Propositional Logic as Symbolic Free Form Grammar

by Sven Nilsen, 2019

In this paper I argue that propositional logic is isomorphic to proving theorems about the existence of symbols in a free form grammar sentence. This gives an interpretation to logical theorem proving.

A language grammar is made up of rules such as:

<subject> <predicate> <object>

For example:

Alice sees Bob
Alice is_not_far_away_from Bob

If Alice sees Bob, then Alice is not far away from Bob. With other words, the meaning of one sentence might imply the meaning of another sentence, making inference possible.

Alice sees Bob => Alice is_not_far_away_from Bob

However, instead of saying that one sentence implies the meaning of another sentence, one can express the full meaning as a single sentence:

Alice sees ∧ is_not_far_away_from Bob

Such sentences have meaning because of Higher Order Operator Overloading. Two predicates that operate on the same input can be combined into a new predicate since the truth value of a predicate is boolean.

```
sees : subject × object → bool
is_not_far_away_from : subject × object → bool

sees ∧ is_not_far_away_from : subject × object → bool =
    \(s : subject, o : object) = sees(s, o) ∧  is_not_far_away_from(s, o)
```

To distinguish between `Alice` as subject from `Alice` as object, one can create two propositions:

```
alice_as_subject : bool
alice_as_object : bool
```

This makes the grammar invariant with respect to position, a free form grammar, where propositions have a truth value defined to be the existence of a symbol within a sentence:

alice_as_subject := ∃ sym : Sentence { sym = “alice_as_subject” }

If every sentence of the grammar of the language requires a subject, but does not permit more than one, the XOR operator `⊻` can be used to do theorem proving about subjects:

alice_as_subject ⊻ bob_as_subject

¬alice_as_subject => bob_as_subject

Therefore, propositional logic is isomorphic to proving theorems about the existence of symbols in a free form grammar sentence. Since the grammar is in free form, it is language independent. By encoding the grammar as logical axioms, one can use propositional logic to study which properties of a set of sentences depends on which parts of the grammar definition.

The two most important building blocks of type theory are product types and sum types:

(x : A) × (y : B)	x ∧ y
(x : A) + (y : B)	x ⊻ y

These two building blocks corresponds to logical AND and logical XOR of the variables.

For example, when defining a struct in Rust, one takes the product of two types:

struct Person { first_name : str, last_name : str } => str × str

Person { first_name: “Alice”, last_name: “Albertson” }

alice_as_person_first_name ∧ albertson_as_person_last_name

For example, when defining an enum in Rust, one takes the sum of two types:

enum Option<T> { None, Some(T) } => None + Some(T)

Option<bool> => None + Some(false) + Some(true)

none ⊻ some_false ⊻ some_true

One can create more complex data structures from previously defined data structures:

struct Alive { person : Person, status: Option<bool> }

```
Alive {
        person: Person { first_name: “Alice”, last_name: “Albertson” },
        status: Some(true)
}
```

(alice_as_alive_person_first_name ∧ albertson_as_alive_person_last_name) ∧
some_true_as_alive_status

A concrete variable is a sentence where propositions about existence of symbols are joined by `∧`.

Now, I will show that this can be generalized to prove things about sub-types.

Assume that there is a sub-type:

x : ([first_name] “Alice” ∨ [first_name] “Bob”) ∧ [last_name] “Albertson”

Writing it in another form:

x := Person { first_name: “Alice” | “Bob”, last_name: “Albertson” }

(alice_as_person_first_name ⊻ bob_as_person_first_name) ∧ albertson_as_person_last_name

A sub-type is a sentence where propositions about existence of symbols are joined by `∧` and `⊻`.

The XOR operator `⊻` assumes that the two arguments are exclusive. This might lead to unsoundness if a symbol is reused across multiple contexts.

To avoid this, one must make sure that a term `a` that occurs in both parts is joined by a term that is different in the two parts, `b` and `c`:

a ∧ b ⊻ a ∧ c

The above can be written:

a ∧ (b ⊻ c)

This is the reason that safe sum types, such as `enum` in Rust, uses a symbol to identify the variant.

Now, take a closer look at the following sub-type:

x : [first_name] “Alice” ∨ [first_name] “Bob”

The OR operator `∨` here means a Higher Order Operator Overloading of two lambda/closures:

[first_name] “Alice” := \(x : Person) = x.first_name == “Alice”
[first_name] “Bob” := \(x : Person) = x.first_name == “Bob”

[first_name] “Alice” ∨ [first_name] “Bob” :=
\(x : Person) = (x.first_name == “Alice”) ∨ (x.first_name == “Bob”)

However, in the propositional free form grammar the XOR operator `⊻` is used:

alice_as_person_first_name ⊻ bob_as_person_first_name

The reason for this is that the function `first_name` can only return one value for each input. Since it returns “Alice” for one input `$x_0$` and “Bob” for another input `$x_1$`, it means that `$x_0 \neg= x_1$`. This constraint needs to be encoded in propositional logic explicitly, by using `⊻` instead of `∨`. For every input which a function returns something for, the sub-type can be written as propositions joined by `⊻`, therefore a finite sub-type can be translated into free form grammar, which is propositional logic.