

Closure Properties of General Grammars Formally Verified

Martin Dvorak, Jasmin Blanchette

2023-08-03

Overview

- ▶ Lean 3
- ▶ Mathlib as of 2022-03-15
- ▶ 12 500 lines of code
- ▶ no dependent types for data
- ▶ no effort towards constructivism

<https://github.com/madvorak/grammars/tree/publish>

Symbols

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

Symbols

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

- ▶ We don't explicitly state that T and N must be finite.
- ▶ Only a finite amount of symbols will appear in rewrite rules.

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

Rules

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

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

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

Grammars

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



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

Grammar transformations

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

```

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

```

Grammar derivations

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

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

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

Words generated by a grammar

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

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

def language (T : Type) : Type :=
  set (list T)
```

Language of a grammar

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

def language (T : Type) : Type :=
  set (list T)

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



```
def language (T : Type) : Type :=  
  set (list T)
```

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

Type-0 languages

```
def language (T : Type) : Type :=  
  set (list T)  
  
def grammar_language (g : grammar T) :  
  language T :=  
  set_of (grammar_generates g)  
  
def is_T0 (L : language T) : Prop :=  
  ∃ g : grammar T, grammar_language g = L
```

Union of languages

```
def set.union (s1 s2 : set T) : set T :=  
{ a | a ∈ s1 ∨ a ∈ s2 }  
  
instance : language.has_add (language T) :=  
⟨set.union⟩
```

Union of languages

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

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

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

Proof of closure of type-0 language (under union)

- ▶ given g_1 and $g_2 \rightsquigarrow$ construct g

Proof of closure of type-0 language (under union)

- ▶ given g_1 and $g_2 \rightsquigarrow$ construct g
- ▶ prove, for any word,
 - ▶ generated by g_1 or $g_2 \implies$ can be generated by g

Proof of closure of type-0 language (under union)

- ▶ given g_1 and $g_2 \rightsquigarrow$ construct g
- ▶ prove, for any word,
 - ▶ generated by g_1 or $g_2 \implies$ can be generated by g
 - ▶ generated by $g \implies$ can be generated by g_1 or g_2

Reversal of a language

```
def reverse_lang (L : language T) :  
  language T :=  
   $\lambda w : \text{list } T, w.\text{reverse} \in L$ 
```


Reversal of a language

```
def reverse_lang (L : language T) :  
  language T :=  
  λ w : list T, w.reverse ∈ L  
  
theorem T0_of_reverse_T0 (L : language T) :  
is_T0 L → is_T0 (reverse_lang L)
```

Concatenation of languages

```
def set.image2 (f :  $\alpha \rightarrow \beta \rightarrow \gamma$ )  
  (s : set  $\alpha$ ) (t : set  $\beta$ ) : set  $\gamma$  :=  
{ c |  $\exists a b, a \in s \wedge b \in t \wedge f a b = c$  }  
  
instance : language.has_mul (language T) :=  
{set.image2 (++)}
```

Concatenation of languages

```
def set.image2 (f :  $\alpha \rightarrow \beta \rightarrow \gamma$ )  
  (s : set  $\alpha$ ) (t : set  $\beta$ ) : set  $\gamma$  :=  
{ c |  $\exists a b, a \in s \wedge b \in t \wedge f a b = c$  }  
  
instance : language.has_mul (language T) :=  
<set.image2 (++)>  
  
theorem T0_of_T0_c_T0 (L1 L2 : language T) :  
  is_T0 L1  $\wedge$  is_T0 L2  $\rightarrow$  is_T0 (L1 * L2)
```

Kleene star of a language

```
def language.star (L : language T) :  
  language T :=  
{ x |  $\exists S : \text{list (list T)}, x = S.\text{join}$   
       $\wedge \forall y \in S, y \in L$  }
```

Kleene star of a language

```
def language.star (L : language T) :  
  language T :=  
{ x |  $\exists S : \text{list (list T)}, x = S.\text{join}$   
       $\wedge \forall y \in S, y \in L$  }  
  
theorem T0_of_star_T0 (L : language T) :  
  is_T0 L  $\rightarrow$  is_T0 L.star
```

Related work

Chomsky hierarchy's lower levels

- ▶ Finite automata and regular expressions
 - ▶ in many languages
- ▶ Context-free grammars
 - ▶ (1992) Carlson et al. [Mizar]
 - ▶ (2007) Minamide [Isabelle]
 - ▶ (2010) Barthwal and Norrish [HOL4]
 - ▶ (2015) Firsov and Uustalu [Agda]
 - ▶ (2019) Ramos [Coq]

Related work

Turing-complete models

- ▶ Turing machines
 - ▶ (2001) Chen and Nakamura [Mizar]
 - ▶ (2012) Asperti and Ricciotti [Matita]
 - ▶ (2013) Xu et al. [Isabelle]
 - ▶ (2019) Carneiro [Lean]
 - ▶ (2020) Forster et al. [Coq]
 - ▶ (2023) Balbach [Isabelle]
- ▶ Random access machines
 - ▶ (2003) Coen [Coq]
- ▶ Lambda calculus
 - ▶ (2017–2021) Forster, Kunze, and their colleagues [Coq]
- ▶ Partial recursive functions
 - ▶ (2011) Norrish [HOL4]
 - ▶ (2019) Carneiro [Lean]

Future work

Chomsky hierarchy

- ▶ Mathlib has regular languages.
- ▶ We have context-free languages.
- ▶ We want context-sensitive languages.

Equivalence (computability theory) between TMs and grammars

- ▶ Turing machine \rightsquigarrow General grammar
- ▶ General grammar \rightsquigarrow Kuroda-NF grammar \rightsquigarrow
Nondeterministic multi-stack machine \rightsquigarrow Turing machine