section 6.2, we comment on its formalization.

6.1 Proof Sketch

Let $L \subseteq T^*$ be a language generated by the grammar $G = (N, T, P, S)$. We construct a grammar $G_* = (N_*, T, P_*, Z)$ to generate the language L^*. The new nonterminal set

$$N_* = N \cup \{Z, \#, R\}$$

expands N with three additional nonterminals: a new starting symbol (Z), a delimiter ($\#$), and a marker for final rewriting (R). The new set of rules is

$$P_* = P \cup \{Z \to ZS\#,\ Z \to R\#,\ R\# \to R,\ R\# \to \epsilon\} \cup \{Rt \to tR \mid t \in T\}$$

Intuitively, $\#$ builds compartments that isolate the words from the language L, and then R acts as a cleaner that traverses the string from beginning to end and removes the compartment delimiters $\#$, thereby ensuring that only terminals are present to the left of R.

To see how G_* works, consider the following grammar over $T = \{a, b\}$. Let $N = \{S\}$ and $P = \{S \to aSb,\ S \to \epsilon\}$. The set of rules becomes

$$P_* = \{S \to aSb,\ S \to \epsilon,\ Z \to ZS\#,\ Z \to R\#,\ R\# \to R,\ R\# \to \epsilon,\ Ra \to aR,\ Rb \to bR\}$$

The following is an example of G_* derivation:

$$\begin{aligned}
&Z \Rightarrow ZS\# \Rightarrow ZS\#S\# \Rightarrow ZaSb\#S\# \Rightarrow ZaaSbb\#S\# \Rightarrow ZS\#aaSbb\#S\# \Rightarrow\\
&ZaSb\#aaSbb\#S\# \Rightarrow ZaSb\#aaaSbbb\#S\# \Rightarrow ZaSb\#aaabbb\#S\# \Rightarrow\\
&R\#aSb\#aaabbb\#S\# \Rightarrow RaSb\#aaabbb\#S\# \Rightarrow aRSb\#aaabbb\#S\# \Rightarrow\\
&aRb\#aaabbb\#S\# \Rightarrow abR\#aaabbb\#S\# \Rightarrow abRaaabbb\#S\# \Rightarrow\\
&abaRaabbb\#S\# \Rightarrow abaaRabbb\#S\# \Rightarrow abaaaRbbb\#S\# \Rightarrow\\
&abaaabRbb\#S\# \Rightarrow abaaabRbb\#aSb\# \Rightarrow abaaabbRb\#aSb\# \Rightarrow\\
&abaaabbRb\#ab\# \Rightarrow abaaabbbR\#ab\# \Rightarrow abaaabbbRab\# \Rightarrow\\
&abaaabbbaRb\# \Rightarrow abaaabbbabR\# \Rightarrow abaaabbbab
\end{aligned}$$

▶ **Lemma 1.** *Let $w_1, w_2, \dots, w_n \in L$. Then G_* can derive $Zw_1\#w_2\#\dots w_n\#$.*

Proof. By induction on n. The base case $Z \Rightarrow^* Z$ is trivial.

Now assume $Z \Rightarrow^* Zw_1\#w_2\#\dots w_n\#$ and $S \Rightarrow^* w_{n+1}$. We start with the rule $Z \to ZS\#$. We observe $ZS\# \Rightarrow^* Zw_1\#w_2\#\dots w_n\#S\# \Rightarrow^* Zw_1\#w_2\#\dots w_n\#w_{n+1}\#$. By transitivity, we obtain $Z \Rightarrow^* Zw_1\#w_2\#\dots w_n\#w_{n+1}\#$. ◀

From now on, let $[m]$ denote the set of m natural numbers $\{1, 2, \dots, m\}$.

▶ **Lemma 2.** *If $\alpha \in (T \cup N)^*$ can be derived by G_*, then one of these conditions holds:*

1. $\exists x_1, x_2, \dots, x_m \in (T \cup N)^*\ (\forall i \in [m]\ (S \Rightarrow^* x_i)\ \wedge\ \alpha = Zx_1\#x_2\#\dots x_m\#)$;
2. $\exists x_1, x_2, \dots, x_m \in (T \cup N)^*\ (\forall i \in [m]\ (S \Rightarrow^* x_i)\ \wedge\ \alpha = R\#x_1\#x_2\#\dots x_m\#)$;
3. $\exists w_1, w_2, \dots, w_n \in L\ (\exists \beta \in T^*\ (\exists \gamma, x_1, x_2, \dots, x_m \in (T \cup N)^*$
   $(S \Rightarrow^* \beta\gamma\ \wedge\ \forall i \in [m]\ (S \Rightarrow^* x_i)\ \wedge\ \alpha = w_1w_2\dots w_n\,\beta R\gamma\#x_1\#x_2\#\dots x_m\#)))$;
4. $\alpha \in L^*$;
5. $\exists \sigma \in (T \cup N)^*\ (\alpha = \sigma R)$;
6. $\exists \omega \in (T \cup N \cup \{\#\})^*\ (\alpha = \omega\#)$.

In the example above, case 1 arises when $\alpha = ZaaSbb\#S\#$. We can check that $m = 2$, $x_1 = aaSbb$, $x_2 = S$, and condition 1 holds.

In the example above, case 2 arises when $\alpha = R\#aSb\#aaabbb\#S\#$. We can check that $m = 3$, $x_1 = aSb$, $x_2 = aaabbb$, $x_3 = S$, and condition 2 holds.

In the example above, case 3 arises when $\alpha = abaaabRbb\#aSb\#$. We can check that $n = 1$, $m = 1$, $w_1 = ab$, $\beta = aaab$, $\gamma = bb$, $x_1 = aSb$, and condition 3 holds.

Case 4 arises only at the end of a successful computation, which is $\alpha = abaaabbbab$ in the example above.

The remaining two cases do not arise in the example above because they describe an unsuccessful computation (like taking a one-way street ending in a blind alley).

Case 5 arises if the rule $R\# \to R$ is used in the final position (where $R\# \to \epsilon$ should be used instead). The nonterminal R in the final position prevents the derivation from terminating.

Case 6 arises if the rule $R\# \to \epsilon$ is used too early (that is, anywhere but the final $\#$ position). The nonterminal $\#$ in the final position during the absence of R and Z in α prevents the derivation from terminating.

Proof. By induction on G_* derivation steps. The base case $\alpha = Z$ satisfies condition 1 by setting $m = 0$.

Now assume $Z \Rightarrow^* \alpha \Rightarrow \alpha'$ and proceed by case analysis on the conditions.

1. $\exists x_1, x_2, \ldots, x_m \in (T \cup N)^* \, (\forall i \in [m] \, (S \Rightarrow^* x_i) \;\wedge\; \alpha = Zx_1\#x_2\#\ldots x_m\#)$:
   - If $\alpha \Rightarrow \alpha'$ used a rule from P, it could be applied only in some x_i. Hence $S \Rightarrow^* x_i \Rightarrow x_i'$, so the same condition holds after replacing x_i by x_i'.
   - If $\alpha \Rightarrow \alpha'$ used the rule $Z \to ZS\#$, it was applied at the beginning of α. Therefore, we set $m' := m + 1$, we set $x_1' := S$, and we increase all indices by one, that is, $x_2' := x_1,\ x_3' := x_2,\ \ldots,\ x_{m'}' := x_m$. The same condition holds.
   - If $\alpha \Rightarrow \alpha'$ used the rule $Z \to R\#$, we keep all variables the same and condition 2 holds.
   - The rules $R\# \to R$, $R\# \to \epsilon$, and $Rt \to tR$ are not applicable (since α does not contain R).
2. $\exists x_1, x_2, \ldots, x_m \in (T \cup N)^* \, (\forall i \in [m] \, (S \Rightarrow^* x_i) \;\wedge\; \alpha = R\#x_1\#x_2\#\ldots x_m\#)$:
   - If $\alpha \Rightarrow \alpha'$ used a rule from P, it could be applied only in some x_i. Hence $S \Rightarrow^* x_i \Rightarrow x_i'$, so the same condition holds after replacing x_i by x_i'.
   - The rules $Z \to ZS\#$ and $Z \to R\#$ are not applicable (since α does not contain Z).
   - If $\alpha \Rightarrow \alpha'$ used the rule $R\# \to R$, it was applied at the beginning of α. If $m = 0$, condition 5 holds (a dead end). Otherwise, we set $m' := m - 1 \geq 0$ and $\gamma := x_1$, and we decrease all indices by one, that is, $x_1' := x_2,\ x_2' := x_3,\ \ldots,\ x_{m'}' := x_m$. Since there is nothing before the nonterminal R, we set $n := 0$ and $\beta := \epsilon$. Now, condition 3 holds.
   - If $\alpha \Rightarrow \alpha'$ used the rule $R\# \to \epsilon$ then: if $m = 0$, we obtain the empty word (which belongs to L^*, satisfying condition 4); if $m > 0$, condition 6 holds (because $\#$ remained at the end of α'; at the same time R disappeared, and Z did not appear).
   - The rule $Rt \to tR$ is not applicable (the only R in α is immediately followed by $\#$).
3. $\exists w_1, w_2, \ldots, w_n \in L \, (\exists \beta \in T^* \, (\exists \gamma, x_1, x_2, \ldots, x_m \in (T \cup N)^*$
   $(S \Rightarrow^* \beta\gamma \;\wedge\; \forall i \in [m] \, (S \Rightarrow^* x_i) \;\wedge\; \alpha = w_1 w_2 \ldots w_n \, \beta R \gamma \# x_1 \# x_2 \# \ldots x_m \#)))$:
   - If $\alpha \Rightarrow \alpha'$ used a rule from P, it could be applied in γ or in some x_i. In the first case, $\gamma \Rightarrow \gamma'$ implies $\beta\gamma \Rightarrow \beta\gamma'$, hence $S \Rightarrow^* \beta\gamma \Rightarrow \beta'\gamma'$. In the remaining cases, we observe $S \Rightarrow^* x_i \Rightarrow x_i'$ as we did at the beginning of our case analysis. As a result, the same condition still holds.
   - The rules $Z \to ZS\#$ and $Z \to R\#$ are not applicable (α does not contain Z).
   - If $\alpha \Rightarrow \alpha'$ used the rule $R\# \to R$, then γ must have been empty. If $m = 0$, condition 5 holds (a dead end). Otherwise, we set $n' := n + 1$, $w_{n'} := \beta$, $\beta' := \epsilon$, $\gamma' := x_1$, and $m' := m - 1$, and we decrease the indices of x_i by one, that is, $x_1' := x_2,\ x_2' := x_3,\ \ldots,\ x_{m'}' := x_m$. Since $w_{n'} = \beta = \beta\gamma \in T^*$ and $S \Rightarrow^* \beta\gamma$, we have $w_{n'} \in L$. The same condition holds.
   - If $\alpha \Rightarrow \alpha'$ used the rule $R\# \to \epsilon$, then γ must have been empty. If $m = 0$, we get $\alpha = w_1 w_2 \ldots w_n \beta$ and $\beta \in L$; hence condition 4, $\alpha' \in L^*$, is satisfied. If $m > 0$, condition 6 now holds (because $\#$ remained at the end of α'; at the same time R disappeared, and Z did not appear).
   - If $\alpha \Rightarrow \alpha'$ used a rule of the form $Rt \to tR$ ($t \in T$), we have $\delta \in (T \cup N)^*$ such that $\gamma = t\delta$. We put $\beta' := \beta t$ and $\gamma' := \delta$. Since $\beta\gamma = \beta t \delta = \beta'\gamma'$, the same condition holds.
4. $\alpha \in L^*$:
   - No rule is applicable (since α contains only terminals). The step $\alpha \Rightarrow \alpha'$ cannot have happened.
5. $\exists \sigma \in (T \cup N)^* \, (\alpha = \sigma R)$:
   - No matter which rule was applied, it happened within σ. No rule could match the final R. The same condition holds for $\alpha' = \sigma' R$.

6. $\exists \omega \in (T \cup N \cup \{\#\})^* \; (\alpha = \omega\#)$:
   - If $\alpha \Rightarrow \alpha'$ used a rule from P, the same condition still holds because $\#$ is not on the left-hand side of any rule from P and neither Z nor R is on the right-hand side of any rule from P.
   - The rules $Z \to ZS\#$, $Z \to R\#$, $R\# \to R$, $R\# \to \epsilon$, and $Rt \to tR$ are not applicable (since α contains neither Z nor R). ◀

▶ **Theorem 3.** *The class of type-0 languages is closed under the Kleene star.*

Proof. We need to show that the language of G_* equals L^*. We prove two inclusions.

For "$\supseteq L^*$", we use Lemma 1. If $w \in L^*$, there exist words $w_1, w_2, \ldots, w_n \in L$ such that $w_1 w_2 \ldots w_n = w$. We see $Zw_1\#w_2\# \ldots w_n\# \Rightarrow R\#w_1\#w_2\# \ldots w_n\#$. Since all words w_i are made of terminals only, by repeated application of $R\# \to R$ and $Rt \to tR$ (for all $t \in T$) we get $R\#w_1\#w_2\# \ldots w_n\# \Rightarrow^* w_1 w_2 \ldots w_n R\#$. Finally, $w_1 w_2 \ldots w_n R\# \Rightarrow w_1 w_2 \ldots w_n$ is obtained by the rule $R\# \to \epsilon$. We conclude that G_* generates w.

For "$\subseteq L^*$", we use Lemma 2 and observe that if G_* generates $\alpha \in T^*$, then $\alpha \in L^*$ because all the remaining cases require α to contain a nonterminal. ◀

6.2 Formalization

The formalization closely follows the proof sketch. The main difference between the two is that where the proof sketch states that an expression belongs to a set, the formalization specifies a type for a term and sometimes a condition that further restricts the term's values.

Lemma 1 is implemented by lemma `short_induction`, which takes w in reverse order for technical reasons. Its proof uses the `lifted_grammar` approach outlined in Section 3. The part $R\#w_1\#w_2\# \ldots w_n\# \Rightarrow^* w_1 w_2 \ldots w_n R\#$ is implemented by lemma `terminal_scan_ind`, which employs a nested induction to pass R to the right. The final step of the easy direction is performed inside the theorem `T0_of_star_T0` itself.

Lemma 2 is implemented by lemma `star_induction`, whose formal proof spans over 3000 lines. The base case is discharged immediately. For the induction step, we developed six lemmas `star_case_1` to `star_case_6` distinguished by which of the six conditions α satisfies. In each of them, except for `star_case_4`, which took only four lines to prove, we perform a case analysis on which rule was used for the $\alpha \Rightarrow \alpha'$ transition.

For each case, unless a short ex-falso-quodlibet proof suffices, we need to narrow down where in α the rule could be applied. This analysis is challenging for the rules that were inherited from the original grammar. Consider `case_1_match_rule`, where the informal argument literally says: "If $\alpha \Rightarrow \alpha'$ used a rule from P, it could be applied only in some x_i."

It turns out that this deduction is so complicated that it was worth creating an auxiliary lemma `cases_1_and_2_and_3a_match_aux` to detach the head Z from α and perform the analysis on $x_1\#x_2\# \ldots x_m\#$ in order to make the proof easier. As a useful side effect, the auxiliary lemma becomes applicable to similar situations in `star_case_2` and `star_case_3`, as shown in `case_2_match_rule` and `case_3_match_rule`, where more adaptations are needed but the same core argument is used.

From a formal point of view, we abused the symbol '…' in the proof sketch. Replacing it by a formal statement usually leads to `list.join` of `list.map` of something. For example, compare case 1 in the proof sketch

$$\exists x_1, x_2, \ldots, x_m \in (T \cup N)^* \; (\forall i \in [m] \; (S \Rightarrow^* x_i) \;\wedge\; \alpha = Zx_1\#x_2\# \ldots x_m\#)$$

to its formal counterpart:

```
∃ x : list (list (symbol T g.nt)),
  (∀ xᵢ ∈ x, grammar_derives g [symbol.nonterminal g.initial] xᵢ) ∧
  (α = [Z] ++ list.join
    (list.map (++ [H]) (list.map (list.map wrap_sym) x)))
```

The nonterminal # is represented by the letter `H` in the code. Notice how easy it is to write the quantification $\exists x_1, x_2, \ldots, x_m \in (T \cup N)^*$ in Lean. The part $\forall i \in [m]\ (S \Rightarrow^* x_i)$ is also elegant. However, the expression $Zx_1\#x_2\#\ldots x_m\#$ leads to a fairly complicated Lean term.

Because many lemmas need to work with expressions like the above, it is important to master how to manipulate terms that combine `list.join` with other functions. For example, the following lemma is useful:

```
lemma append_join_append {s : list α} (L : list (list α)) :
  s ++ (list.map (λ l, l ++ s) L).join =
       (list.map (λ l, s ++ l) L).join ++ s
```

This lemma allows us to move the parentheses in $s(l_1 s)(l_2 s)\ldots(l_n s)$ to get $(sl_1)(sl_2)\ldots(sl_n)s$ and vice versa.

Working with expressions such as $Zx_1\#x_2\#\ldots x_m\#$ is tedious in Lean. We see this, however, not as a weakness of Lean but rather as an indication that the '...' notation is highly informal. Mathematical expressions with '...' tend to be ambiguous and require the reader's cooperation to make sense of them. In the absence of support for '...' in the proof assistant [19], it is natural that formalizing such expressions leads to verbose code.

In contrast to concatenation, the above proof cannot be reused to establish the closure of context-free languages under the Kleene star because our construction adds rules with two symbols on their left-hand side. However, there exists an easier construction for context-free languages that could be formalized separately if desired.

7 Related Work

To our knowledge, no one has formalized general grammars before. Context-free grammars were formalized by Carlson et al. [5] using Mizar, by Minamide [23] using Isabelle/HOL, by Barthwal and Norrish [4] using HOL4, by Firsov and Uustalu [11] using Agda, and by Ramos [25] using Coq.

Finite automata have often been subjected to verification. In particular, Thompson and Dillies [22] formalized finite automata, which recognize regular languages, using Lean. Thomson [22] also formalized regular expressions, which recognize regular languages as well.

There is ample verification work also for other models of computation:

- Turing machines were formalized using Mizar [7], Matita [2], Isabelle/HOL [26], Lean [6], Coq [15], and recently again Isabelle/HOL [3]. Of these, the most impressive development is probably the last one, by Balbach. It uses multi-tape Turing machines and culminates with a proof of the Cook–Levin theorem, which states that SAT is **NP**-complete.
- The λ-calculus was formalized by Norrish [24] using HOL4 and later by Forster, Kunze, and their colleagues [16, 20, 12, 13, 14, 17] using Coq. The latter group of authors proposed an untyped call-by-value λ-calculus as a convenient basis for computability and complexity theory because it naturally supports compositionality.
- The partial recursive functions were formalized by Norrish [24] using HOL4 and by Carneiro [6] using Lean.
- Random access machines were formalized by Coen [8] using Coq.

8 Conclusion

We defined general grammars in Lean and used them to establish closure properties of recursively enumerable or type-0 languages. We found that closure under union and reversal were straightforward to formally prove, but had to invest considerable effort to prove closure under concatenation and the Kleene star. Despite the tedium of some of the proofs, we believe that grammars are probably a more convenient formalism than Turing machines for showing closure properties. On the other hand, since grammars do not define any of the important complexity classes (such as **P**), formalization of Turing machines and other computational models is needed to further develop the formal theory of computer science.

As future work, results about context-sensitive, context-free, and regular grammars could be incorporated into our library. A comprehensive Lean library encompassing the entire Chomsky hierarchy would be valuable. We already have some results about context-free grammars, and the `mathlib` results about regular languages could be connected to our library. As a more ambitious goal, we might attempt to prove the equivalence between general grammars and Turing machines.

References

1 Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall, 1st edition, 1972.

2 Andrea Asperti and Wilmer Ricciotti. Formalizing Turing Machines. In *Logic, Language, Information and Computation*, volume 7456 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2012. `doi:10.1007/978-3-642-32621-9_1`.

3 Frank J. Balbach. The Cook-Levin theorem. *Archive of Formal Proofs*, 2023. `https://isa-afp.org/entries/Cook_Levin.html`, Formal proof development.

4 Aditi Barthwal and Michael Norrish. Mechanisation of PDA and Grammar Equivalence for Context-Free Languages. In Anuj Dawar and Ruy de Queiroz, editors, *WoLLIC 2010*, volume 6188 of *Lecture Notes in Computer Science*, pages 125–135. Springer, 2010. `doi:10.1007/978-3-642-13824-9_11`.

5 Patricia L. Carlson, Grzegorz Bancerek, and Im Pan. Context-Free Grammar — Part 1. *J. Formaliz. Math.*, 1992. URL: `http://mizar.org/JFM/pdf/lang1.pdf`.

6 Mario Carneiro. Formalizing Computability Theory via Partial Recursive Functions. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *ITP 2019*, volume 141 of *LIPIcs*, pages 12:1–12:17. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ITP.2019.12`.

7 Jing-Chao Chen and Yatsuka Nakamura. Introduction to Turing Machines. *J. Formaliz. Math.*, 9(4), 2001. URL: `https://fm.mizar.org/2001-9/pdf9-4/turing_1.pdf`.

8 Claudio Sacerdoti Coen. A Constructive Proof of the Soundness of the Encoding of Random Access Machines in a Linda Calculus with Ordered Semantics. In Carlo Blundo and Cosimo Laneve, editors, *ICTCS 2003*, volume 6188 of *Lecture Notes in Computer Science*, pages 37–57. Springer, 2003. `doi:10.1007/978-3-540-45208-9_5`.

9 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. `doi:10.1007/978-3-319-21401-6_26`.

10 Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.*, 1(ICFP):1–29, 2017. `doi:10.1145/3110278`.

11 Denis Firsov and Tarmo Uustalu. Certified Normalization of Context-Free Grammars. In *CPP 2015*, pages 167–174. ACM, 2015. `doi:10.1145/2676724.2693177`.

12 Yannick Forster and Fabian Kunze. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *ITP 2019*, volume 141 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ITP.2019.17`.

13 Yannick Forster, Fabian Kunze, and Marc Roth. The Weak Call-by-Value Lambda-Calculus is Reasonable for Both Time and Space. *Proc. ACM Program. Lang.*, 4(POPL):27:1–27:23, 2019. `doi:10.1145/3371095`.

14 Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. A Mechanised Proof of the Time Invariance Thesis for the Weak Call-By-Value λ-Calculus. In Liron Cohen and Cezary Kaliszyk, editors, *ITP 2021*, volume 193 of *LIPIcs*, pages 19:1–19:20. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ITP.2021.19`.

15 Yannick Forster, Fabian Kunze, and Maximilian Wuttke. Verified Programming of Turing Machines in Coq. In *CPP 2020*, pages 114–128. ACM, 2020. `doi:10.1145/3372885.3373816`.

16 Yannick Forster and Gert Smolka. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *ITP 2017*, volume 10499 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2017. `doi:10.1007/978-3-319-66107-0_13`.

17 Lennard Gäher and Fabian Kunze. Mechanising Complexity Theory: The Cook-Levin Theorem in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *ITP 2021*, volume 193 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ITP.2021.20`.

18 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 3rd edition, 2007.

19 Fulya Horozal, Florian Rabe, and Michael Kohlhase. Flexary Operators for Formalized Mathematics. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics*, Lecture Notes in Computer Science, pages 312–327. Springer, 2014. `doi:10.1007/978-3-319-08434-3_23`.

20 Fabian Kunze, Gert Smolka, and Yannick Forster. Formal Small-Step Verification of a Call-by-Value Lambda Calculus Machine. In Sukyoung Ryu, editor, *APLAS 2018*, volume 11275 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2018. `doi:10.1007/978-3-030-02768-1_15`.

21 Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990. URL: `https://era.ed.ac.uk/bitstream/handle/1842/12487/Luo1990.Pdf`.

22 The `mathlib` Community. The Lean Mathematical Library. In Jasmin Blanchette and Cătălin Hriţcu, editors, *CPP 2020*, pages 367–381. ACM, 2020. `doi:10.1145/3372885.3373824`.

23 Yasuhiko Minamide. Verified Decision Procedures on Context-Free Grammars. In Klaus Schneider and Jens Brandt, editors, *TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007. `doi:10.1007/978-3-540-74591-4_14`.

24 Michael Norrish. Mechanised Computability Theory. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *ITP 2011*, volume 6898 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011. `doi:10.1007/978-3-642-22863-6_22`.

25 Marcus Vinícius Midena Ramos. Formalization of Context-Free Language Theory. *Bull. Symbol. Log.*, 25(2):214–214, 2019. `doi:10.1017/bsl.2019.3`.

26 Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising Turing Machines and Computability Theory in Isabelle/HOL. In *Interactive Theorem Proving*, volume 7998, pages 147–162. Springer Berlin Heidelberg, 2013. Lecture Notes in Computer Science. `doi:10.1007/978-3-642-39634-2_13`.