A formal proof of the equivalence between pushdown automata and context-free grammars

Tobias Leichtfried

January 2025

Abstract

We introduce pushdown automata and context-free grammars and their associated language classes. We then provide a formalization of these definitions in the interactive proof assistant Lean. Further we show that this two language classes are equal and provide a formalized proof of this fact in Lean.

1 Introduction

The goal of this bachelor thesis was to formalize a proof of the equivalence of pushdown automata and context-free grammars in the interactive proof assistant *Lean*. This document is intended to give an overview of the resulting formal proof. We will introduce pushdown automata, context-free grammars and the main theorems about them, as well as compare them with their formalization in Lean.

2 Pushdown Automata and Context-Free Grammars

The PDA can be (informally) imagined as a machine consisting of states Q equipped with a tape from which the input is read and a form of memory called *stack*. In each step of the computation the following happens: The input tape is moved one letter forward, this letter is then consumed, the topmost symbol of the stack is consumed and according the the combination of state, letter and stack symbol which the machine has now ingested, it moves in a new state and pushes an arbitrary long string of symbols on the stack. This happens, possibly, in a nondeterministic manner. So a given triple of letter, state and stack symbol may allow many different next states and stack pushes. Also the consumption of a letter from the input tape is optional. If the machine does not perform a read, the behaviour only depends on state and current stack symbol. The combination of remaining input tape, state and stack is called a configuration. More formally:

Definition 1. *A pushdown automaton (PDA) is a tuple* $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ *where*

1. *Q is the finite set of* states
2. *Σ is the alphabet of the input*
3. *Γ is the alphabet of the* stack
4. *$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is the* transition function*, fullfilling $|\delta(q, a, Z)| < \infty$ for all $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$*
5. *$q_0 \in Q$ is the* initial state
6. *$Z_0 \in \Gamma$ is the* start symbol
7. *$F \subseteq Q$ are the* final states

In the whole documentation excerpts of lean source code and traditional mathematics will be interlaced, to complement each other. The traditional mathematics to document and explain the ideas behind the source code, and the source code to demonstrate the practical implementation.

```
structure PDA (Q T S : Type) [Fintype Q] [Fintype T] [Fintype S] where
  initial_state : Q
  start_symbol : S
  final_states : Set Q
  transition_fun : Q → T → S → Set (Q × List S)
  transition_fun' : Q → S → Set (Q × List S)
  finite (q : Q)(a : T)(Z : S): (transition_fun q a Z).Finite
  finite' (q : Q)(Z : S): (transition_fun' q Z).Finite
```

A tuple translates usually to a structure in Lean, while it would be possible to define the PDA directly as tuple the ability to name fields in a structure makes working with the so defined PDA less cumbersome.

If we compare the structure in the source code listing with the definition given before, we see two additional fields (`finite, finite'`) and notice that the transition function looks somewhat different.

While the original definition of a PDA uses just one transition function to model both computation steps which read from the input and which do not read from the input (ε-transistion), this distinction is made into two transition functions in the Lean source code. While it would be possible to just use one transition function at this point, definitions later on would be more convoluted if this distinction where not made.

The fields `finite` and `finite'` contain proofs that the transistion function fulfills $|\delta(q, a, Z)| < \infty$ forall $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$. This means that, if one wants to construct a `PDA` given `initial_state`, `start_symbol`, `final_state`, `transition_fun`, `transition_fun'` they still need a proof that this requirement holds.

Definition 2. *We call a Tuple $(q, x, \alpha) \in Q \times \Sigma^* \times \Gamma^*$ a* configuration *of the PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.*

```
structure conf (p : PDA Q T S) where
  state : Q
  input : List T
  stack : List S
```

Here we note that the structure `conf` depends on a `PDA` p. So for every `PDA` there exists a seperate type `conf p`

Definition 3. *We say* $(q, x, \alpha) \vdash^1 (p, y, \beta)$ *or configuration* (q, x, α) reaches (p, y, β) in one step *iff there exist* $a \in \Sigma \cup \{e\}$, $Z \in \Gamma$ *and* $\nu, \mu \in \Gamma^*$ *so that* $x = ay$, $\alpha = Z\nu$, $\beta = \mu\nu$ *and* $(p, \mu) \in \delta(q, a, Z)$

For $n \in \mathbb{N}$ *fullfilling* $n \geq 2$ *we say* $(q, x, \alpha) \vdash^n (p, y, \beta)$ *or configuration* (q, x, α) reaches (p, y, β) in n steps *iff there exist* $n - 1$ *configurations* c_i *so that* $(q, x, \alpha) \vdash^1 c_1 \vdash^1 \cdots \vdash^1 c_{n-1} \vdash^1 (p, y, \beta)$ *Additionally we say* $(q, x, \alpha) \vdash^0 (p, y, \beta)$ *iff* $(q, x, \alpha) = (p, y, \beta)$.

Finally we say $(q, x, \alpha) \vdash (p, y, \beta)$ *or configuration* (q, x, α) reaches (p, y, β) *iff there exists* $n \in \mathbb{N}$ *so that* $(q, x, \alpha) \vdash^n (p, y, \beta)$.

```
def step (r₁ : conf pda) : Set (conf pda) :=
  match r₁ with
    | ⟨q, a::w, Z::α⟩ =>
        { r₂ : conf pda | ∃ (p : Q) (β : List S), (p,β) ∈ pda.transition_fun q a Z ∧
                              r₂ = ⟨p, w, (β ++ α)⟩ } ∪
        { r₂ : conf pda | ∃ (p : Q) (β : List S), (p,β) ∈ pda.transition_fun' q Z ∧
                              r₂ = ⟨p, a :: w, (β ++ α)⟩ }
    | ⟨q, [], Z::α⟩ => { r₂ : conf pda | ∃ (p : Q) (β : List S),
                          (p,β) ∈ pda.transition_fun' q Z ∧ r₂ = ⟨p, [], (β ++ α)⟩ }
    | ⟨_, _, []⟩ => ∅

def Reaches₁ (r₁ r₂ : conf pda) : Prop := r₂ ∈ step r₁
def Reaches : conf pda → conf pda → Prop := Relation.ReflTransGen Reaches₁

inductive ReachesIn : ℕ → conf pda → conf pda → Prop where
  | refl : (r₁ : conf pda)  → ReachesIn 0 r₁ r₁
  | step : {n: ℕ} → {r₁ r₂ r₃ : conf pda} → ReachesIn n r₁ r₂ → Reaches₁ r₂ r₃ →
    ReachesIn (n+1) r₁ r₃
```

Comparing these two definitions the first noteable fact is the `step` function, which only exists in the Lean code. The function `step` receives a configuration of a `PDA` as input and returns the set of possible next configurations. Looking closely we recognize two different sets, one corresponding to `transistion_fun` and one to `transistion_fun'`, so one modeling computation with read and one computation without read. The definition of these sets is subtle different, demonstrating the need for seperating `transistion_fun` and `transistion_fun'` instead of using one transition function as in the mathematical definiton.

The relation `Reaches₁` is defined in the obvious way, less obvious are `Reaches` and `ReachesIn`. The definition of `Reaches` uses a feature of *Mathlib*, the Lean library of formalized mathematics, the *reflexive, transitive closure*. This is consistent with the defintion of $\vdash$ but more idiomatic than the traditional definition given. The relation `ReachesIn` is defined inductively, in manner virtually the same as the implementation of `Relation.ReflTransGen` but counting the steps of computation along the way.

This distinction is important as the fundamental method of proof in Lean is structural induction. The induction principle generated for `Reaches` is somewhat weak, as it only allows to split the computation at the first or last step of computation. When using `ReachesIn`, one can use the strong induction of the natural numbers on the number of computation steps. This allows splitting the computation in more complicated parts, manipulating them in non obvious ways and still being able to apply the induction hypthesis.

Definition 4. *For a PDA M* $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ *we define*

$$N(M) = \{w \in \Sigma^* \mid \exists q \in Q : (q_0, w, Z_0) \vdash (q, \varepsilon, \varepsilon)\}$$

the Language of the PDA accepted by empty stack.

```
def acceptsByEmptyStack (pda : PDA Q T S) : Language T :=
  { w : List T | ∃ q : Q,
      Reaches (⟨pda.initial_state, w, [pda.start_symbol]⟩ : conf pda) ⟨q, [], []⟩ }
```

The definition of the language of the PDA is as expected, noteworthy is the type annotation `Language T`. This is the Mathlib type of a language over the alphabet `T` and is definitionally equal to the type `Set (List T)`.

It is clear that $\vdash$ is reflexive and transitive and so is `Reaches`. To use these properties in Lean they need to be proven. This is not particularly challenging as `Relation.ReflTransGen` already provides corresponding theorems.

```
theorem Reaches.refl (r₁ : conf pda) : Reaches r₁ r₁ := Relation.ReflTransGen.refl

theorem Reaches.trans {r₃ : conf pda} (h₁ : Reaches r₁ r₂) (h₂ : Reaches r₂ r₃) :
    Reaches r₁ r₃ := Relation.ReflTransGen.trans h₁ h₂
```

Following properties of `ReachesIn`are easily proved :

```
theorem reachesIn_zero (h: ReachesIn 0 r₁ r₂) : r₁ = r₂

theorem reaches₁_iff_reachesIn_one : Reaches₁ r₁ r₂ ↔ ReachesIn 1 r₁ r₂

theorem reachesIn_one : ReachesIn 1 r₁ r₂ ↔ r₂ ∈ step r₁
```

And the next three very useful properties require a little work and induction.

```
theorem reachesIn_iff_split_last {n : ℕ} :
    (∃ c : conf pda, ReachesIn n r₁ c ∧ ReachesIn 1 c r₂) ↔ ReachesIn (n+1) r₁ r₂

theorem reachesIn_iff_split_first {n : ℕ}:
    (∃ c : conf pda, ReachesIn 1 r₁ c ∧ ReachesIn n c r₂) ↔ ReachesIn (n+1) r₁ r₂

theorem reaches_iff_reachesIn : Reaches r₁ r₂ ↔ ∃ n : ℕ, ReachesIn n r₁ r₂
```

We will now examine the proof of a simple lemma closer, before embarking to more interesting matters. The lemma states that after a single step of computation the input of the PDA either stays the same or a prefix is removed. No other change is possible. This is consistent with the interpretation of a PDA as a machine consuming input from a tape in a sequential manner. In fact the input decreases by at most one letter, this is not part of the lemma for a reason: This lemma will be used only to prove a similar statement for arbitrary many steps of computation, where of course no such restriction applies. The proof below is more verbose than necessary, in order to allow it to be stepped through in an interactive manner easily.

```
theorem decreasing_input_one (h : ReachesIn 1 r₁ r₂) :
    ∃ w : List T, r₁.input = w ++ r₂.input := by
  apply reachesIn_one.mp at h          -- Apply characterization of ReachesIn 1
  rcases r₁ with ⟨q, w,  _ | ⟨Z, β⟩⟩     -- To simplify step we have to split cases
  · simp [step] at h                   -- If the stack is empty no computation can happen
  · rcases w with  _ | ⟨a, w⟩           -- Again case split if a read is possilbe
    · dsimp [step] at h
      obtain ⟨_,_,h⟩ := h               -- If the tape is empty no read can happen
      use []
      simp [h.2]                       -- Closes the goal
    · dsimp [step] at h
      rw [Set.mem_union] at h          -- Convert membership of union to or
      rcases h with h|h                -- Split cases on wether a read is happening
      · rw [Set.mem_setOf] at h        -- Convert membership of set builder to predicate
        obtain ⟨p,β,h⟩ := h             -- We know that a is read
        use [a]
        simp [h.2]                     -- Closes the goal
      · obtain ⟨_,_,h⟩ := h             -- No read is happening, so as before
        use []
        simp [h.2]
```

This lemma is used to prove the following theorem:

```
theorem decreasing_input (h : Reaches r₁ r₂) : ∃ w : List T, r₁.input = w ++ r₂.input
```

The source code contains a few more useful lemmas about `ReachesIn` which we will encounter looking at the proofs of the main results of the formalization. Before continuing we introduce *context-free grammars* and their formalization in Mathlib.

Definition 5. *A context-free grammar (CFG) is a Tuple* (T, N, P, S) *where*

1. T *is the finite set of* terminals
2. N *is a finite set of* nonterminals
3. $P \subseteq N \times (T \cup N)^*$ *is a finite set of* production rules *(we often write* $A \to \alpha$ *instead of* (A, α) *for elements of* P*)*
4. S *is the* start symbol

For two strings of terminal and nonterminal symbols $v, w \in (T \cup N)^*$ *we say* v derives w *in one step (or in symbols* $v \Rightarrow^1 w$*) iff:*

$$\exists v_1, v_2 \alpha \in (N \cup T)^*, A \in N : v = v_1 A v_2 \ \wedge \ w = v_1 \alpha v_2 \ \wedge \ (A, \alpha) \in P$$

Smiliar to PDAs we define $\Rightarrow^0, \Rightarrow^n, \Rightarrow$*. We call* $L(G) = \{w \in T^* \mid S \Rightarrow w\}$ *the language generated by* G*.*

In Mathlib the derives relation is called `Derives` and is defined very similar to `Reaches`. If one replaces the definition of $\Rightarrow^1$ with

$$\exists v_1 \in T^*, v_2, \alpha \in (N \cup T)^*, A \in N : v = v_1 A v_2 \ \wedge \ w = v_1 \alpha v_2 \ \wedge \ (A, \alpha) \in P$$

one obtains the definition of leftmost derivations. It can be shown that, if we replace derivation with leftmost derivation in the definition of $L(G)$, the resulting language stays the same. To make proofs easier we will only work with leftmost derivations from now on (and interpret $\Rightarrow^n, \Rightarrow$ accordingly). The proofs in this formalization make thus use of two variations of `Derives` not provided in Mathlib, namely `DerivesLeftmost` and `DerivesLeftmostIn`. At this point it should be noted that Mathlib also does not provide a `DerivesIn` relation. The implementation of `DerivesLeftmost` provided in the formalization is a pull request to Mathlib currently in review.

3 CFG to PDA

With this vocabulary at our disposal we can now begin with the first major result of the formalizia-tion:

Theorem 1. *Let G be a CFG with $L = L(G)$ then there exists a PDA M so that $N(M) = L$.*

Proof. Let $G = (N, T, P, S)$ be a context-free grammar. We construct a PDA M, and show N(M)=L(G). So $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is defined as follows:

$$Q = \{q_0\} \qquad \Sigma = T \qquad \Gamma = T \cup N \qquad Z_0 = S \qquad F = \emptyset$$

$$\delta(q_0, a, Z) = \begin{cases} \{(q_0, \beta) \mid Z \to \beta \in P\} & \text{if} \quad a = \varepsilon \wedge Z \in N \\ \{(q_0, \varepsilon)\} & \text{if} \quad a \in T \wedge Z \in T \wedge a = Z \\ \emptyset & \text{else} \end{cases}$$

We show now L(G)=N(M). So let $w \in L(G)$ be arbitrary. So we know that there exists a sequence of leftmost derivations

$$S \Rightarrow_G^1 \alpha_1 \Rightarrow_G^1 \cdots \Rightarrow_G^1 \alpha_n \Rightarrow_G^1 w$$

by induction on the number of steps we show that there exists a computation

$$(q_0, w, S) \vdash_M^1 c_1 \vdash_M^1 \cdots \vdash_M^1 c_m \vdash_M^1 (q_0, \varepsilon, \varepsilon).$$

We need however a slightly stronger induction hypothesis. Instead of S we will work with $\alpha \in (N \cup T)^*$ and $w \in T^*$. For the base case we have $\alpha \Rightarrow_G^0 w$ this means $\alpha = w$. Per construction of M we know $(q_0, \varepsilon) \in \delta(q_0, a, a)$, by repeatedly applying this we obtain $(q_0, w, w) \vdash_M (q_0, \varepsilon, \varepsilon)$ and conclude the base case. Now assuming $\forall \alpha : (\alpha \Rightarrow_G^n w \implies (q_0, w, \alpha) \vdash_M (q_0, \varepsilon, \varepsilon))$, we want to show the same for $\alpha \Rightarrow_G^{n+1} w$. If $\alpha \Rightarrow_G^{n+1} w$ we know there exists a $\alpha_1 \in (N \cup T)^*$ so that

$$\alpha \Rightarrow_G^1 \alpha_1 \Rightarrow_G^n w.$$

Because $\alpha \Rightarrow_G^1 \alpha_1$ we can write $\alpha = w_1 A \alpha'$ and $\alpha_1 = w_1 \beta \alpha'$ where $w_1 \in T^*$, $\alpha', \beta \in (N \cup T)^*$ and $A \to \beta \in P$. By applying the induction hypotheses we obtain $(q_0, w, \alpha_1) \vdash_M (q_0, \varepsilon, \varepsilon)$. That is $(q_0, w, w_1 \beta \alpha') \vdash_M (q_0, \varepsilon, \varepsilon)$, our construction of M guarantees then that following computation is happening: $(q_0, w, w_1 \beta \alpha') \vdash_M (q_0, w', \beta \alpha') \vdash_M (q_0, \varepsilon, \varepsilon)$ with $w = w_1 w'$ for some $w' \in T^*$. Similarly as in the base case we have $(q_0, w, \alpha) = (q_0, w_1 w', w_1 A \alpha') \vdash_M (q_0, w', A\alpha')$. As $A \to \beta \in P$ we also know $(q_0, w', A\alpha') \vdash_M (q_0, w', \beta\alpha')$. It suffices now that $(q_0, w', \beta\alpha') \vdash_M (q_0, \varepsilon, \varepsilon)$, which we already established.

For the other direction let again $w \in T^*$ be arbitrary. We again show by induction on the number of computation steps $(q_0, w, \alpha) \vdash_M^n (q_0, \varepsilon, \varepsilon)$ implies $\alpha \Rightarrow_G w$ for every $w \in T^*, \alpha \in (T \cup N)^*$. For the base case we have $(q_0, w, \alpha) \vdash_M^0 (q_0, \varepsilon, \varepsilon)$. This implies $w = \varepsilon$ and $\alpha = \varepsilon$, so we see $\alpha \Rightarrow_G w$. For the induction step we assume $\forall \alpha : ((q_0, w, \alpha) \vdash_M^n (q_0, \varepsilon, \varepsilon) \implies \alpha \Rightarrow_G w)$ and $(q_0, w, \alpha) \vdash_M^{n+1} (q_0, \varepsilon, \varepsilon)$. So

$$(q_0, w, \alpha) \vdash_M^1 (q_0, w', \alpha_1) \vdash_M^n (q_0, \varepsilon, \varepsilon).$$

Obviously there exists $w_1 \in T^*$ so that $w = w_1 w'$. By the induction hypothesis we have $\alpha_1 \Rightarrow_G w'$. We distinguish two possible cases for the first computation step: Either there are $A \in N$ and $\beta \in (T \cup N)^*$ so that $w = w'$ (that is $w_1 = \varepsilon$), $\alpha = A\alpha'$, $\alpha_1 = \beta\alpha'$ and $A \to \beta \in P$ or $w = aw'$, $\alpha = a\alpha_1$. In the first case we have $\alpha \Rightarrow_G \alpha_1$ and as already established $\alpha_1 \Rightarrow_G w' = w$. So $\alpha \Rightarrow_G w$. In the second case we have $\alpha = a\alpha' \Rightarrow_G aw' = w$. So in either case we have the desired result. By applying this to $S \Rightarrow_G w$ we have $L(G) \subseteq N(M)$.

□

The formaliziation of this proof is in the Lean file `CFG_to_PDA.lean` and is split across multiple lemmas and definitions. The first step of course is given a grammer G to construct M. Than we have to prove the "obvious" properties of M before showing the main result with induction. The shown source code is slightly abbreviated.

```
structure Q where loop ::

abbrev S (G : ContextFreeGrammar T) [Fintype G.NT] := Symbol T G.NT

abbrev transition_fun (G : ContextFreeGrammar T) [Fintype G.NT] (_ : Q) (a : T) (Z : S G)
    : Set (Q × List (S G)) :=
  match Z with
  | terminal b => if a=b then {(Q.loop, [])} else ∅
  | _ => ∅

abbrev transition_fun' (G : ContextFreeGrammar T) [Fintype G.NT] (_ : Q) (Z : S G) : Set
    (Q × List (S G)) :=
  match Z with
  | nonterminal N => { (Q.loop, α) | (α : List (S G)) (_ : ⟨N, α⟩ ∈ G.rules) }
  | _ => ∅

abbrev M (G : ContextFreeGrammar T) [Fintype G.NT] : PDA Q T (S G):= {
  initial_state := Q.loop
  start_symbol := nonterminal G.initial
  transition_fun := transition_fun G
  transition_fun' := transition_fun' G
  finite := --
  finite' := --
```

For Q we need a set with one element which translates in Lean to a type with a single element. The type `Q` has a single construtor with no arguments, which results in exactly one term of type `Q` namely `loop : Q`. The set of stack symbols is a union of N and T, in Lean both `T` (T) and `G.NT` (N) are types not sets, so a union is not possible, the sum type `Symbol T G.NT` fills therefore the role of $N \cup T$. For brevity we call this type `S G`. If we look at the transition functions `transition_fun`, `transition_fun'` we see that reads only happen with a terminal symbol on the stack and ε-transitions only with a nonterminal on the stack (otherwise the set of possible next configurations is empty). The sets themselves are exactly as in the proof.

The automaton `M` is just the tuple of these components, and the two proofs that the sets returned by the transition functions are really finite. We will show the more interesting `finite'`:

```
finite' : ∀(q : Q)(Z : S): (transition_fun' q Z).Finite :=
    -- Introduce vars, split case on terminal, nonterminal
  rintro q (⟨x⟩|⟨N⟩)
  · exact Set.finite_empty            -- for terminals the empty set is returned
  · -- Build a large finite set and show that our set is a subset
    let R  := {r | r ∈ G.rules}
    have hR : R.Finite := by simp [R] -- G.rules is a Finset
    let S  := (λ ⟨N, α⟩ ↦ (Q.loop, α)) '' R      -- Our set has a different form
     -- Finitess is preserved under images
    have hS : S.Finite := by apply Set.Finite.image; exact hR
    -- The set we want prove to be finite
    let A := (transition_fun' G q (nonterminal N))
    have : A ⊆ S := by
      intro ⟨_, α⟩ h                      -- introduce the element of A
      dsimp [A, transition_fun'] at h    -- simplify the proof of (_, α) ∈ A
      obtain ⟨α', hr, he⟩ := h            -- seperate h in to two parts
      obtain ⟨_, hα⟩ := Prod.mk.inj he    -- α and α' are obviously equal
      rw [hα] at hr
      rw [Set.mem_image]                 -- we want to show that (_, α) ∈ S
      use ⟨N, α⟩                          -- Our candiate for the preimage
      simp [hr, R]                       -- simplifcation closes the goal
    exact Set.Finite.subset hS this
```

As the proof of Theorem 1 is split into multiple lemmas in Lean, I will illustrate how the "obvious facts" used in the traditional proof translate into Lean lemmas.

The base case of the first induction is realized in Lean via an application of following lemma (with `w':=α:=[]`):

```
theorem M_consumes_terminal_string  (w w': List T) (α : List (S G)):
    (M G).Reaches ⟨Q.loop, w++w', w.map terminal ++ α⟩ ⟨Q.loop, w', α⟩
```

Which states that `M` in fact consumes terminals as we intended, the proof boils down to an induction on the `List w` and simplification with the definition of `transition_fun`.

To formalize the induction step we make use of following lemmas:

```
theorem M_consumes_terminal_string  (w w': List T) (α : List (S G)):
    (M G).Reaches ⟨Q.loop, w++w', w.map terminal ++ α⟩ ⟨Q.loop, w', α⟩

theorem M_consumes_nonterminal {r : ContextFreeRule T G.NT} (h : r ∈ G.rules) (w : List T)
    (α : List (S G)):
    (M G).ReachesIn 1 ⟨Q.loop, w, nonterminal r.input :: α⟩ ⟨Q.loop, w, r.output ++ α⟩
```

```
theorem M_deterministic_of_terminal_stack (w v: List T) (β  : List (S G)):
    (M G).Reaches ⟨Q.loop, w, v.map terminal ++ β⟩ ⟨Q.loop, [], []⟩ →
    ∃ w' : List T,  w = v ++ w' ∧ (M G).Reaches ⟨Q.loop, w', β⟩ ⟨Q.loop, [], []⟩
```

The first one we have already seen, and the second one is very similar to the first one (but easier to prove as no induction is required). The third one however requires some work to obtain and its proof is split into two further lemmas. Equipped with these lemmas the formalization of the induction step as in the traditional proof is quite straightforward.

```
theorem M_reaches_off_G_derives (α : List (Symbol T G.NT)) (w : List T)
    (h : G.DerivesLeftmost α (w.map terminal)):
    (M G).Reaches ⟨Q.loop, w, α⟩ ⟨Q.loop, [], []⟩ := by
  induction' h using Relation.ReflTransGen.head_induction_on with α β hα _ ih
  case refl =>
    convert M_consumes_terminal_string w [] [] <;> simp
  case head =>
    obtain ⟨r,hrg,hrα⟩ := hα
    rw [rewrites_leftmost_iff] at hrα
    obtain ⟨p,q,hα',hβ'⟩ := hrα
    rw [hβ'] at ih
    rw [List.append_assoc] at ih
    apply M_deterministic_of_terminal_stack at ih
    obtain ⟨w', hw', hr ⟩ := ih
    have hpart₁ : (M G).Reaches ⟨Q.loop, w,α⟩ ⟨Q.loop, w', nonterminal r.input :: q ⟩ := by
      rw [hα', List.append_assoc, hw']
      apply M_consumes_terminal_string p  _
    have hpart₂ : (M G).Reaches ⟨Q.loop, w', nonterminal r.input :: q⟩ ⟨Q.loop, w',
    r.output ++ q⟩ := by
      rw [reaches_iff_reachesIn]
      use 1
      exact M_consumes_nonterminal hrg _ q
    have := Reaches.trans hpart₁ hpart₂
    exact Reaches.trans this hr
```

As opposed to the traditional proof, the formalized proof makes use of a structural induction principle `Relation.ReflTransGen.head_induction_on`. This is more idiomatic as Mathlib provides this principle of induction directly, for proofs actually using a induction on the number of deriviations steps a lot of custom code is necessary. Looking at the source code we see that without interactive feedback this proof is difficult to understand. It is included mainly to highlight that the three lemmas from before are really sufficient for the induction step and (as further inspection maybe reveals) that the induction step mirror the traditional proof very closely.

The other direction of the proof requires the following additional theorems:

```
theorem reachesIn_one_on_empty_stack {q p: Q}{w w': List T}{α : List S}:
    ¬pda.ReachesIn 1 ⟨q, w, []⟩ ⟨p, w', α⟩

theorem G_rule_of_M_consumes_nonterminal {w w': List T}{α β: List (S G)}{N : G.NT}  :
    (M G).ReachesIn 1 ⟨Q.loop, w, nonterminal N :: α⟩ ⟨Q.loop, w', β⟩ →
    ∃(γ : List (S G)), (⟨N,γ⟩ ∈ G.rules) ∧ β = γ ++ α ∧ w=w'
```

The first of the two is just to exclude the trivial case in the induction step, that is $\alpha = \varepsilon$. So that just the two cases discussed in the traditional proof remain. The second one is more interesting as it allows us to conclude from behavior of `M` (consumption of a nonterminal) the existence of a production rule in `G`. If we look back at the first half of the proof, all our conclusions where of the form:

$$\text{Knowledge of } \texttt{G} \Longrightarrow \text{Knowledge of } \texttt{M}$$

Which is more natural (and easier to prove) as we constructed `M` out of `G`. The base case is in Lean as clear as in the traditional proof. The main difference in the induction step is the nature of the case split, in Lean we perform an case split on the top most stack element (terminal, nonterminal, empty), show that the empty case is contradictory and proof in the remaining cases that the computation we described in the traditional proof is actually happening. This aside the induction step in Lean is again virtually identical to the traditional proof:

```
theorem G_derives_of_M_reaches {α : List (Symbol T G.NT)} {w : List T}
    (h: (M G).Reaches ⟨Q.loop,w,α⟩ ⟨Q.loop,[], []⟩):
    G.Derives α (w.map terminal) := by
  rw [reaches_iff_reachesIn] at h
  obtain ⟨n,hr⟩ := h
  induction' n  with n ih generalizing w α
  · apply reachesIn_zero at hr
    apply conf.mk.inj at hr
    simp [hr, Derives.refl]
  · rw [←reachesIn_iff_split_first] at hr
    obtain ⟨⟨_,w',β⟩, h₁, h₂⟩ :=  hr
    apply ih at h₂
    rcases α with _|⟨⟨a⟩|⟨N⟩,α'⟩
    · -- trivial case, stack is empty
      apply reachesIn_one_on_empty_stack at h₁
      contradiction
    · -- topmost stack symbol is terminal
      apply M_deterministic_step_of_terminal_stack_cons at h₁
      rw [h₁.1,h₁.2]
      convert ContextFreeGrammar.Derives.append_left h₂ [terminal a]
    · -- topmost stack symbol is nonterminal
      apply G_rule_of_M_consumes_nonterminal at h₁
      obtain ⟨γ,hr,hγ,hw⟩ := h₁
      rw [hw.symm] at h₂
      have : G.Derives ([nonterminal N] ++ α') β := by
        have : G.Derives [nonterminal N] γ := by
          apply Produces.single
          use ⟨N,γ⟩, hr
          convert ContextFreeRule.rewrites_of_exists_parts ⟨N,γ⟩ [] []
          simp
        convert ContextFreeGrammar.Derives.append_right this α'
      exact Derives.trans this h₂
```

Again the proof is difficult to read, but it should be noticeable that the structure is as described. The last case seems overly long, but this is just caused by piecing together a proof that $\alpha \Rightarrow_G \alpha_1 \Rightarrow_G w' = w$. Which is clear and easy to prove but somewhat lengthy.

Combining these two results we get:

```lean
theorem pda_of_cfg (G : ContextFreeGrammar T)[Fintype G.NT] : G.language  = (M
    G).acceptsByEmptyStack
```

With this we have shown that every language generated by a CFG can be recognized with a PDA, now we show the opposite.

4 PDA to CFG

Theorem 2. *Let M be a PDA with $L = N(M)$ then there exists a CFG G so that $L(G) = L$.*

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA, we define a grammar $G = (N, \Sigma, P, S)$ as follows: Let $n_0 = \max\{|\alpha| \mid \exists q, p \in Q,\ a \in \Sigma,\ Z \in \Gamma : (p, \alpha) \in \delta(q, a, Z)\} + 1$ we define the nonterminals

$$N = \{[p, Z, q] \mid p, q \in Q, Z \in \Gamma\} \cup \{\langle p, \alpha, q\rangle \mid p, q \in Q, \alpha \in \Gamma^*, |\alpha| \leq n_0\} \cup \{S\}$$

(note: $[p, Z, q] \neq \langle p, Z, q\rangle$) and following productions

$$\langle q, \varepsilon, q\rangle \to \varepsilon \quad \text{for every } q \in Q \tag{1}$$
$$[q, Z, p] \to a\langle q_1, \gamma, p\rangle \quad \text{if } (q_1, \gamma) \in \delta(q, a, Z) \tag{2}$$
$$\langle q, Z\alpha, p\rangle \to [q, Z, q_1]\langle q_1, \alpha, p\rangle \quad \text{for every } q_1 \in Q \tag{3}$$
$$S \to [q_0, Z_0, p] \quad \text{for every } p \in Q. \tag{4}$$

We will now proof $L(G) = N(M)$ by first showing

$$\forall \langle q, \gamma, p\rangle \in N : \langle q, \gamma, p\rangle \Rightarrow_G x \Leftrightarrow (q, x, \gamma) \vdash_M (p, \varepsilon, \varepsilon)$$

We begin with the reverse implication, so assume $(q, x, \gamma) \vdash_M (p, \varepsilon, \varepsilon)$ we proof $\langle q, \gamma, p\rangle \Rightarrow_G x$ by induction on the number of computation steps. For the base case we have $(q, x, \gamma) \vdash^0_M (p, \varepsilon, \varepsilon)$ so $\gamma = \varepsilon$, $x = \varepsilon$ and $q = p$, we know $\langle q, \varepsilon, q\rangle \to \varepsilon$ by (3.1). For the induction step we assume $(q, x, \gamma) \vdash^n_M (p, \varepsilon, \varepsilon)$ and assume the statement holds for natural numbers less than n. We further assume $\gamma = Z\gamma'$ as otherwise no computation would be possible and write $x = ay$, noting that $a = \varepsilon$ is possible.

We split at the first step of the computation:

$$(q, ay, Z\gamma') \vdash^1_M (q_1, y, \alpha\gamma') \vdash^{n-1}_M (p, \varepsilon, \varepsilon)$$

where $(q_1, \alpha) \in \delta(q, a, Z)$. From $(q_1, y, \alpha\gamma') \vdash^{n-1}_M (p, \varepsilon, \varepsilon)$ we obtain the existence of $\tilde{q} \in Q$, $y_1, y_2 \in \Sigma^*$, $m_1, m_2 < n$ so that $y = y_1 y_2$, $(q_1, y_1, \alpha) \vdash^{m_1}_M (\tilde{q}, \varepsilon, \varepsilon)$ and $(\tilde{q}, y_2, \gamma') \vdash^{m_2}_M (p, \varepsilon, \varepsilon)$. By applying the induction hypothesis twice we obtain $\langle q_1, \alpha, \tilde{q}\rangle \Rightarrow_G y_1$ and $\langle \tilde{q}, \gamma', p\rangle \Rightarrow_G y_2$. Here it should be noted $\langle q_1, \alpha, \tilde{q}\rangle$ and $\langle \tilde{q}, \gamma', p\rangle$ are really members of N, as α is a stack push and γ' is shorter than the stack push γ. By rules (3.3) and (3.4) we have $\langle q, \gamma, p\rangle = \langle q, Z\gamma', p\rangle \Rightarrow_G [q, Z, \tilde{q}]\langle \tilde{q}, \gamma', p\rangle \Rightarrow_G a\langle q_1, \alpha, \tilde{q}\rangle\langle \tilde{q}, \gamma', p\rangle \Rightarrow_G a\langle q, \alpha, \tilde{q}\rangle\langle \tilde{q}, \gamma', p\rangle$. Taking all this together gives us $\langle q, \gamma, p\rangle \Rightarrow_G ay_1y_2 = x$

For the other direction we use induction on the number of deriviation steps, for the base case we have $\langle p, \gamma, q\rangle \Rightarrow^0_G x$ as assumption, this however cannot be the case, so the base case is concluded.

For the induction step we assume $\langle p, \gamma, q\rangle \Rightarrow_G^n x$ and assume further that the implication holds for natural numbers less than n. We differentiate if $\gamma = \varepsilon$ or $\gamma = Z\gamma'$. In the first case we immediately see $q = p$ and $x = \varepsilon$ and conclude $(q, \varepsilon, \varepsilon) \vdash_M (q, \varepsilon, \varepsilon)$. We now consider the case $\gamma = Z\gamma'$. As we are working with leftmost derivations we know the first two steps of the derivation:

$$\langle p, \gamma, q\rangle = \langle p, Z\gamma', q\rangle \Rightarrow_G^1 [p, Z, \tilde{q}]\langle \tilde{q}, \gamma', q\rangle \Rightarrow_G^1 a\langle p_1, \alpha, \tilde{q}\rangle\langle \tilde{q}, \gamma', q\rangle \Rightarrow_G^{n-2} x$$

Where $p_1 \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ and $(p_1, \alpha) \in \delta(p, a, Z)$. We can now split $x = ax_1x_2$ so that

$$\langle p, \alpha, \tilde{q}\rangle \Rightarrow_G x_1 \qquad \langle \tilde{q}, \gamma', p\rangle \Rightarrow_G x_2$$

in fewer than n steps. By applying the induction hypothesis we have $(p_1, x_1, \alpha) \vdash_M (\tilde{q}, \varepsilon, \varepsilon)$ and $(\tilde{q}, x_2, \gamma') \vdash_M (p, \varepsilon, \varepsilon)$. By putting this two computations together we obtain $(p_1, x_1x_2, \alpha\gamma') \vdash_M (q, \varepsilon, \varepsilon)$. Together with $(p, x, \gamma) = (p, ax_1x_2, Z\gamma') \vdash_M (p_1, x_1x_2, \alpha\gamma')$ we show the result. □

The formalization again begins by defining the construction and and proving some technicalities. Compared to Theorem 1 this construction is more involved and quite long (in fact nearly as long as the whole formalization of Theorem 1). We will know elaborate where this complexity comes from:

To define a CFG `G` we need a type of nonterminals, terminals and a `Finset` of rules. The type `Finset` is one of the possible ways to represent finite sets in Lean. The other approach is to use the ordinary `Set` type and the `Set.Finite` predicate.

These two approaches differ significantly (`Finset` is more akin to a data structure than to a mathematical set), but it is quite straightforward to switch between them (Mathlib `ContextFreeGrammar` uses `Finset` while this formaliziation uses `Set.Finite`).

Note that there is no requirement that the type of nonterminals is finite. This makes sense, as if the set of rules is finite there are only a finite number of nonterminals which can be reached from the start symbol anyway. We will later make use of this fact.

However if we look at the proof of Theorem 2 we see that

$$N = \{[p, Z, q] \mid p, q \in Q, Z \in \Gamma\} \cup \{\langle p, \alpha, q\rangle \mid p, q \in Q, \alpha \in \Gamma^*, |\alpha| \leq n_0\} \cup \{S\}$$

with

$$n_0 = \max\{|\alpha| \mid \exists q, p \in Q,\ a \in \Sigma,\ Z \in \Gamma : (p, \alpha) \in \delta(q, a, Z)\} + 1$$

is a finite set and that P is only finite because N is finite. If we would drop the restriction $|\alpha| \leq n_0$ in the definition of N the proof would still work, but the resulting grammar would not be a CFG. The implementation of N without this restriction is obvious:

```
inductive N (M: PDA Q T S)  where
  | start : N M
  | single : Q → S → Q → N M
  | list : Q → List S → Q → N M
```

It is however less obvious how to add the restriction $|\alpha| \leq n_0$ which guarantees finiteness. If we would modify the definition of `N` like that:

```
inductive N (M: PDA Q T S)  where
  | start : N M
  | single : Q → S → Q → N M
  | list : Q → (α : List S) → α.length ≤ n₀ → Q → N M
```

It would more accurately reflect the set N. But any function with codomain `N`, would always be required to implement a proof that α`.length` $\leq$ n_0 holds. Which is quite cumbersome, furthermore it would still not be obvious how to prove the type `N` to be finite. To avoid these problems, the type `N` is constructed as before, without any restriction on the length of the `list` constructor. This results in an infinite type. The production rules are however still required to be finite, this is solved by the construction of a finite set of nonterminals (`AllowedNonterminals`) which is then used to build a finite set of production rules. So while the set of all nonterminals is infinite only a finite number of them is used to construct production rules, the set of them is therefore finite.

```
abbrev AllStackPushes (M : PDA Q T S) : Set (List S) :=
  (Prod.snd '' ⋃(q : Q)(a : T)(Z : S), M.transition_fun q a Z) ∪
  Prod.snd '' ⋃(q : Q)(Z : S), M.transition_fun' q Z

theorem allStackPushes_finite (M : PDA Q T S) : (AllStackPushes M).Finite := --

abbrev AllStackPushes' (M : PDA Q T S): Finset (List S) :=
  (allStackPushes_finite M).toFinset

abbrev max_push (M : PDA Q T S)  := max ((AllStackPushes' M).image List.length).max 1

abbrev N.IsAllowed: N M → Prop
  | N.start => True
  | N.single _ _ _ => True
  | N.list _ α _ => α.length ≤ max_push M

abbrev AllowedNonterminals : Set (N M) := {n : N M | n.IsAllowed}
```

Looking at the code we see that n_0 from before is now called `max_push`, which is obtained by collecting all possible stack pushes in a set and than taking the maximum of its image under the `List.length` function. Note the different notation for image in `AllStackPushes` and `max_push`, this occurs because one is the image of a `Set` the other of a `Finset`. A `Finset` occurs here because taking the maximum (as all operations only defined for finite sets) requires a `Finset`. The following obvious seeming result is then proven:

```
theorem allowedNonterminals_finite : (AllowedNonterminals : Set (N M)).Finite
```

and for completeness I show following somewhat technical results:

```
theorem push_le_max_push (α : List S)(q : Q)(Z : S)(a : T)
    (h : α ∈ Prod.snd '' M.transition_fun q a Z): α.length ≤ max_push M

theorem push_le_max_push' (α : List S)(q : Q)(Z : S)
    (h : α ∈ Prod.snd '' M.transition_fun' q Z): α.length ≤ max_push M
```

Now we can definite the set of production rules:

```
abbrev epsilon_rule (q : Q): Set (ContextFreeRule T (N M)) := {⟨N.list q [] q ,[]⟩}

abbrev compute_rule (q p: Q) (a : T) (Z : S) : Set (ContextFreeRule T (N M)) :=
  (λ ⟨q₁,α⟩ ↦ ⟨N.single q Z p, [terminal a, nonterminal (N.list q₁ α p)]⟩) ''
    M.transition_fun q a Z

abbrev compute_rule' (q p: Q) (Z : S) : Set (ContextFreeRule T (N M)) :=
  (λ ⟨q₁,α⟩ ↦ ⟨N.single q Z p, [nonterminal (N.list q₁ α p)]⟩) '' M.transition_fun' q Z

abbrev split_rule (q₁:Q) :(n : N M) → Set (ContextFreeRule T (N M))
  | N.start => ∅
  | N.single _ _ _=> ∅
  | N.list _ [] _ => ∅
  | N.list q (Z::α)  p =>
    {⟨N.list q (Z::α) p, [nonterminal (N.single q Z q₁),nonterminal (N.list q₁ α p)]⟩}

abbrev start_rule (q: Q): Set (ContextFreeRule T (N M)) :=
  {⟨N.start, [nonterminal (N.list (M.initial_state) [M.start_symbol] q)]⟩}

abbrev RuleSet : Set (ContextFreeRule T (N M)) :=
  (⋃q:Q,  epsilon_rule q) ∪ (⋃(q:Q)(p:Q)(a:T)(Z:S), compute_rule q p a Z)
  ∪ (⋃(q:Q)(p:Q)(Z:S), compute_rule' q p Z) ∪ (⋃(q:Q)(n ∈ AllowedNonterminals),
    split_rule q n)
  ∪ (⋃(q:Q), start_rule q)

theorem ruleSet_finite : (RuleSet : Set (ContextFreeRule T (N M))).Finite := --

abbrev rules : Finset (ContextFreeRule T (N M)) := ruleSet_finite.toFinset
```

Looking at the definition of `RuleSet`, we see there a five types of rules, as in the construction in the traditional proof. If we look at the union containing `split_rule` we see that the index `n` is bounded by `AllowedNonterminals` thus also making `RuleSet` finite.

And finally we can define the grammar:

```
abbrev G (M : PDA Q T S) : ContextFreeGrammar T := {
  NT := N M
  initial := N.start
  rules := rules
}
```

Before proving the first implication we need following easy to prove facts about `G`:

```
theorem produces_epsilon (q : Q) :(G M).Produces [nonterminal (N.list q [] q)] (List.map
    terminal [])

theorem produces_split (q q₁ p : Q){α : List S}{Z : S}(h : (Z :: α).length ≤ max_push M ):
    (G M).Produces [nonterminal (N.list q (Z :: α) p)]
    [nonterminal (N.single q Z q₁), nonterminal (N.list q₁ α p)]

theorem produces_compute {q q₁ p : Q}{α : List S}{a : T}{Z : S}
```

```
    (h : (q₁, α) ∈ M.transition_fun q a Z) :
    (G M).Produces [nonterminal (N.single q Z p)] [terminal a, nonterminal (N.list q₁ α p)]

theorem produces_compute' {q q₁ p : Q}{α : List S}{Z : S}
    (h : (q₁, α) ∈ M.transition_fun' q Z) :
    (G M).Produces [nonterminal (N.single q Z p)] [nonterminal (N.list q₁ α p)]
```

Which just state that G realizes our intended deriviations. And following characterization of $\texttt{Reaches}_1$.

```
theorem reaches₁_push {q : Q}{x : List T}{Z : S}{γ : List S}{c : pda.conf}
    (h : pda.Reaches₁ ⟨q, x, Z::γ⟩ c) :
    (∃(a : T)(y : List T)(p : Q)(α : List S), x = a::y ∧ c = ⟨p, y, α ++ γ⟩ ∧
    (p, α) ∈ pda.transition_fun q a Z) ∨
    (∃(p : Q)(α : List S), c = ⟨p, x, α ++ γ⟩ ∧  (p, α) ∈ pda.transition_fun' q Z)
```

Here we note, that while in the traditional proof we do not a require a case distinction on wether a read happens, this is necessary in the formalization. As we can clearly see in the disjunction in $\texttt{reaches}_1\texttt{\_push}$. We note that in the formalization of theorem 1 no lemma like $\texttt{reaches}_1\texttt{\_push}$ was necessary as the automaton there was constructed by us, and so we were able to prove more specific lemmas about its behavior. The last puzzle piece is following lemma.

```
theorem split_stack {n : ℕ}{q p : Q}{x : List T}{α β : List S}
    (h : pda.ReachesIn n ⟨q, x, α ++ β⟩ ⟨p, [], []⟩):
    ∃(q₁ : Q)(m₁ m₂ : ℕ)(y₁ y₂ : List T), x=y₁++y₂ ∧ m₁ ≤ n ∧ m₂ ≤ n ∧
    pda.ReachesIn m₁ ⟨q, y₁, α⟩ ⟨q₁, [], []⟩ ∧ pda.ReachesIn m₂ ⟨q₁, y₂, β⟩ ⟨p, [], []⟩
```

It formalizes that if a certain configuration results in a successful computation, we can split the stack at an arbitrary point and obtain two separate successful computations. One for each part of the stack.

Because of the case split, the way grammars work and the required book keeping for `AllowedNonterminals` the proof turns out rather lengthy. I will show a shortened version.

```
theorem derives_of_reachesIn {γ : List S}{q p : Q}{x : List T}{n : ℕ}
    (hγ : γ.length ≤ max_push M) (h : M.ReachesIn n ⟨q,x,γ⟩ ⟨p,[],[]⟩) :
    (G M).Derives [nonterminal (N.list q γ p)] (x.map terminal) := by
  induction' n using Nat.strong_induction_on with n ih generalizing x γ p q
  rcases n with _ | ⟨n⟩
  · apply reachesIn_zero at h
    injection h with h₁ h₂ h₃
    rw [h₁,h₂,h₃]
    apply Produces.single
    exact produces_epsilon _
  · rcases γ with _ | ⟨Z, γ⟩
    · obtain ⟨_, h, _⟩ := reachesIn_iff_split_first.mpr h
      apply reachesIn_one_on_empty_stack at h
      contradiction
    · obtain ⟨⟨q₀, x, γ'⟩, h₁, h₂⟩ := reachesIn_iff_split_first.mpr h
      rw [←reaches₁_iff_reachesIn_one] at h₁
      rcases reaches₁_push h₁ with ⟨a, y, q₁, α, rfl, hc, hα⟩ | ⟨q₁, α, hc, hα⟩
      · obtain ⟨rfl, rfl, rfl⟩ := conf.mk.inj hc
        obtain ⟨q₁, m₁, m₂, y₁, y₂, hy, hm₁, hm₂, h₂₁, h₂₂⟩ := split_stack h₂
```

```
    have hα_allowed : α.length ≤ max_push M  := --
    have hγ_allowed : γ.length ≤ max_push M  := --
    apply ih m₁ (Nat.lt_succ_of_le hm₁) hα_allowed at h₂₁
    apply ih m₂ (Nat.lt_succ_of_le hm₂) hγ_allowed at h₂₂
    convert calc
      (G M).Derives
        [nonterminal (N.list q (Z :: γ) p)]
        ([nonterminal (N.single q Z q₁)]++[nonterminal (N.list q₁ γ p)]) := --
      (G M).Derives _
        ([terminal a, nonterminal (N.list q₀ α q₁)] ++
        [nonterminal (N.list q₁ γ p)]) := --
      (G M).Derives _
         ([terminal a, nonterminal (N.list q₀ α q₁)]++ (List.map terminal y₂)) := --
      (G M).Derives _
        ([terminal a] ++ List.map terminal y₁++ (List.map terminal y₂)) :=--
    simp [hy]
  · --
```

Inspecting the code reveals that the induction base is exactly as in the traditional proof, that the trivial case of an empty stack in the induction step is easily discharged and that the application of `reaches₁_push` results in a case split. Only the case where a read occurs is included. Looking at this case we see that the split stack lemma is applied, splitting the stack at the most recent push. Again as in the traditional proof. The induction hypotheses is applied twice, and the rest is bookkeeping and gluing together derivations.

If we look at the traditional proof of the other implication, we see that it is very straightforward. We begin by assuming a derivation is happening and because of the way we constructed our grammar we know how it has to look. These facts are formalized as following the four lemmas in Lean:

```
theorem deriviation_empty {n : ℕ}{x : List T}{q p : Q}
    (h : (G M).DerivesLeftmostIn [nonterminal (N.list q [] p)] (List.map terminal x) n) :
    q = p ∧ x = []

theorem produces_cons {q p : Q}{Z : S} {γ : List S}
    {u : List (Symbol T (N M))} (h : (G M).ProducesLeftmost [nonterminal (N.list q (Z::γ)
    p)] u):
    ∃q₁:Q, u = [nonterminal (N.single q Z q₁), nonterminal (N.list q₁ γ p)]

theorem produces_single {q p : Q}{Z : S}
    {u v: List (Symbol T (N M))}
    (h : (G M).ProducesLeftmost ((nonterminal (N.single q Z p)) :: v) u) :
    (∃(α : List S)(q₀ : Q)(a : T), (q₀, α) ∈ M.transition_fun q a Z
      ∧ u = (terminal) a :: (nonterminal (N.list q₀ α p)) :: v) ∨
    (∃(α : List S)(q₀ : Q), (q₀, α) ∈ M.transition_fun' q Z
      ∧ u = (nonterminal (N.list q₀ α p)) :: v)
```

These lemmas are proved by an exhaustive case split. All of them have as hypotheses that a deriviation is happening and conclude its general form and in some cases also some information about the `PDA M`. If we look at `produces_single` we see again the case split from before.

```
 theorem reachesIn_of_derivesLeftmostIn {γ : List S}{q p : Q}{x : List T}{n : ℕ}
    (hγ : γ.length ≤ max_push M)
```

```
    (h : (G M).DerivesLeftmostIn [nonterminal (N.list q γ p)] (x.map terminal) n) :
    M.Reaches ⟨q, x, γ⟩ ⟨p, [], []⟩ := by
  induction' n using Nat.strong_induction_on with n ih generalizing x q p γ
  · rcases γ with _ | ⟨Z,γ'⟩
    · apply deriviation_empty at h
      simp only [h]
      rfl
    · rcases n with _ | ⟨n⟩
      · obtain h := h.zero -- contradictory case
        cases x <;> simp at h
      obtain ⟨u, h₁, h₂⟩ := h.head_of_succ
      obtain ⟨q₁, rfl⟩ := produces_cons h₁
      rcases n with _ | ⟨n⟩
      · have := h₂.zero  -- contradictory case
        cases x <;> simp at this
      obtain ⟨u, h₂₁, h₂₂⟩ := h₂.head_of_succ
      obtain ⟨α, q₀, a, hα, rfl⟩ | ⟨α, q₀, hα, rfl⟩ := produces_single h₂₁
      · obtain ⟨w, x', m₁, m₂, hm₁, hm₂, rfl, hw, hx'⟩ := derivesLeftmostIn_cons' h₂₂
        conv at hw => arg 2; change [a].map terminal; rfl
        obtain hw := hw.terminal
        rcases w with _ | ⟨a', w'⟩
        · simp at hw -- contradictory case
        obtain ⟨rfl, rfl⟩ : (a' = a  ∧ w' = []) := by simpa using hw
        obtain ⟨w₁, w₂, m₁', m₂', hm₁', hm₂', rfl, hw₁, hw₂⟩ := derivesLeftmostIn_cons' hx'
        have hα_allowed : α.length ≤ max_push M := --
        have hγ'_allowed : γ'.length ≤ max_push M := --
        have r₁ : M.Reaches ⟨q, a' :: (w₁ ++ w₂), Z :: γ'⟩ ⟨q₀, w₁ ++ w₂, α ++ γ'⟩ := by
          apply Relation.ReflTransGen.single
          simp [Reaches₁, step, hα]
        have r₂ := ih m₁' (by linarith) hα_allowed hw₁
        have r₃ := ih m₂' (by linarith) hγ'_allowed hw₂
        have r₂ := r₂.append_stack γ'
        rw [unconsumed_input w₂] at r₂
        apply Reaches.trans r₁
        apply Reaches.trans r₂
        exact r₃
      · --
```

In this proof we see the need for `DerivesLeftmostIn` as otherwise strong induction would not be possible. The case split in the traditional proof if γ is empty or not, is clearly visible. The destructuring of the deriviation uses quite a lot of work, even though the lemmas from before are already proven. Closer inspection also reveals the use of the `derivesLeftmostIn_cons'` lemma, which is necessary to split a deriviation resulting in a list of nonterminals into two separate deriviations. As is used in the traditional proof, to apply the induction hypothesis twice.

Now we can finally prove

```
theorem cfg_of_pda (M : PDA Q T S) : (G M).language  = M.acceptsByEmptyStack
```

which does require still a little effort, as our proofs until now did not even mention the start symbol of `G`. So each direction of this theorem, does a little preparation before calling the corresponding implications proven before. With this theorem the formalization is completed.

5 Conclusion

As the formalization is now presented completely, we will use this section to highlight the challenges and takeaways from the project. The traditional proof, if written in a very detailed manner (as it is the case in this documentation) plus the definition of PDA, CFG, $\vdash$ and $\Rightarrow$ takes about three full pages. Whereas the formalization in Lean takes about 2000 lines of code or if one would print the source code 37 printed pages. These numbers are just to give an impression how much longer a fully formalized proof is. A lot of this additional code are just "obvious" lemmas, which require work to be formalized, but do not really add complexity to the proof. There are however a few areas, where the formalization really required additional thought and work. The obvious example is the finiteness of the set of production rules in theorem 2. In a traditional style of proof the finiteness of the construction is obvious immediately. Whereas in the formalized variant much thought went into the design of the production rules, so that this proof would be somewhat straightforward.

A less obvious example is the usage of strong induction throughout the formalization. In the traditional proofs, induction was always performed on a number and strong and normal induction where used freely. In Lean, however, the standard method of proof is structural induction. If structural induction is used however, the induction hypothesis can only be applied to the next "smaller" object, not to arbitrary "smaller" objects. The proof of theorem 2 requires the application of the induction hypotheses on arbitrary smaller derivations and computations, something which adds significantly on complexity in Lean. As infrastructure (in the form of `ReachesIn`, `DerivesLeftmostIn` and accompanying lemmas) has to be programmed and proven in order to enable these strong inductions.

Another example of additional complexity which has not been mentioned until now, is that some proofs can not be formalized at all in Lean. A different proof of theorem 2, for example, uses a construction of a CFG, where some productions results not necessarily in two nonterminals but in a list of, more or less, arbitrary length. While this could *theoretically* be translated into Lean, it would add mountains of complexity. Because the `split_stack` theorem we already encountered, would than necessarily have to handle arbitrary numbers of stack splits, and the induction would need to bookeep all the families of stack splits and productions. The proof in this documentation is based on this difficult to formalize proof, but altered the construction significantly so it can be translated into Lean more cleanly.

So in conclusion: The formalized proof is significantly longer than its traditional counterpart. To some degree because of the additional complexity the formalization brings, and to some degree because of simple "legwork" which has to be performed, but has not impact on the proof or its complexity otherwise. Some of the additional complexity stems from the incompatibility of structural induction with some traditional proofs, while some of it stems just from the additional rigor that is required for a formal proof.