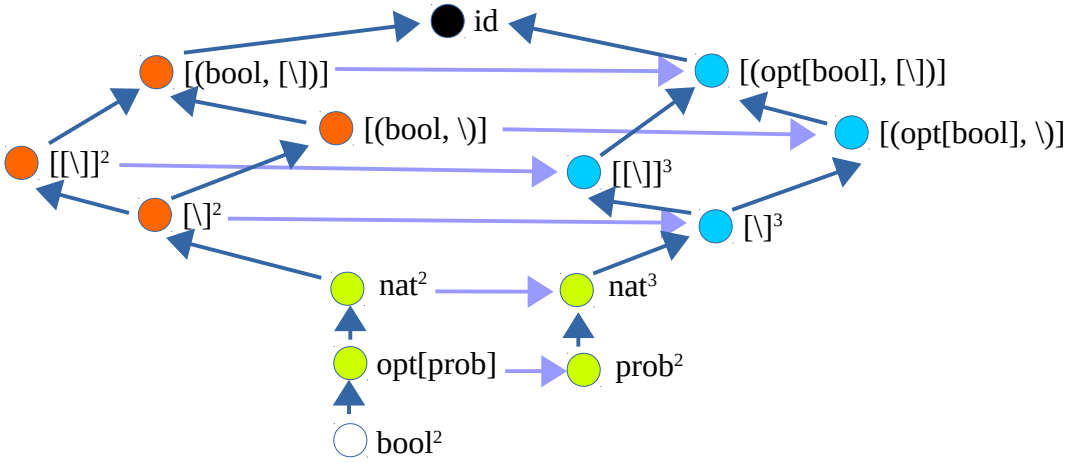
Logical Interpretations of Boolean Path Semantics

by Sven Nilsen, 2019

In this paper I introduce the category of 14 logical interpretations of Boolean Path Semantics.

An overview of how various interpretations are related to each other can be represented as a category, where arrows mean “provable in”. Equivalent languages and “obvious” arrows are not shown:



Colors shows that non-trivial interpretations are grouped into commutative squares.

The standard interpretation of probability theory `prob` is not included, because it is not “logical”.
Only interpretations that preserve the structure of the existential path are considered “logical”.
The reason for this is explained and proven later in this paper.

In Boolean Path Semantics, any function `f` is restricted to the type:

$$f : bool^N \to bool$$

The trivial path of `f` is the domain constraint `∀f` that also has the same type:

$$\forall f : bool^N \to bool$$

The identity of a function changes the when the trivial path changes.

Using parameter notation for Higher Order Operator Overloading,
one can generalize Boolean Path Semantics to all functions that return `bool`:

$$f : \backslash bool$$
$$\forall f : \backslash bool$$

In path semantics, when talking about some object `x`, it is common to use a sub-type:

$$x : [f]\ true$$

Simplified to:

$$x : f$$

It is implied from this expression that `x` would then also satisfy the trivial path:

$$x : f \wedge \forall f$$

The trivial path of a family of functions is the intersection of all individual domain constraints:

$$x : f_0 \wedge f_1 \wedge f_2 \wedge \ldots \wedge \forall f \qquad\qquad \forall f <=> \backslash(x) = \forall\ i\ \{\ \forall f_i(x)\ \}$$

If all individual domain constraints are the same, they are equal to their intersection:

$$\forall f_0 <=> \forall f_1 <=> \forall f_2 <=> \ldots <=> \forall f$$

The property `∀f` above can be thought of as a set of objects.
This set contains all possible things or ideas that one can talk about.
Using a family of functions is like having a vocabulary in some language.

The probability interpretation of Boolean Path Semantics is a way of transforming the truth values of the language from booleans to probability. It is desirable that a such interpretation does not loose power of expressiveness during the transformation. Primarily, this means the power of logic.

Previously, one talked about some member of the set `∀f` having a property `f`:

$$x : f$$

However, in logic this is often not very interesting. Instead, it is more interesting to talk about the set itself. This is because the central topic of interest in logic is theorem proving. To do theorem proving one needs proofs. Proofs are quantified over the set `∀f`. Therefore, proofs say something about the set itself, which might not be about particular members of the set. To quantify, the most natural “loops” are for-all (`∀`) and there-exists (`∃`).

In logic, one is interested in the following kind of questions:

$\forall\ x : \forall f\ \{\ f(x)\ \}$	Does all members of the set `∀f` satisfy the property `f`?
$\exists\ x : \forall f\ \{\ f(x)\ \}$	Does any member of the set `∀f` satisfy the property `f`?

These expressions are used to talk about the set `∀f`.
Notice that the function `f` encodes the meaning of the “loop” itself.

The probability interpretation is a method of unifying `∀` and `∃` loops.

Instead of working with the set `∀f` directly, we want to use a simpler representation, primarily:

$(\exists f_p\{\forall f\})(true)$	How often is the property `f` true?

Here `$\exists f_p\{\forall f\}$` is the probabilistic existential path of `f`,
which might be thought of as a construction of a `count` function:

$(\exists f_p\{\forall f\})(true)$	=	$\lvert f\rvert / (\lvert f\rvert + \lvert \neg f\rvert)$

$|f| <=> count(f, true)$
$|¬f| <=> count(f, false)$

count := \(f : \bool, t : bool) = ∑ x : ∀f { if f(x) == t { 1 } else { 0 } }

Yet, the existential path of functions that returns `bool` have the following existential path equations:

∃f{∀f} <=> \false	none probability
∃f{∀f} <=> not	0
∃f{∀f} <=> id	1
∃f{∀f} <=> \true	some probability between 0 and 1

Notice that there is no way to express “none probability” using a single real number in the unit interval.

To preserve the existential path, one must use truth values of the following type:

opt[prob]	none() | some(x) for `x : prob`

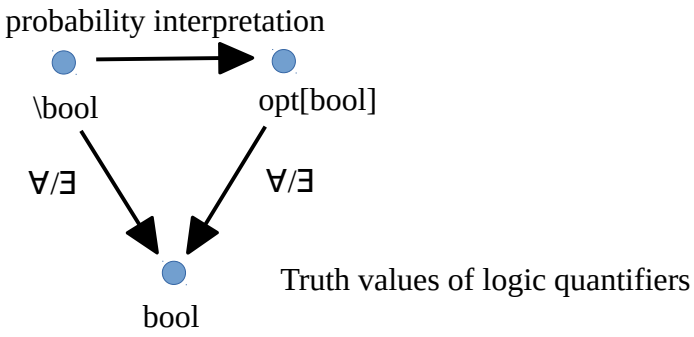
What does it mean? It turns out that that “none probability” has an interpretation in logic.

The equation `∃f{∀f} <=> \false` is true if and only if `∀f <=> \false` is true.

∀f <=> \false	=>	∀ x : ∀f { f(x) } = true
∀f <=> \false	=>	∃ x : ∀f { f(x) } = false

The definition of `∀` and `∃` quantifiers uses two different boolean values.
This is why it is impossible to encode the full existential path as a single number in the unit interval.
The statement “none probability” simply means that the “loop” was empty.

To visualize the transformation, one can use the following diagram:



The probability interpretation preserves the structure of the existential path,
and therefore preserve the power of logic.

Standard probability theory therefore assume a non-empty model set `∀f`:

∀f <¬=> \false	Standard probability theory

Interpreting `\bool` is not unique, it just happens that `opt[prob]` is most similar to standard probability.
The types of truth values can be used to classify various interpretations and their generalizations:

$bool^2$	A tuple `(∃f{∀f}(true), ∃f{∀f}(false))`
bool → bool	A function `∃f{∀f}`
prob	Standard probability theory
opt[prob]	“Correct” probability interpretation
nat^2	A tuple `(|f|, |¬f|)`
nat^3	A tuple `(|f|, |¬f|, |¬∀f|)`
$[\]^2$	Lists of `true` and `false` segments
$[\]^3$	Lists of `true`, `false` and “ignored” segments
$[[\]]^2$	Lists of `true` and `false` segments for nested loops
$[[\]]^3$	Lists of `true`, `false` and “ignored” segments for nested loops
[(bool, \)]	Ordered list of `true` and `false`
[(opt[bool], \)]	Ordered list of `true`, `false` and “ignored”
[(bool, [\])]	Ordered list of `true` and `false` for nested loops
[(opt[bool], [\])]	Ordered list of `true`, `false` and “ignored” for nested loops
id	The set `∀f` mapping to the truth value of itself

All theorems that can be proven in standard probability theory `prob` can be proven in `opt[prob]`.
All theorems that can be proven in `opt[prob]` can be proven in $`nat^2`$, and so on.

In the `id` representation, the full information we have about the set `∀f` is available.
There is no more complex interpretation than the interpretation of all ideas or things themselves.
This interpretation depends on how ideas are expressed, which might be non-computational properties.
One can say that the truth value depends on the expression of `f`.

We would like the model of the world to be reasonable, but not unreasonable. We might want to
examine the language that preserves logically equivalent properties, but ignores all other information.

The most complex interpretation that does not depend on the expression of `f` is the following:

[(opt[bool], [\])]

This can be thought of as a transformed function returning `opt[bool]`:

f’ : \opt[bool]

∀ x : ¬∀f { f’(x) == none() }
∀ x : ∀f { f’(x) = some(f(x)) }

This transform function `f'` normalizes the trivial path of `f`, such that:

```
∀f' <=> \true
```

However, the same could be said about another interpretation:

```
[(opt[bool], \)]
```

Some languages can have more powerful answers depending on how you ask questions.
The language `[(opt[bool], [\])]` has the most powerful answers of Boolean Path Semantics.
This representation contains all information that you need to prove all logical equivalent theorems.
However, it does not contain any more information than necessary.
The language `[(opt[bool], \)]` is almost as powerful as `[(opt[bool], [\])]`, but not quite.
The difference is about the semantics of nested loops.

The value `[\]` instead of `\` means that if you nest, or compose, logical loops, a tuple exists for every possible input that "proves" the proof over the nested quantifiers. One can create a language that looks like logic, but instead transforms the expression into a logically equivalent form:

```
∀ x : a { ∃ y : b { f(x, y) } }          : [(opt[bool], (∀a, ∀b))]
```

Under this interpretation, the loop composition is normalized:

```
∀ x : ∀a {
        if ¬a(x) { continue }
        else { ∃ y : ∀b {
                if ¬b(y) {continue}
                else { f(x, y) }
        } }
}
```

When the language uses `continue`, it adds `none()` to the resulting list, together with a tuple `(x, y)`.
Otherwise, it adds `some(f(x, y))` to the resulting list, together with the tuple `(x, y)`.
The tuple `(x, y)` informs which state the first component of type `opt[bool]` was obtained from.

A fixed deterministic order of looping through can reduce the representation:

```
[(opt[bool], [\])]        =>        [opt[bool]] + fixed deterministic order
```

Here, the arrow `=>` is not mean to be interpreted logically, but abstractly as a transformation.

It is not necessary to keep track of the state that "proves" the proof, because this information can be retrieved from a function that takes the index of the list and returns the state:

```
nat → (∀a, ∀b)
```

All languages with fixed deterministic order equipped with a such function,
are equivalent logically powerful to `[(opt[bool], [\])]`:

[(opt[bool], [\])] <=> [opt[bool]] + fixed deterministic order + nat → (∀a, ∀b)

Conjecture: Proving two languages at this level to be equivalent powerful is undecidable (in general).

On the other hand, the two languages `bool$^2$` and `bool → bool` are easily proven equivalent:

bool^2 <=> bool → bool

This is because a tuple of type `bool$^2$` can encode all functions of type `bool → bool`:

(false, false) <=> \false
(false, true) <=> not
(true, false) <=> id
(true, true) <=> \true

These two languages are the simplest and encodes the existential path of functions that returns `bool`.

The `prob$^2$` language is quite interesting, because it encodes the probabilistic existential path:

prob^2 <=> bool → prob

Possible interpretations:

$\exists f_p\{\forall f\}$: bool → prob
$((\exists f_p\{\forall f\})(true), (\exists f_p\{\forall f\})(false))$: prob^2
$((\exists f_p\{\forall f\})(true), (\exists \forall f_p)(true))$: prob^2

Since probability adds up to `1`, one can use the following law:

$(\exists f_p\{\forall f\})(true) + (\exists f_p\{\forall f\})(false) + (\exists \forall f_p)(false) = 1$

This is why it is sufficient to keep 2 numbers instead of 3.

For example, if a function returns `true` 30% of the time and `false` 30% of the type, then the remaining 40% must be because the input fails to pass the domain constraint.

The two tuple interpretations can be translated into each other, and they both encode the same information as the full probabilistic existential path.

From this law, one can show that `prob$^2$` has an analogue of `prob` relative to `opt[prob]`.

By normalizing away the term `$(\exists \forall f_p)(false)$`, one can reason using a transform `$\exists f'_p$`:

$(\exists f'_p\{\forall f\})(true) + (\exists f'_p\{\forall f\})(false) = 1$

This language assumes the axiom of excluded middle and can be categorized as `prob'$^2$`.

Any non-zero probability in `prob'$^{2}$` corresponds to a non-zero probability in `prob$^{2}$`.
Any zero probability in `prob'$^{2}$` corresponds to a zero probability in `prob$^{2}$`.

$$(\exists f'_{p}\{\forall f\})(x) : (\neg = 0) \quad <=> \quad (\exists f_{p}\{\forall f\})(x) : (\neg = 0)$$
$$(\exists f'_{p}\{\forall f\})(x) : (= 0) \quad <=> \quad (\exists f_{p}\{\forall f\})(x) : (= 0)$$

Therefore, the non-zero property of the probability is logically equivalent for the two languages:

$$(\neg = 0) \cdot (\exists f'_{p}\{\forall f\}) \quad <=> \quad (\neg = 0) \cdot (\exists f_{p}\{\forall f\})$$

Since the non-zero property of the probabilistic existential path is just the existential path:

$$(\neg = 0) \cdot (\exists f_{p}\{\forall f\}) \quad <=> \quad \exists f\{\forall f\}$$

It means that `prob'$^{2}$` seemingly preserves the existential path, but this turns out not to be the case.
The following assumption was made:

$$(\exists f'_{p}\{\forall f\})(true) + (\exists f'_{p}\{\forall f\})(false) = 1$$

If one is zero, the other must be one.
They can not be both zero.
It corresponds to an assumption on the existential path:

$$\forall f <\neg=> \backslash false$$

Which is the same assumption as standard probability theory.
The language `prob'$^{2}$` is reducible to `prob`:

$$prob'^{2} \quad <=> \quad prob$$

However, probabilities can be encoded with different semantics this way, even without normalizing:

$$((\exists f_{p}\{\forall f\})(true) + (\exists f_{p}\{\forall f\})(false)) + (\exists \forall f_{p})(false) = 1$$
$$(\exists f_{p}\{\forall f\})(true) + ((\exists f_{p}\{\forall f\})(false) + (\exists \forall f_{p})(false)) = 1$$
$$((\exists f_{p}\{\forall f\})(true) + (\exists \forall f_{p})(false)) + (\exists f_{p}\{\forall f\})(false) = 1$$

Conjecture: For all functions, this method of encoding semantics becomes a symmetry that makes it impossible to tell whether you are in one language or another from its internal structure.

The result is that `prob` is ambiguous and not considered “logical”.
With other words, it is impossible to translate from Boolean Path Semantics to `prob`
without loosing some of the power of expressiveness that logic gives you.