Probabilistic Paths

by Sven Nilsen, 2017

Probabilistic existential paths are written `$\exists_p f$` and used to tell how frequently a function `f` returns some value. They are powerful but not enough to describe the probabilistic analogue of symmetric paths, written `f[g]`, or asymmetric paths, written `$f[g_{i \to n}]$` and used to predict properties of the output from properties of the input (these are sometimes called "normal" paths). It is desirable to predict probabilities of output satisfying some constraint from probabilistic knowledge about the input constraints, but it is incredibly hard to wrap your head such kind of problems. Having a precise definition makes it easier to derive the algorithm. In this paper I represent the higher order path semantical algorithm for finding probabilistic paths and then give two demonstrations. One curious property of this algorithm is when using negative constraints, it operates using normal laws of probability but for numbers that are outside the valid probability range, but somehow the answer always end up in the valid probability range. A lot of new notation had to be developed to express the idea of probabilistic paths, so this is advanced path semantics at its finest!

A probabilistic path is defined as:

$f[g_{i\to n}]_p$:= \(b : [] ∧ [len] $|g_{i\to n}|$, p : [real] ∧ [len] $|g_i|$) = ∑ j 2^$|g_i|$ {
    $(\exists_p(g_n \cdot f)\{2\text{\^{}}([g_i]\ b)\}(\beta_j))(b_n) \cdot \prod k\ |g_i|\ \{\ \beta_{jk}((p_k-(\exists_p g_{ik})(b_k))/(1-(\exists_p g_{ik})(b_k)))\ \}$
}

A simpler version with probabilities replaced by pre-computed confidence factors:

$f[g_{i\to n}]_c$:= \(b : [] ∧ [len] $|g_{i\to n}|$, c : [real] ∧ [len] $|g_i|$) = ∑ j 2^$|g_i|$ {
    $(\exists_p(g_n \cdot f)\{2\text{\^{}}([g_i]\ b)\}(\beta_j))(b_n) \cdot \prod k\ |g_i|\ \{\ \beta_{jk}(c_k)\ \}$
}

As usual in path semantics, one can not use this definition to compute stuff directly, but it describes how to derive the algorithm for specific problems.

This is better explained by using an example:

add[even] <=> eq

add : nat × nat → nat
even : nat → bool
eq : bool × bool → bool

This is a commonly known path in path semantics, that tells if you want to predict whether adding two numbers results in an even number, you can take the `even : nat → bool` property of the input numbers and check if they are equal. Adding two even numbers results in an even number, and adding two odd numbers results in an even number, because the "evenness" property of the input numbers are equal. Otherwise, you get an odd number, in that case the `even` function returns `false`.

Another way to write this is:

even(add(a, b)) = eq(even(a), even(b))

Equations of this form appears everywhere, and this makes path semantics so useful. Paths are equivalent to equations, but they are much easier to manipulate because we can eliminate the variables and use the algebraic rules that follows naturally from the notation.

More, path semantics allows us to reason about sub-types in a similar way to dependently types, but without needing to encode the information into the types:

add(a : [even] e_a, b : [even] e_b) $\rightarrow$ [even] e_a == e_b { a + b }

This is equal to constraining the `add` function, written in front of arguments using curly braces `{}`:

add{[even] e_a, [even] e_b}(a, b) $\rightarrow$ [even] e_a == e_b { a + b }

By extracting the type level of this function, one can see this as lifting `[even]` outside the curly braces:

add[even](e_a, e_b) = e_a == e_b

In short form this is just written:

add[even] <=> eq

Now, we want to find the probabilistic path:

add[even]$_p$

This can be thought of as assigning a probability to inputs and output having some “evenness” value:

a : [even] e_a	Assigned a probability `$p_a$`
b : [even] e_b	Assigned a probability `$p_b$`
add(a, b) : [even] e_c	Assigned a probability `$p_c$`

The probabilistic path is a function that takes a list of values `[$e_a$, $e_b$, $e_c$]`
and a list of probabilities `[$p_a$, $p_b$]` and computes the probability `$p_c$`.

In the algorithm for deriving a probabilistic path, we start by replacing the following expression:

$(\exists_p g_{ik})(b_k) \Rightarrow (\exists_p \text{even})(e_k) \Rightarrow 0.5$

This means how likely the `even` function is to return `$e_k$ = true` (an even number) or `$e_k$ = false` (an odd number). Half of the numbers are even, and the other half is odd, so it returns 0.5 on all inputs. Now we can simplify the following expression:

$(p_k-(\exists_p g_{ik})(b_k))/(1-(\exists_p g_{ik})(b_k)) \Rightarrow (p_k-0.5)/(1-0.5) \Rightarrow (p_k-0.5)/0.5 \Rightarrow 2p_k-1$

The expression above converts probabilities into numbers called “confidence factors”. A confidence factor is almost the same as a probability, but has a different meaning. A probability 0 means “this does not happen” while a confidence factor 0 means “does not know whether it happens or not”. A confidence factor is expressing evidence and value 0 means absence of evidence, but it could still happen anyway.

When a default probability `$p_0$` is the likelihood something happens in absence of any evidence, then the observed probability `$p$` is related to confidence factor `$c$` by the following equation:

$$p = c + (1 - c)p_0$$

Solving for `$c$`:

$$c = (p - p_0) / (1 - p_0)$$

This conversion is used because the algorithm for deriving probabilistic paths can only constrain an input of probabilistic existential paths or not constrain the input at all. It can not constraint the input in a *negative way* as a step through the calculation. The constraining of probabilistic existential paths is expressed in the algorithm as:

$$\exists_p(g_n \cdot f)\{2\text{^}([g_i]\ b)\}(\beta_j)$$

The `$2\text{^}([g_i]\ b)$` inside the curly braces here are not normal constraints/sub-types, but a way to construct the power-set of the constraints and then control them with a bits ` $\beta_j$`. The bits turn the constraints on/off. It is interpreted as “take the probabilistic existential path of the function composition `$g_n \cdot f$`, given that such and such constraints are turn on”. The result is a function that tells how frequently a given output is returned when the constraints are active.

It is the mechanism of probabilistic existential paths that can not handle normal probabilities, so one needs to convert probabilities to confidence factors.

However, when assigning a probability 0, one gets a negative confidence factor:

$a : [\text{even}]\ e_a$ Assigned a probability `0`

$2p_k\text{-}1 \Rightarrow 2\cdot 0\text{-}1 \Rightarrow -1$

This is a way to express “this number is not even” without telling explicitly that it is an odd number. In other examples, such as when rolling a dice, this means “the dice does not roll this number of eyes”.

Although constraining probabilistic existential paths in a negative way is not allowed, the confidence factors can turn negative. Some terms might cancel each other out, such that the output probability behaves as if it understands that “this number is not even” is equivalent to “this number is odd”.

The algorithm sums the products of some probability of the function returning some output over some given constraints, multiplied with the confidence factors of these constraints, where the confidence factor is flipped using normal probability laws, although they can get outside normal probability range, by the same bits controlling the constraints. Quite a mouthful, right?

Let us break this down. The algorithm in the general form:

f[$g_{i\to n}$]$_p$:= \(b : [] ∧ [len] |$g_{i\to n}$|, p : [real] ∧ [len] |g_i|) = ∑ j 2^|g_i| {
(∃$_p$(g_n · f){2^([g_i] b)}(β_j))(b_n) · ∏ k |g_i| { β_{jk}((p_k-(∃$_p g_{ik}$)(b_k))/(1-(∃$_p g_{ik}$)(b_k))) }
}

Inserting what we got so far:

add[even]$_p$:= \(b : [bool] ∧ [len] 3, p : [real] ∧ [len] 2) = ∑ j 4 {
(∃$_p$(even · add){$2^{[\text{even}]\ b}$}(β_j))(b_n) · ∏ k 2 { β_{jk}(2p_k-1) }
}

Now we need to unroll the sum loop. The likelihood of adding numbers having evenness \`$e_c$\`, given:

1. (∃$_p$(even · add))(e_c)
   We don't know whether any number is even or not
2. (∃$_p$(even · add){[even] e_a, _})(e_c)
   We know the first number has evenness \`$e_a$\`
3. (∃$_p$(even · add){_, [even] e_b})(e_c)
   We know the second number has evenness \`$e_b$\`
4. (∃$_p$(even · add){[even] e_a, [even] e_b})(e_c)
   We know the first number has evenness \`$e_a$\` and the second number has evenness \`$e_b$\`

Inserting by analyzing each case:

1.	0.5	Half of the numbers are even and the other half are odd
2.	0.5	For all cases
	0.5	Adding an even number with another number results in 50% chance it is even
	0.5	Adding an even number with another number results in 50% chance it is odd
	0.5	Adding an odd number with another number results in 50% chance it is even
	0.5	Adding an odd number with another number results in 50% chance is is odd
3.	0.5	Same as 2)
4.	if (e_a == e_b) == e_c { 1 } else { 0 }	Combining cases
	if e_a == e_b { 1 } else { 0 }	If \`$e_c$\` is \`true\`
	if e_a ¬= e_b { 1 } else { 0 }	If \`$e_c$\` is \`false\`

The expression \`$\beta_{jk}$(...)\` means if the bit is \`true\`, return the argument, but if it is \`false\`, subtract the argument from 1.

Unrolling the sum loop and inserting:

add[even]$_p$:= \([e_a, e_b, e_c] : [bool] ∧ [len] 3, [p_a, p_b] : [real] ∧ [len] 2) =
0.5 · (1 – (2p_a-1)) · (1 – (2p_b-1)) +
0.5 · (2p_a-1) · (1 – (2p_b-1)) +
0.5 · (1 – (2p_a-1)) · (2p_b-1) +
if (e_a == e_b) == e_c { 1 } else { 0 } · (2p_a-1) · (2p_b-1)

Notice how this is derived directly from the general algorithm by solving the sub-problems.

Testing:

```
add[even]$_p$([true, true, true], [1, 1]) = 0 + 0 + 0 + 1 = 1
add[even]$_p$([true, true, true], [0, 0]) = 2 + -1 + -1 + 1 = 1
add[even]$_p$([true, false, true], [1, 1]) = 0 + 0 + 0 + 0 = 0
add[even]$_p$([true, false, true], [1, 0]) = 0 + 1 + 0 + 0 = 1
add[even]$_p$([false, false, true], [0, 0]) = 2 + -1 + -1 + 1 = 1
add[even]$_p$([true, true, true], [0.5, 0.5]) = 0.5 + 0 + 0 + 0 = 0.5
add[even]$_p$([true, true, true], [0.25, 0.25]) = 1.125 + -0.375 + -0.375 + 0.25 = 0.625
add[even]$_p$([true, true, true], [0.75, 0.75]) = 0.125 + 0.125 + 0.125 + 0.25 = 0.625
```

The fact that the terms are sometimes outside normal probability range, but for some seemingly magical reason it ends up in the valid range, might result in lots of sleepless nights for aspiring philosophers of programming.

Here is another example:

```
mul[even] <=> or
mul[even]$_p$ := \(b : [bool] ∧ [len] 3, p : [real] ∧ [len] 2) = ∑ j 4 {
    (∃$_p$(even · mul){$2^{[\text{even}]\ b}$}($β_j$))($b_n$) · ∏ k 2 { $β_{jk}$(2$p_k$-1) }
}
mul[even]$_p$ := \([$e_a$, $e_b$, $e_c$] : [bool] ∧ [len] 3, p : [real] ∧ [len] 2) =
    (∃$_p$(even · mul))($e_c$) · (1 – (2$p_a$-1)) · (1 – (2$p_b$-1)) +
    (∃$_p$(even · mul){[even] $e_a$, _})($e_c$) · (2$p_a$-1) · (1 – (2$p_b$-1)) +
    (∃$_p$(even · mul){_, [even] $e_b$})($e_c$) · (1 – (2$p_a$-1)) · (2$p_b$-1) +
    (∃$_p$(even · mul){[even] $e_a$, [even] $e_b$})($e_c$) · (2$p_a$-1) · (2$p_b$-1)
mul[even]$_p$ := \([$e_a$, $e_b$, $e_c$] : [bool] ∧ [len] 3, p : [real] ∧ [len] 2) =
    0.75 · (1 – (2$p_a$-1)) · (1 – (2$p_b$-1)) +
    if $e_a$ { 1 } else { 0.5 } · (2$p_a$-1) · (1 – (2$p_b$-1)) +
    if $e_b$ { 1 } else { 0.5 } · (1 – (2$p_a$-1)) · (2$p_b$-1) +
    if $e_a$ ∧ $e_b$ { 1 } else { 0 } · (2$p_a$-1) · (2$p_b$-1)
```

Testing:

```
mul[even]$_p$([true, true, true], [1, 1]) = 0 + 0 + 0 + 1 = 1
mul[even]$_p$([true, true, true], [0, 0]) = 3 + -2 + -2 + 1 = 0
mul[even]$_p$([true, true, true], [0.5, 0.5]) = 0.75 + 0 + 0 + 0 = 0.75
```