Strong Normalization for Well-Typed Parsing Expression Grammars

Guilherme Augusto Anício Drummond do Nascimento
Programa de Pós-Graduação em Ciência da Computação,
Universidade Federal de Ouro Preto
Ouro Preto, Brazil
guilherme.drummond@aluno.ufop.edu.br

Elton Máximo Cardoso
Programa de Pós-Graduação em Ciência da Computação,
Universidade Federal de Ouro Preto
Ouro Preto, Brazil
elton.cardoso@aluno.ufop.edu.br

Samuel da Silva Feitosa
Universidade Federal da Fronteira Sul
Chapecó, Brazil
samuel.feitosa@uffs.edu.br

Rodrigo Ribeiro
Programa de Pós-Graduação em Ciência da Computação,
Universidade Federal de Ouro Preto
Ouro Preto, Brazil
rodrigo.ribeiro@ufop.edu.br

Abstract

Parsing Expression Grammars (PEGs) are a deterministic formalism for describing syntax that has gained wide adoption in practice. A natural question is whether a PEG parser always terminates: could a grammar send a parser into an infinite loop? We show that this is not the case for well-typed PEGs, using a logical-relations argument inspired by the well-known strong normalization proof for the simply typed λ-calculus. The proof is mechanized in the Agda proof assistant.

Keywords

Parsing Expression Grammars, Strong Normalization, Logical Relations, Agda, Proof Mechanization

1 Introduction

Parsing Expression Grammars (PEGs), introduced by Ford [7], are a formalism for describing recursive-descent parsers in a purely declarative style. A PEG over an alphabet Σ with n nonterminals is a system of n mutually recursive rules, each defining a *parsing expression* built from terminals, sequences, ordered choices, Kleene stars, and not-predicates. The semantics is given by a deterministic relation $G \vdash e,\ s \Rightarrow r$, stating that expression e, applied to input string s in grammar G, yields result r: either a pair of the consumed prefix and the remaining input, or an explicit failure.

Like all recursive-descent formalisms, PEGs do not support left-recursive rules, and the body of a Kleene-star expression must consume input on every iteration; otherwise the star loops.[1] A PEG that succeeds or fails on every input is called **complete**. Deciding completeness is undecidable in general [7]; Ford therefore proposed a decidable *well-formedness* predicate defined by induction on the expression syntax, together with a fixpoint computation over all sub-expressions of the grammar. Notation $G \vdash e$ denotes that expression e is well-formed in grammar G.

[1] An expression e *succeeds* on input s in grammar G if $G \vdash e,\ r \Rightarrow$ and r is not a failure.

As observed by Ribeiro et al. [12], however, this approach has a subtle issue: Ford's sequence rule (CatW),

$$\frac{G \vdash e_1 \qquad e_1 \to 0 \Rightarrow G \vdash e_2}{G \vdash e_1\, e_2}\ \{\textsc{CatW}\}$$

requires e_2 to be well-formed *only* when e_1 is nullable ($e_1 \to 0$). Consequently, the expression $a(!\,b \,/\, c)^*$ passes the inductive check: since the terminal a is not nullable, the sub-expression $(!\,b \,/\, c)^*$ is never inspected by that rule. Yet on any input with prefix ax where $x \neq b$, the not-predicate $!\,b$ succeeds without consuming input, causing the star to loop indefinitely. In order to avoid such degenerate cases, Ford propose a fixpoint computation to build the set of all well-formed sub-expressions of a grammar. While correct, Ford's definition is not direct, since it demands an additional step to guarantee that the grammar will not loop on inputs. In response, Ribeiro et al. proposed a type system for PEGs [12] offering a more direct and compositional well-formedness criterion, and conjectured, without formal proof, that every well-typed PEG terminates on every input.

In this work we prove that conjecture. We use a logical-relations argument (Tait's method), inspired by the classical STLC strong-normalization proof [8]. The proof is fully mechanized in Agda [11] with the standard library [2].

More specifically, we make the following contributions:

- We define a **logical relation** $\mathcal{R}\ G\ \Gamma\ \tau\ e$ for PEGs, where G is the grammar, Γ a typing context, τ the type of e, and e a parsing expression. $\mathcal{R}$ is the conjunction of two invariants: *termination* (condition I: e yields a result on every input) and *progress* (condition II: if the type of e is non-nullable, every successful parse of e consumes at least one character). Condition II is essential to show that Kleene-star bodies cannot loop silently.
- We prove a **fundamental lemma**: given a hypothesis asserting that every rule satisfies the logical relation $\mathcal{R}$ (playing the role of a semantic environment), then every well-typed expression also satisfies $\mathcal{R}$. The proof is by structural induction on the typing derivation. A key difficulty absent in the STLC setting is that such hypothesis is circular: rules may refer to one another. We resolve this via lexicographic

well-founded induction on input length and non-terminals set cardinality.
- We derive **strong normalization**: every well-typed PEG terminates on every input. This follows immediately by applying the fundamental lemma with the lexicographically-constructed evidence for $\mathcal{R}$.

The rest of this paper is organized as follows. Section 2 introduces the Agda programming language and presents the syntax, semantics, and type system for PEGs. Section 2.3 reviews the strong normalization proof for STLC. The strong normalization argument for PEGs is detailed in Section 3. A discussion on the similarities and differences between our proof for PEGs and the well-known normalization argument for the STLC is presented in Section 4. Section 5 surveys related work, and Section 6 concludes.

2 Background

2.1 The Agda Programming Language

Agda is a dependently typed functional programming language and proof assistant [11]. Its type system is based on Martin-Löf intuitionistic type theory: types are first-class values and may depend on other values, allowing precise specifications to be expressed and enforced at compile time. We give a brief overview of the Agda features used throughout this paper; readers unfamiliar with Agda are referred to [15].

Data types. Agda data types are introduced with the `data` keyword using GADT-style syntax. As a running example, consider the Peano natural numbers and the finite type `Fin`:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Because the index `n` varies across constructors, `Fin` is an *indexed family*: `Fin n` has exactly n inhabitants. It is used throughout the formalization to represent nonterminal indices and De Bruijn term indices in a scope-safe way.

Propositions as types. Agda implements the Curry–Howard correspondence: propositions are types and proofs are terms. A value of type `t ≡ u` is evidence of propositional equality; its sole constructor `refl` witnesses reflexivity. A theorem is then a function from hypotheses to its conclusion, type-checked by the same checker as ordinary programs. For example, symmetry and transitivity of equality are proved by pattern matching on `refl`:

```
sym : ∀ {A : Set}{x y : A} → x ≡ y → y ≡ x
sym refl = refl

trans : ∀ {A : Set}{x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl q = q
```

Induction as recursion. In Agda, proof by induction is definitionally structural recursion: to prove a property of an inductive type it suffices to give one function clause per constructor with recursive calls on strict subterms. Agda's termination checker enforces this structural condition, which corresponds precisely to the well-foundedness required by induction. Every proof in this paper is an Agda function defined in this style. As a concrete example, the following proof shows that list concatenation distributes over `length`:

```
length-++ : ∀ {A : Set}(xs ys : List A)
   → length (xs ++ ys) ≡ length xs + length ys
length-++ [] ys = refl
length-++ (x ∷ xs) ys = cong suc (length-++ xs ys)
```

The base case holds definitionally; the inductive step applies the congruence combinator `cong` to the recursive call. This particular lemma is used in the fixpoint theorem to witness that a suffix of a non-empty parsed prefix is strictly shorter.

Records. Agda's `record` types bundle named fields under a single type. Fields are accessed with the usual dot notation. In this paper, the logical relation $\mathcal{R}$ is a record with two fields (termination and progress), making it straightforward to construct and project each condition independently. As illustration, consider how an existential type is encoded as a dependent record:

```
record ∃ {A : Set}(B : A → Set) : Set where
  constructor _,_
  field
    witness : A
    proof : B witness
```

The dependent field `proof` has type `B witness`, which refers to the value of the earlier field `witness`. This pattern is used throughout the proofs to pair evaluation witnesses with their correctness evidence.

With-abstraction. Agda's `with` keyword allows intermediate results to be brought into scope for further pattern matching, similar to a local case split. Writing `with` e_1 `|` e_2 scrutinises two expressions simultaneously, avoiding deeply nested branches. This construct is used extensively in the fundamental lemma to inspect results of recursive calls on sub-expressions. A simple illustration filters a list by a predicate:

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x ∷ xs) with p x
... | true  = x ∷ filter p xs
... | false = filter p xs
```

Each `...` branch inherits the full left-hand side and extends it with the result of scrutinising `p x`. Readers unfamiliar with Agda are referred to [11, 15].

2.2 An Overview Of PEGs

Syntax and semantics. Fix a finite alphabet Σ and $n \in \mathbb{N}$ nonterminals, identified with indices in $\{1, \ldots, n\}$. The syntax of a *parsing expression* is defined by the following context free grammar:

$$e \ ::= \ \varepsilon \mid a \mid A_i \mid e_1 \cdot e_2 \mid e_1 / e_2 \mid e^* \mid !e$$

A *PEG grammar* G consists of a rule G_i for each nonterminal i and a start expression G_s.

Each PEG construct has a deterministic semantics on a string input. The expression ε always succeeds consuming no input, while terminal a succeeds consuming a if the next character matches and fails otherwise. Nonterminal A_i behaves as the expression G_i defined for rule i. Sequence $e_1 \cdot e_2$ tries e_1 first; if it succeeds consuming

p_1 and leaving suffix s', then tries e_2 on s'; the sequence succeeds consuming $p_1 \cdot p_2$ if both succeed, and fails if either fails. Ordered choice e_1/e_2 tries e_1 first and returns its result if it succeeds; only if e_1 fails does it try e_2. Unlike CFG alternation, the choice is deterministic and committed. The Kleene star e^* applies e repeatedly as long as it succeeds, accumulating the consumed prefix; it always succeeds, possibly consuming nothing. Finally, the not-predicate $!e$ succeeds consuming *nothing* if e fails on the current input, and fails if e succeeds; it never consumes input regardless of outcome.

In Agda, nonterminals are `Fin n` indices and expressions form an indexed inductive type:

```
data PExp (n : ℕ) : Set where
  empty : PExp n
  term : Char → PExp n
  var : Fin n → PExp n
  seq : PExp n → PExp n → PExp n
  choice : PExp n → PExp n → PExp n
  star : PExp n → PExp n
  not : PExp n → PExp n
```

The semantics is given by a big-step relation $G \vdash e,\ s \Rightarrow r$, where a result r is either (p, s') (consumed prefix p, remaining suffix s') or $\bot$ (failure). In Agda, results are:

```
data ParseResult : Set where
  ok : Word → Word → ParseResult
  fail : ParseResult
```

The relation `Parses G e s r` has one constructor per semantic rule. The star rules, which drive the termination argument, are:

```
starBase : Parses G e s fail
  → Parses G (star e) s (ok [] s)

starStep : Parses G e s (ok p₁ s')
  → Parses G (star e) s' (ok p₂ s'')
  → Parses G (star e) s (ok (p₁ ++ p₂) s'')
```

Constructor `starBase` captures the base case: when e fails on s, the iteration e^* succeeds consuming the empty prefix and leaving s unchanged. Constructor `starStep` captures one iteration: e succeeds on s consuming prefix p_1 and leaving suffix s', and then e^* continues from s' consuming p_2 and leaving s''; the combined result concatenates both prefixes. Because the type system requires e to be non-nullable (nullability flag **f**), every application of `starStep` strictly shortens the remaining input: $|s'| < |s|$. This strict decrease is the first component of the lexicographic termination measure $(|s|, |\tau.F|)$.

A Type System for PEGs. Following Ribeiro et al. [12], we assign each parsing expression a type $\tau = (\eta, F)$ where η is a nullability bit and $F \subseteq \{1, \ldots, n\}$ is the *head set*: a subset of non-terminals which can appear at the leftmost position of a rule. Notations $\tau.\eta$ and $\tau.F$ denote the nullability and the head set fields for a type τ.

```
record Ty (n : ℕ) : Set where
  constructor mkTy
  field
    nullable : Bool
    first : Subset n
```

A typing context Γ assigns an assumed type to each nonterminal. The judgment $\Gamma \vdash e : \tau$ is:

$$\frac{}{\Gamma \vdash \varepsilon : (\mathbf{t}, \emptyset)} \qquad \frac{}{\Gamma \vdash a : (\mathbf{f}, \emptyset)}$$

$$\frac{}{\Gamma \vdash A_i : \big(\Gamma(i).\eta,\ \{i\} \cup \Gamma(i).F\big)}$$

$$\frac{\Gamma \vdash e_1 : (\eta_1, F_1) \quad \Gamma \vdash e_2 : (\eta_2, F_2)}{\Gamma \vdash e_1 \cdot e_2 : \big(\eta_1 \wedge \eta_2,\ F_1 \cup (\text{if } \eta_1 \text{ then } F_2 \text{ else } \emptyset)\big)}$$

$$\frac{\Gamma \vdash e_1 : (\eta_1, F_1) \quad \Gamma \vdash e_2 : (\eta_2, F_2)}{\Gamma \vdash e_1/e_2 : (\eta_1 \vee \eta_2,\ F_1 \cup F_2)} \qquad \frac{\Gamma \vdash e : (\mathbf{f}, F)}{\Gamma \vdash e^* : (\mathbf{t}, F)}$$

$$\frac{\Gamma \vdash e : (\eta, F)}{\Gamma \vdash !\,e : (\mathbf{t}, F)}$$

The nonterminal rule propagates $\{i\} \cup \Gamma(i).F$ unconditionally, so $\Gamma(i).F$ is the transitive closure of head-reachability. The star rule requires a non-nullable body.

Definition 2.1 (Well-typed grammar). *A grammar G is* well-typed *if there exists a context Γ satisfying: (1)* ***Fixpoint****: $\Gamma \vdash G_i : \Gamma(i)$ for every i; (2)* ***Acyclicity****: $i \notin \Gamma(i).F$ for every i.*

In Agda:

```
record WellTyped {n : ℕ} (G : Grammar n) : Set where
  field
    Γ : Ctx n
    hfix : ∀ i → HasType Γ (Grammar.rules G i) (Γ i)
    hacy : ∀ i → i ∉ first (Γ i)
    hstart : ∃ λ τ → HasType Γ (Grammar.start G) τ
```

Together, the fixpoint and acyclicity conditions prevent all left-recursive loops.

The nonterminal rule differs from Ribeiro et al. [12] in a way that is essential for the termination argument. In the original presentation, the rule for A_i is an axiom that looks up $\Gamma(i)$ and enforces $i \notin \Gamma(i).F$ inline; the type of A_i is simply $\Gamma(i)$. Here, the rule unconditionally produces $(\Gamma(i).\eta,\ \{i\} \cup \Gamma(i).F)$, making i explicit in the head set, while the acyclicity condition $i \notin \Gamma(i).F$ is factored out into the `WellTyped` record.

This reformulation is what enables the lexicographic measure $(|s|, |\tau.F|)$ used in the strong-normalization proof. When A_i is expanded, we pass from a type with head set $\{i\} \cup \Gamma(i).F$ to the rule body G_i with head set $\Gamma(i).F$. Because $i \notin \Gamma(i).F$ by acyclicity, the cardinality decreases by exactly one. In the original presentation, A_i and G_i carry the same head set, so this decrease is invisible.

2.3 Strong Normalization Via Logical Relations

In this section we review the logical-relations method for the STLC, since our proof for PEGs follows a similar argument. We start by reviewing the syntax and semantics for STLC. Then, we show the type system and the definitions necessary for proving strong normalization for STLC.

Syntax. Types are $\tau ::= \mathbf{Bool} \mid \sigma \Rightarrow \tau$. Terms use de Bruijn indices [5] to avoid naming issues:

$$t ::= \#i \mid \lambda_\sigma\, t \mid t_1\ t_2 \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$$

In Agda, type syntax is represented by the following type:

```
data Ty : Set where
  bool : Ty
  _⇒_  : Ty → Ty → Ty
```

and λ-calculus are encoded by the following inductive type:

```
data Term : Set where
  var : ℕ → Term
  lam : Ty → Term → Term
  app : Term → Term → Term
  tru : Term
  fls : Term
  ite : Term → Term → Term → Term
```

Operational semantics. We use a big-step, environment-based semantics: a runtime environment ρ is a list of values; values are either booleans or closures $\mathbf{clos}(\sigma, t, \rho')$.

```
data Val : Set where
  bool : Bool → Val
  clos : Ty → Term → List Val → Val
```

The judgment `Eval ρ t v` ("t evaluates to v under ρ") has seven constructors. The most important are:

```
eLam  : Eval ρ (lam τ t) (clos τ t ρ)
eApp  : Eval ρ t₁ (clos τ body ρ')
   → Eval ρ t₂ v₂
   → Eval (v₂ :: ρ') body v
   → Eval ρ (app t₁ t₂) v
eIteT : Eval ρ c (bool true)  → Eval ρ t v
   → Eval ρ (ite c t e) v
eIteF : Eval ρ c (bool false) → Eval ρ e v
   → Eval ρ (ite c t e) v
```

Constructor `eLam` is axiomatic: a lambda abstraction $\lambda_\sigma\, t$ evaluates immediately to the closure $\mathbf{clos}(\sigma, t, \rho)$, capturing the current environment ρ without evaluating the body. Constructor `eApp` evaluates $t_1\, t_2$ in three steps: t_1 must reduce to a closure $\mathbf{clos}(\sigma, \mathit{body}, \rho')$; the argument t_2 evaluates to some value v_2; and then the body is evaluated in the extended environment $v_2 :: \rho'$, yielding the result v. Constructors `eIteT` and `eIteF` handle the conditional: the guard c is evaluated first, and depending on whether it yields **true** or **false**, only the chosen branch is evaluated.

Typing. A typing context Γ is a list of types indexed by de Bruijn position. The judgment $\Gamma \vdash t : \tau$ is defined inductively:

```
htVar : Γ [ i ]:= τ
  → HasType Γ (var i) τ

htLam : HasType (σ :: Γ) t τ
  → HasType Γ (lam σ t) (σ ⇒ τ)

htApp : HasType Γ t₁ (σ ⇒ τ)
  → HasType Γ t₂ σ
  → HasType Γ (app t₁ t₂) τ

htTru : HasType Γ tru bool
htFls : HasType Γ fls bool

htIte : HasType Γ c bool
  → HasType Γ t τ → HasType Γ e τ
  → HasType Γ (ite c t e) τ
```

Constructor `htVar` looks up the type of a de Bruijn variable at position i in Γ. Constructor `htLam` types a lambda $\lambda_\sigma\, t$ by checking the body t in the extended context $\sigma :: \Gamma$, producing type $\sigma \Rightarrow \tau$. Constructor `htApp` requires the function to have an arrow type $\sigma \Rightarrow \tau$ and the argument to have the domain type σ. Constructors `htTru` and `htFls` are axioms for the boolean constants. Constructor `htIte` requires the guard to have type `bool` and both branches to share the same result type τ.

The Logical Relation for the STLC. A direct proof that every well-typed term terminates by induction on the typing derivation fails at the lambda case: to show that $\lambda_\sigma\, t$ terminates, one needs to know that the body t terminates for *every* well-typed argument it may receive, but those arguments are not part of the current derivation. Tait's method [8] resolves this by replacing the syntactic notion of "well-typed" with a stronger, semantic notion: a predicate $\mathrm{SemTy}_\tau(v)$ that asserts not only that v is a value of type τ, but that applying it to any semantically good argument still yields a semantically good result. For base type **Bool** this is just being a boolean value; for function types $\sigma \Rightarrow \tau$ it requires that the closure body, when evaluated in an environment extended with any SemTy_σ-argument, evaluates to a SemTy_τ-value. Because the relation is defined by recursion on the type (and not on the term), it sidesteps the circularity of the lambda case. The fundamental lemma then shows that every well-typed term evaluates to a value in the relation, from which termination is immediate.

Definition 2.2 (Semantic type). *The predicate* $\mathrm{SemTy}_\tau(v)$ *is defined by structural recursion on* τ*:*

$$\begin{aligned}
\mathrm{SemTy}_{\mathbf{Bool}}(v) &\triangleq \exists b,\ v = \mathbf{bool}\, b \\
\mathrm{SemTy}_{\sigma\Rightarrow\tau}(v) &\triangleq \exists t\, \rho,\ v = \mathbf{clos}(\sigma, t, \rho) \ \wedge \\
&\qquad \forall v_2,\ \mathrm{SemTy}_\sigma(v_2) \rightarrow \\
&\qquad\quad \exists v',\ (v_2 :: \rho \vdash t \Downarrow v') \\
&\qquad\qquad \wedge\ \mathrm{SemTy}_\tau(v')
\end{aligned}$$

The definition is well-founded because $\mathrm{SemTy}_{\sigma\Rightarrow\tau}$ refers only to strictly smaller types. In Agda, this becomes a function defined by pattern matching on `Ty`:

```
SemTy : Ty → Val → Set
SemTy bool v = ∃ λ b → v ≡ bool b
SemTy (σ ⇒ τ) v = ∃ λ body → ∃ λ ρ →
   v ≡ clos σ body ρ × (∀ v₂ → SemTy σ v₂
      → ∃ λ v' → Eval (v₂ :: ρ) body v' × SemTy τ v')
```

Definition 2.3 (Semantic environment). $\mathrm{SemEnv}(\Gamma, \rho)$ *holds when* $\mathrm{SemTy}_{\Gamma[i]}(\rho[i])$ *for every* i*:*

```
data SemEnv : Ctx → List Val → Set where
  snil  : SemEnv [] []
  scons : SemTy τ v → SemEnv Γ ρ → SemEnv (τ :: Γ) (v :: ρ)
```

Constructor `snil` witnesses that the empty runtime environment [] is semantically consistent with the empty typing context []. Constructor `scons` extends a semantic environment: given a proof that value v satisfies SemTy_τ and a proof that ρ is semantically consistent with Γ, it produces a proof that $v :: \rho$ is consistent with $\tau :: \Gamma$. In other words, `SemEnv` asserts that every value stored in the runtime environment is semantically good at the type its corresponding variable was assigned in the typing context.

The Fundamental Lemma and Main Theorem for the STLC.

Lemma 2.4 (Fundamental Lemma — STLC). *If* $\Gamma \vdash t : \tau$ *and* $\mathrm{SemEnv}(\Gamma, \rho)$*, then there exists* v *such that* $\rho \vdash t \Downarrow v$ *and* $\mathrm{SemTy}_\tau(v)$.

Proof sketch. By induction on the typing derivation, carrying a `SemEnv` witness for the current runtime environment. The `htVar` case retrieves the value's SemTy proof directly from the `SemEnv` witness. The `htLam` case produces a closure; the $\text{SemTy}_{\sigma\Rightarrow\tau}$ witness is a function that, given any v_2 satisfying SemTy_σ, extends the semantic environment with v_2 and recurses on the body. The `htApp` case applies the induction hypothesis to both sub-terms, extracts the closure from the function's result, and invokes its SemTy certificate on the semantically good argument, assembling the final `eApp` derivation. The boolean and conditional cases are straightforward. □

Theorem 2.5 (Strong Normalization — STLC). *If $\vdash t : \tau$, then there exists v such that $\vdash t \Downarrow v$.*

Proof sketch. Apply Lemma 2.4 with the empty semantic environment `snil`. The lemma yields v and an evaluation derivation $\vdash t \Downarrow v$; the SemTy component is discarded. Well-foundedness of SemTy follows from structural acyclicity of types: each recursive call targets a strictly smaller type, so no separate well-founded argument is needed. □

3 Strong Normalization For PEGs

This section presents the strong normalization proof for well-typed PEGs, organized into four subsections. Section 3.1 introduces the logical relation $\mathcal{R}$ and establishes its closure under each PEG operator; unlike the STLC relation, $\mathcal{R}$ must also enforce a *progress* condition to handle the Kleene-star operator. Section 3.2 states the Fundamental Lemma, which lifts $\mathcal{R}$ from individual operators to arbitrary well-typed expressions, given a hypothesis h_Γ that packages $\mathcal{R}$ for every grammar rule. Section 3.3 resolves the circularity in constructing h_Γ: a lexicographic well-founded induction on the pair $(|s|, |\tau.F|)$ builds h_Γ unconditionally for any well-typed grammar. Finally, these three components are assembled into the main Strong Normalization Theorem.

3.1 The Logical Relation for PEGs

The STLC argument carries over to PEGs, but two new challenges arise. First, the STLC relation SemTy is defined by recursion on the *type*, which is structurally smaller at every recursive call; for PEGs, types do not shrink when expanding a nonterminal, because the grammar rules can be mutually recursive. Second, the Kleene-star operator has no STLC counterpart: showing that e^* terminates requires knowing not only that each iteration of e terminates, but also that every successful iteration is non-empty; otherwise the star loops forever.

These two challenges motivate the two conditions of the relation $\mathcal{R}\ G\ \Gamma\ \tau\ e$. Condition (I) is the direct analogue of SemTy: it asserts that e terminates on every input. Condition (II), *progress*, strengthens (I) for non-nullable expressions: every successful parse must consume at least one character. This is what enables the Kleene-star case: each `starStep` application strictly reduces the remaining input, making the iteration well-founded. The circularity introduced by nonterminal expansion is handled not inside the relation itself, but in the construction of the hypothesis h_Γ, via a lexicographic well-founded induction on the pair $(|s|, |\tau.F|)$ (Section 3.3).

Definition 3.1 (Logical relation). *Let $G, \Gamma, \tau = (\eta, F)$, e be given. $\mathcal{R}\ G\ \Gamma\ \tau\ e$ is the conjunction of:*

(I) ***Termination:*** *for every s, $\exists r,\ G \vdash e,\ s \Rightarrow r$.*
(II) ***Progress:*** *if $\eta = \mathbf{f}$, then for every s, p, s', $G \vdash e,\ s \Rightarrow (p, s') \Rightarrow p \neq \varepsilon$.*

In Agda, $\mathcal{R}$ is a record type:

```
record R (G : Grammar n) (Γ : Ctx n) (τ : Ty n)
         (e : PExp n) : Set where
  field
    terminates : ∀ s → ∃ λ r → Parses G e s r
    progress : nullable τ ≡ false → ∀ s p s' →
         Parses G e s (ok p s') → p ≢ []
```

Condition (II) is the PEG analogue of the head-reduction invariant in STLC proofs: without it, the Kleene star case cannot be shown to terminate.

The $\mathcal{R}$ Closure Lemmas. We prove that $\mathcal{R}$ is closed under each PEG operator. The most important case is the star:

Lemma 3.2 ($\mathcal{R}$ for Kleene star). *Assume $\Gamma \vdash e : (\mathbf{f}, F)$ and $\mathcal{R}\ G\ \Gamma\ (\mathbf{f}, F)\ e$. Then $\mathcal{R}\ G\ \Gamma\ (\mathbf{t}, F)\ (e^*)$.*

Proof sketch. Condition (II) holds vacuously because the type of e^* has nullability flag $\mathbf{t}$. For condition (I), we use well-founded induction on the input length $|s|$. At each step, the `terminates` field of $\mathcal{R}\ e$ is invoked on the current input. If e fails, `starBase` closes the goal immediately. If e succeeds consuming prefix p and leaving suffix s', the `progress` field of $\mathcal{R}\ e$ supplies $p \neq \varepsilon$, which implies $|s'| < |s|$; the induction hypothesis then yields a derivation for e^* on s', and `starStep` assembles the combined result.

The remaining operators (ε, terminals, sequence, ordered choice, and not-predicate) are handled by straightforward structural arguments: termination follows from the hypotheses for their sub-expressions, and progress from the type-directed form of each rule. □

3.2 The Fundamental Lemma

The closure lemmas show that $\mathcal{R}$ is preserved by each PEG operator in isolation, but they say nothing about complete grammars. The fundamental lemma bridges the gap: given a hypothesis h_Γ asserting that every rule G_i already satisfies $\mathcal{R}$, every well-typed expression also satisfies $\mathcal{R}$. This mirrors the role of SemEnv in the STLC proof: just as SemEnv packages semantic goodness for every variable in scope, h_Γ packages $\mathcal{R}$ for every nonterminal in the grammar, allowing the structural induction to proceed without encountering unsupported nonterminal calls. The key challenge absent in the STLC setting is that h_Γ is inherently circular: rules can refer to each other, so h_Γ cannot be constructed before the lemma is proved. The fixpoint theorem (Section 3.3) resolves this by constructing h_Γ via a separate well-founded induction.

Lemma 3.3 (Fundamental Lemma). *Let G be a grammar, Γ a context and $h_\Gamma : \forall i,\ \mathcal{R}\ G\ \Gamma\ (\Gamma(i))\ (G_i)$. If $\Gamma \vdash e : \tau$ then $\mathcal{R}\ G\ \Gamma\ \tau\ e$.*

Proof sketch. By structural induction on the typing derivation $\Gamma \vdash e : \tau$. At each case, the corresponding closure lemma is applied: for ε, terminals, sequence, ordered choice, not-predicate,

and Kleene star, the closure lemmas from Section 3.1 supply $\mathcal{R}$ from the induction hypotheses on sub-expressions. For a nonterminal A_i, no sub-expression induction is needed: the hypothesis $h_\Gamma(i)$ directly provides $\mathcal{R}\ G\ \Gamma\ (\Gamma(i))\ (G_i)$, and the nonterminal closure lemma lifts this to $\mathcal{R}\ G\ \Gamma\ \tau\ (A_i)$. □

This lemma directly analogizes the STLC fundamental lemma: h_Γ plays the role of SemEnv. The difficulty, absent in the STLC case, is establishing h_Γ without circularity.

3.3 The Fixpoint Theorem

The fundamental lemma (Lemma 3.3) assumes a hypothesis h_Γ that asserts $\mathcal{R}$ for every grammar rule, but provides no means of constructing it. In the STLC proof this is not an issue: SemEnv is built incrementally, and well-foundedness of SemTy follows from structural recursion on the type. For PEGs the situation is more subtle: h_Γ must assert $\mathcal{R}$ for *all* rules simultaneously, and those rules may refer to one another in a mutually recursive fashion. Attempting to build h_Γ rule by rule would require knowing $\mathcal{R}$ for the other rules first, creating a circular dependency with no obvious starting point.

The fixpoint theorem resolves this circularity by constructing h_Γ directly through a **lexicographic well-founded induction**. The key insight is that circularity only appears to be circular: when rule G_i expands nonterminal A_j, the type of A_j has a strictly larger head-set cardinality than the type of G_j (by acyclicity, as shown in Section 2). So the dependency is not circular but rather strictly decreasing in the second component of the measure. At the same time, parsing a non-nullable sub-expression within a sequence or star strictly reduces the input length. Together, these two decreases define the lexicographic measure $(|s|,\ |\tau.F|)$ on which the induction proceeds.

A simpler approach of inducting on $|\Gamma(i).F|$ alone would provide $\mathcal{R}$ for nonterminals with smaller rank, but only for specific inputs. The fundamental lemma requires $\mathcal{R}$ quantified over *all* inputs, making a single-dimensional induction insufficient. Inlining the fundamental lemma into the induction and ranging over both dimensions simultaneously is what makes the argument go through.

The key lemma. We inline the fundamental lemma and induct over typing derivations using $(|s|, |\tau.F|)$ as a lexicographic measure.

Lemma 3.4 (Lexicographic key lemma). *Let G, Γ satisfy the fixpoint and acyclicity conditions. Then for every ℓ, f, every s with $|s| \le \ell$, and every e with $\Gamma \vdash e : \tau$ and $|\tau.F| \le f$:*

$$\exists r,\ G \vdash e,\ s \Rightarrow r$$
$$\wedge\quad \tau.\eta = \mathbf{f} \ \rightarrow\ \forall p\, s',\ G \vdash e,\ s \Rightarrow (p, s') \rightarrow p \neq \varepsilon.$$

Proof sketch. By strong induction on ℓ (outer) and then on f (inner), followed by structural induction on the typing derivation.

- *Nonterminal A_i*: The head set of A_i is $\{i\} \cup \Gamma(i).F$. By acyclicity, $i \notin \Gamma(i).F$, so $|\Gamma(i).F| < |\{i\} \cup \Gamma(i).F| \le f$. We apply the inner IH at strictly smaller cardinality to the fixpoint hypothesis $\Gamma \vdash G_i : \Gamma(i)$.
- *Sequence $e_1 \cdot e_2$*: The structural IH gives termination of e_1 on s. If e_1 consumes $p_1 \neq \varepsilon$, the suffix s' satisfies $|s'| < |s|$; we recurse on e_2 via the outer IH.
- *Kleene star e^**: Structural IH gives termination and progress for e; progress forces $p \neq \varepsilon$, hence $|s'| < |s|$; recurse on e^*.

□

Now we have all the ingredients for proving the fixpoint theorem.

Theorem 3.5 (Fixpoint). *If G is well-typed with witness Γ, then $\forall\, i,\ \mathcal{R}\ G\ \Gamma\ (\Gamma(i))\ (G_i)$.*

Proof sketch. Instantiate Lemma 3.4 for each rule G_i: set $\ell = |s|$ and $f = |\Gamma(i).F|$. The well-typed grammar hypothesis supplies the fixpoint condition $\Gamma \vdash G_i : \Gamma(i)$ and the acyclicity condition $i \notin \Gamma(i).F$, which are the two premises required by the lexicographic key lemma. The two conclusions of that lemma are exactly conditions (I) and (II) of $\mathcal{R}$. □

3.4 Strong Normalization Theorem For PEGs

The three preceding results now compose into the main theorem. The closure lemmas establish that $\mathcal{R}$ is preserved by each PEG operator. The fundamental lemma (Lemma 3.3) lifts this to arbitrary well-typed expressions, provided the hypothesis h_Γ is available. The fixpoint theorem (Theorem 3.5) constructs h_Γ unconditionally for any well-typed grammar, resolving the circularity via the lexicographic key lemma. Combining these three steps yields the strong normalization theorem: every well-typed parsing expression terminates on every input.

Theorem 3.6 (Strong Normalization). *If G is well-typed in a context Γ and $\Gamma \vdash e : \tau$, then for every s there exists r with $G \vdash e,\ s \Rightarrow r$.*

Proof sketch. Apply Theorem 3.5 to obtain h_Γ, then apply the Fundamental Lemma (Lemma 3.3) to $\Gamma \vdash e : \tau$ and h_Γ. The resulting $\mathcal{R}\ G\ \Gamma\ \tau\ e$ provides condition (I), which is the required termination statement. □

Corollary 3.7. *Every well-typed grammar terminates on every input.*

Proof sketch. Apply Theorem 3.6 to the start expression of the grammar. □

4 Discussion

Both proofs share the same high-level architecture: define a logical relation indexed by the type, prove a fundamental lemma by induction on the typing derivation, and derive strong normalization as a corollary. The differences arise from two structural properties that hold for the STLC but fail for PEGs.

The first difference concerns the *logical relation* itself. For the STLC, SemTy_τ is defined by structural recursion on τ: at function type $\sigma \Rightarrow \tau$, the recursive calls target strictly smaller types, so well-foundedness is free. For PEGs, types do not shrink under nonterminal expansion, since rules can refer to one another. The relation $\mathcal{R}$ sidesteps this by capturing only two input-output properties (conditions I and II) rather than encoding the full semantic behavior; the well-foundedness argument is deferred to the fixpoint theorem.

The second difference concerns *circularity*. In the STLC proof, the semantic environment SemEnv is built incrementally: each variable is added to the context one at a time, and there are no cyclic dependencies between variables. For PEGs, the hypothesis h_Γ must cover all grammar rules simultaneously, and those rules may be mutually recursive. The resolution is the lexicographic induction on $(|s|, |\tau.F|)$: expanding nonterminal A_i decreases $|\tau.F|$ by acyclicity, and processing the suffix after a non-nullable sub-expression decreases $|s|$. Neither decrease alone suffices; together they define a well-founded order that covers every case.

A third, smaller difference is the *progress* condition (II). The STLC proof never needs to track how much input a term consumes, because evaluation always terminates by structural recursion on the term. For PEGs, the Kleene star has no such structural guarantee: without knowing that each iteration is non-empty, the termination argument for the star breaks down. Condition (II) of $\mathcal{R}$ supplies this extra invariant and is propagated through all closure lemmas that produce non-nullable types.

5 Related Work

Ford [7] introduced PEGs and established termination through a syntactic well-formedness condition. Ribeiro et al. [12] gave a type-theoretic account; their termination argument is informal. Our work provides the missing formal proof.

Koprowski and Binsztok [9] present TRX, a PEG parser interpreter formally developed in Coq. Their semantics is step-indexed (derivations carry a step count), and they prove determinism, soundness, and completeness of parsing with respect to the PEG semantics. They further extend PEGs with typed semantic actions (XPEGs), yielding certified parsers that compute structured results. Termination is guaranteed by a syntactic check for the absence of indirect left-recursion, a variant of Ford's syntactic well-formedness condition. Certified parsers are extracted from the Coq development via code extraction and validated on XML and Java grammars. The primary contrasts with our work are: (1) TRX targets soundness and completeness of parsing, while we target strong normalization; (2) termination in TRX relies on a syntactic left-recursion check, while we use a type system whose head-set types admit grammars that syntactic checks reject; (3) TRX does not use logical relations.

Medeiros and Olarte [6] present a rewriting-logic semantics for PEGs in Maude, formalizing both local and global cuts as unified backtracking-control mechanisms. Their framework is executable and serves as both a parser and a tool for reasoning about parsing behavior. The primary contrast with our work is the proof technique and objective: they target semantic adequacy and cut formalization in rewriting logic, while we target strong normalization via logical relations in Agda.

A type-inference algorithm for the type system of Ribeiro et al. [12] was proposed in [3]. The approach translates typing constraints generated from a grammar into SMT-LIB formulas and uses the Z3 solver to infer types for all non-terminals automatically, removing the need for manual annotations. The algorithm is shown to be sound and complete with respect to the type system. This work addresses the *usability* of the type system, whereas our work addresses its *soundness*: we provide the formal termination proof that the type system implicitly relies on.

Daher et al. [4] formalize Pest, a prominent PEG-based parser generator for Rust that extends the standard PEG formalism with new repetition operators and an implicit stack. They give a formal semantics for these constructs and extend the type-inference algorithm of [3] to cover them, preserving the guarantee that well-typed grammars do not loop. This work is complementary to ours: they extend the type system to new operators and automate type inference, while we provide the logical-relations proof that well-typed grammars are strongly normalizing.

Krishnaswami and Yallop [10] give a typed, algebraic account of parsing using μ-regular expressions, an algebraic representation of context-free languages. Their type system restricts to LL(1) grammars and guarantees linear-time parsing through a denotational semantics and parser combinator library. The primary contrast with our work is the target formalism: μ-regular expressions admit only LL(1) context-free languages, while well-typed PEGs cover a broader class including ordered choice and the not-predicate. In the LL(1) setting no logical-relations argument is needed because the denotational semantics is directly well-founded; in our setting the mutual recursion between non-terminals requires the lexicographic measure.

Shankar and Lucas [13] formalize a chart parsing approach to PEG parsing in PVS. The parser is modeled as a state machine operating on a scaffold data structure indexed by input position and non-terminal; termination is established via a lexicographic measure on the number of unfinalized scaffold cells. Unlike our approach, they admit possibly left-recursive (ill-formed) PEGs and handle loops dynamically at parse time, also extracting a proof-of-parse representation from the final scaffold. The primary contrast is methodological: their dynamic, chart-based approach in PVS versus our static, type-based approach in Agda.

Logical-relations proofs for typed λ-calculi are classical [8, 14]. Wadler et al. [15] give a pedagogical Agda formalization via small-step semantics with substitution. Our proof (Section 2.3) uses big-step environment-based semantics, avoiding substitution at the cost of evaluating both branches of a conditional in the fundamental lemma.

Abel et al. [1] propose POPLMark Reloaded, a benchmark for comparing proof assistants on mechanizing logical-relations proofs. The benchmark establishes strong normalization of STLC extended with disjoint sums via Kripke-style logical relations, with solutions in Beluga, Coq, and Agda. The Agda solution uses intrinsically-typed de Bruijn terms and type-directed small-step reductions with explicit substitutions. By contrast, our STLC development uses a simpler, non-Kripke logical relation with big-step environment semantics, which avoids the substitution lemmas their mechanization requires. Our STLC proof serves as a pedagogical stepping stone toward the PEG normalization argument rather than as a standalone benchmark.

6 Conclusion

We have presented a mechanized proof of strong normalization for well-typed Parsing Expression Grammars in Agda. The proof employs the logical-relations methodology, introducing a relation $\mathcal{R}$ that captures both termination and a progress invariant. The key technical contribution is the lexicographic key lemma (Lemma 3.4),

which resolves the circularity introduced by mutual rule recursion through a lexicographic well-founded induction over $(|s|, |\tau.F|)$.

Future work includes formalizing the type inference algorithm for PEGs proposed by [3], extending the result to PEG variants with attributes and defining a definitional interpreter for typed PEGs by embedding the measure used by Lemma 3.4 into the intrinsically-typed representation of PEGs.

Artifact Availability

The Agda source code for the formalization is available at the following repository:

https://github.com/lives-group/peg-strong-normalization

and it type-checks in Agda 2.7.0.1 with standard library 2.2.

Use of Generative AI

Generative AI technologies (specifically Gemini) were used in the preparation of this work for grammatical review. The authors reviewed all AI-generated suggestions and take full responsibility for the content and validity of the final text.

References

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2018. POPLMark Reloaded: Mechanizing Proofs by Logical Relations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. Association for Computing Machinery, 1–1. doi:10.1145/3167102

[2] Agda Team. 2024. *Agda Standard Library*. Version 2.2, https://github.com/agda/agda-stdlib.

[3] Elton M. Cardoso, Regina De Paula, Daniel Pereira, Leonardo Reis, and Rodrigo Geraldo Ribeiro. 2023. Type-based Termination Analysis for Parsing Expression Grammars. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC)*. Association for Computing Machinery, Tallinn, Estonia, 1–8. doi:10.1145/3555776.3577620

[4] Guilherme Daher, Elton Cardoso, Leonardo Reis, and Rodrigo Ribeiro. 2025. Pest control: A formal model of the Pest parser generator. In *Proceedings of the XXIX Brazilian Symposium on Programming Languages (SBLP)*. SBC, Recife, Brazil. doi:10.5753/sblp.2025.10399

[5] Nicolaas Govert de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392.

[6] Sérgio Queiroz de Medeiros and Carlos Olarte. 2020. A Semantic Framework for PEGs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. Association for Computing Machinery, 230–245. doi:10.1145/3426425.3426944

[7] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*. 111–122. doi:10.1145/964001.964011

[8] Jean-Yves Girard, Yves Lafont, and Paul Taylor. 1989. *Proofs and Types*. Cambridge University Press.

[9] Adam Koprowski and Henri Binsztok. 2010. TRX: A Formally Verified Parser Interpreter. In *Proceedings of the 19th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 6012)*. 345–365. doi:10.1007/978-3-642-11957-6_19

[10] Neelakantan R. Krishnaswami and Jeremy Yallop. 2019. A Typed, Algebraic Approach to Parsing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, Phoenix, AZ, USA, 379–393. doi:10.1145/3314221.3314625

[11] Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph. D. Dissertation. Chalmers University of Technology.

[12] Rodrigo Ribeiro, Leonardo V. S. Reis, Samuel Feitosa, and Elton M. Cardoso. 2019. Towards Typed Semantics for Parsing Expression Grammars. In *Proceedings of the XXIII Brazilian Symposium on Programming Languages* (Salvador, Brazil) *(SBLP '19)*. Association for Computing Machinery, New York, NY, USA, 70–77. doi:10.1145/3355378.3355388

[13] Natarajan Shankar and Zephyr Lucas. 2024. Robust Verification of PEG Parser Interpreters. In *2024 IEEE Security and Privacy Workshops (SPW)*. 180–188. doi:10.1109/SPW63631.2024.00022

[14] William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32, 2 (1967), 198–212. doi:10.2307/2271658

[15] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda*. https://plfa.inf.ed.ac.uk/.