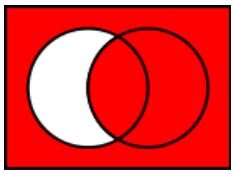
Logical Properties of Propositional Logic Systems

by Sven Nilsen, 2019

In this paper I introduce two functions that be used to construct all logical properties and only those of propositional logic systems. I also prove that all theorems of probability theory are logical properties.

The Venn diagram[1] of material implication is the following:



A B

A => B A implies B

(A => B) <=> (¬A ∨ B)

Imagine moving the circle of `B` around and how it affects the shape of the “moon” in the Venn diagram above. If `A` is fixed while `B` can vary, then `A` is a propositional logic[2] system.

A propositional logic system can be thought of as choosing some set of assumptions and then examine which conclusions follows from various proofs. Another way of thinking of such systems is as modeling something in propositional logic using rules and “import” this model whenever one needs to reason about the system.

A logical property is a property that does not depend on the expression of the model. For example, if I use a blue or red pen to write about logic, then the color is irrelevant to what I mean to say. Logical properties are those things I might wish to communicate.

It is desirable to make distinctions between logical properties of logic in general and those that are relevant for a specific logical system. One technique is to use a formal definition of something which can be used to construct all and only logical properties that are relevant for a specific system.

Formally (this definition adds `_plus` to the name, which is explained later in this paper):

```
propositional_logic_system_plus := \(A : \bool) = \(B : \bool) = |A => B|

propositional_logic_system_plus : \bool → \bool → nat
```

A predicate[3] is a function that returns `bool`, which is written `f : \bool`.
This notation[4] is used in path semantics when reasoning with Higher Order Operator Overloading[5].

`A` is identical to a function that counts true cases of material implication of a concluding predicate. With other words, it turns out that if one can measure the area of red pixels in such Venn diagrams of material implication, then one can derive everything that is possible to know about `A`.

No more or less.

Here, a propositional logic system is defined as a function, because whatever we want to say about such systems can be constructed from this function alone. The full version looks like this:

```
propositional_logic_system_plus := \(A : \bool) = \(B : \bool) =
    ∑ x { if ¬A(x) ∨ B(x) { 1 } else { 0 } }
```

The language that this function permits can be classified as `nat$^{2+}$`[6].

If `s` is a system:

```
s := propositional_logic_system_plus(A)
```

Then a tuple can be constructed for any predicate `B` by:

```
(s(B) - s(\false), s(¬B) - s(\false)) : nat²
```

The `+` symbol in the classification `nat$^{2+}$` comes from the knowledge `s(\false)`. This knowledge leaks information about the size of the system relative to its modeled brute-force theorem proving size. Therefore, one can talk about the properties that the system has as a brute-force theorem prover.

For example, if one wants to know how many bits the system uses:

```
bits_of_plus(s : \bool → nat) = ln(s(\true)) / ln(2)

bits_of_plus : (\bool → nat) → real
```

The optimal number of bits for the system is:

```
optimal_bits_of_plus(s : \bool → nat) = ln(s(\true) – s(\false)) / ln(2)

optimal_bits_of_plus : (\bool → nat) → real
```

To talk about what it means to finds the optimal number of bits, one can use another function:

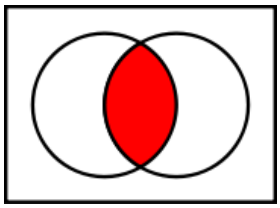
```
propositional_logic_system := \(A : \bool) = \(B : \bool) =
    ∑ x { if A(x) ∧ B(x) { 1 } else { 0 } }

s := propositional_logic_system(A)

(s(B), s(¬B)) : nat²
```

The language that this function permits can be classified as `nat$^{2}$`[6].

Venn diagram:



This function describes a set of systems which are logically equivalent, except how they are implemented as brute-force theorem provers. It is closer to what we mean by a system mathematically.

One can choose either `propositional_logic_system_plus` (nat^{2+}) or `propositional_logic_system` (nat^{2}). Which version one uses depends on whether one wants to reason about brute-force efficiency.

In brute-force theorem proving libraries[7], one might want to use `$nat^{2+}$`, since every system has a specific implementation with that property.

In theorem proving using path semantics, the specific implementation might be unknown, and therefore the `$nat^{2}$` definition is more convenient.

At the first glance, it might look like `$nat^{2+}$` encodes more information than `$nat^{2}$`. However, this is not the case.

The optimal number of bits in `$nat^{2}$` and `$nat^{2+}$` are the same in both definitions:

optimal_bits_of(s : \bool) = s(\true)

optimal_bits_of · propositional_logic_system <=> optimal_bits_of_plus · propositional_logic_system_plus

By inferring `A` from examining its logical properties in `$nat^{2}$`, one can construct `$nat^{2+}$`. The bits that any specific implementation has in `$nat^{2+}$` can be derived from `$nat^{2}$`.

Therefore, all theorems in language `$nat^{2}$` can be proven in language `$nat^{2+}$` and vice versa:

$nat^{2} <=> nat^{2+}$ The languages of `$nat^{2}$` and `$nat^{2+}$` are equivalent

To prove something in either `$nat^{2}$` or `$nat^{2+}$`, one can use the following function:

prove := \(s : \bool → nat) = \(f : \bool) = s(f) == s(\true)

Since languages classified as `opt[prob]` (those that can prove theorems of probability theory) are provable in `$nat^{2}$`, this proves that all theorems of probability theory are logical properties:

opt[prob] → nat^{2} The arrow means “provable in”

References:

[1] “Venn diagram”
Wikipedia
https://en.wikipedia.org/wiki/Venn_diagram

[2] “Propositional calculus”
Wikipedia
https://en.wikipedia.org/wiki/Propositional_calculus

[3] “Predicate (mathematical logic)”
Wikipedia
https://en.wikipedia.org/wiki/Predicate_%28mathematical_logic%29

[4] “Higher Order Operator Overloading and Notation for Parameters”
Sven Nilsen, 2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/higher-order-operator-overloading-and-notation-for-parameters.pdf

[5] “Higher Order Operator Overloading”
Sven Nilsen, 2018
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/higher-order-operator-overloading.pdf

[6] “Logical Interpretations of Boolean Path Semantics”
Sven Nilsen, 2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/logical-interpretations-of-boolean-path-semantics.pdf

[7] “Pocket Prover – A fast, brute force, automatic theorem prover for first order logic”
AdvancedResearch – Sven Nilsen
https://github.com/advancedresearch/pocket_prover