Probabilistic Existential Paths

by Sven Nilsen, 2017

A probabilistic existential path is similar to an existential path, but for reasoning about uncertainty. For example, the following two functions `$f_0$` and `$f_1$` have the same existential path:

f_0 := \(x : nat ∧ (< 100)) = if (x % 2) == 0 { 1 } else { 0 }
f_1 := \(x : nat ∧ (< 100)) = if (x % 10) == 0 { 1 } else { 0 }

$∃f_0$ <=> $∃f_1$ <=> \(x : nat) = x == 0 ∨ x == 1

Yet, the function `$f_0$` returns `1` five times as often compared to `$f_1$`:

f_0 : 1 0 1 0 1 0 1 0 1 0 1 0 1 …
f_1 : 1 0 0 0 0 0 0 0 0 0 1 0 0 …

So, their probabilistic existential paths are different:

$∃_p f_0$:= \(_ : nat) = if x == 0 ∨ x == 1 { 1/2 } else { 0 }
$∃_p f_1$:= \(x : nat) = if x == 0 { 9/10 } else if x == 1 { 1/10 } else { 0 }

In the paper “Assigning Probabilities to Boolean Functions”, all boolean functions are assigned probability of returning `true`, by using a higher order `if` function:

P(if(A, B)(x)) = P(x)·P(A) + (1-P(x))·P(B)

if := \(a, b) = \(c) = if c { a } else { b }

This assumes the probabilities are independent `P(A ∩ B) = P(A) · P(B)`, because the function identity of `if` changes when the probabilities are not independent. In path semantics the function identity changes when adding any domain constraints (sub-type constraints on the input). A function with domain constraints is a partial function and does not have the same path set as the total function. Dependent probabilities automatically lead to domain constraints, as shown on the next page.

The use of probability for Boolean functions (functions type of `bool → bool → bool → ...`) is only used to establish a semantics for probabilistic existential paths. Other types, e.g. `nat ∧ (< 100)`, do not have independent probabilities for each bit in their binary representation. The probability is distributed uniformly for each member of the type when the type is finite. It is sometimes possible to assign a probability to a sub-type of an infinite type without defining it for members, but that is another topic.

The reason one can ignore the binary form, is that when encoding a type the conditional probabilities that follow from the method of encoding, automatically adds domain constraints on the binary function such that these two different ways of reasoning about the same function is logically equivalent. This is also true for the probabilities since they reflect the domain constraints. The probabilities assigned uniformly to each member of a finite type are independent for all input arguments, while the binary form might contain dependent probabilities between encoded bits of a single argument.

Just like the existential path can not inherit knowledge about the input, the probabilistic existential path uses a uniform random distribution for the input by default. This means the probability of input `P(x)` gets replaced by `1/2` after deriving the probability formula for a boolean function. The probabilistic existential path of every boolean function can be constructed by the `probx` function:

probx := \(k : real ∧ [prob] true) = \(x : bool) = if x { k } else { 1 – k }

prob := \(x : real) = x >= 0 ∧ x <= 1

For example, for the function `and`:

$P(and(x_0, x_1)) = P(x_0) \cdot P(x_1) = 1/2 \cdot 1/2 = 1/4$
$\exists_p and <=> probx(1/4)$

In probability theory there are conditional probabilities. The equivalent concept for probabilistic existential paths is constructed by adding sub-type constraints to the input before taking the probabilistic existential path:

$\exists_p and\{(= true), \_\} <=> \exists_p id <=> probx(1/2)$

Translating back and forth between the two notations is straight forward:

$P(and(a, b) \mid a) = (\exists_p and\{(= true), \_\})(true)$
$P(\neg and(a, b) \mid a) = (\exists_p and\{(= true), \_\})(false)$

In proofs, inject the sub-type constraints so you get only free inputs and then take the probabilistic existential path:

$P(and(a, b) \mid a) = (\exists_p id)(true)$
$P(and(a, b) \mid a)$
$P(and(a : (= true), b))$
$P(and\{(= true), \_\}(a, b))$
$(\exists_p and\{(= true), \_\})(true)$
    $and\{(= true), \_\} <=> id$
$(\exists_p id)(true)$

$P(\neg and(a, b) \mid a) = (\exists_p id)(false)$
$P(\neg and(a, b) \mid a)$
$P(\neg and(a : (= true), b))$
$P(\neg and\{(= true), \_\}(a, b))$
    $not \cdot and <=> nand$
$P(nand\{(= true), \_\}(a, b))$
$(\exists_p nand\{(= true), \_\})(true)$
    Here I do a trick by inverting the output.
$(\exists_p and\{(= true), \_\})(false)$
    $and\{(= true), \_\} <=> id$
$(\exists_p id)(false)$