Function Identity

by Sven Nilsen, 2017

Some people get confused by how path semantics treats function identity. This is because of the semantics of functions in path semantics is stricter than how most people think of them in programming. In many theories of mathematics, one uses axioms to define a class of objects and then treat functions as mere computational expressions on these objects, by using the axioms as a safety net for consistency. However, in path semantics one must pay extra attention to the use of functions.

In path semantics, two functions are identical when “everything that can be said” about the two functions are the same: They must be logically equivalent.

```
f <=> h
```

Some ways to disprove logical equivalence:

- Any path sets of normal paths are not equivalent
- Any existential paths, constrained or not, are not logically equivalent
- Any probabilistic existential paths, constrained or not, are not logically equivalent
- Probabilistic paths predicts different probabilities

For example, when you define a function like this:

```
f := \(x : nat, y : nat) = x + y
```

You are not allowed to “control” the arguments in any way, they must be allowed to take any value within their type. This does not mean you are allowed to choose “how” to control them, on the contrary, the arguments can only behave in a particular way, which is to form Cartesian products of any element from their respective types.

```
(x, y)                Any Cartesian product can be constructed of type `(nat, nat)`
```

If you are restricting the arguments in any way, then you are no longer talking about the same function. What you are doing is adding input constraints, which in path semantics changes the meaning of the function. This is why it is called “path semantics”.

For example, assume that the two arguments are equal:

f(x, x)

This is the same as adding the following cross-argument constraint:

f{[eq] true}

This is a partial function, while `f` is a total function. The partial function is not unique in the function space `nat × nat → nat`:

f[id] <=> {f}
f{[eq] true}[id] <=> {f_0, f_1, ...}

Any function that returns `2x` when the arguments are equal is included in the path set. Since the path sets are not equivalent, it means the two functions are not identical.

Why does this matter? Because when you define a probabilistic existential paths for a class of functions, adding any domain constraint will change the prediction of probabilities.

If you have a higher order path semantical function like this (just making up something):

∧f := \(f : nat → nat) = \(x : nat) = (∃f)(x) ∧ (∃f{[even] true})(x)

You are allowed to insert a domain constraint, but you must do it everywhere in the expression. A shorter way of writing the function above is:

∧f := \(f : nat → nat) = ∃f ∧ ∃f{[even] true}

Inserting a function `inc` with domain constraint `{(< 20)}`:

∧inc{(< 20)} := ∃inc{(< 20)} ∧ ∃inc{(< 20)}{[even] true}
∧inc{(< 20)} := ∃inc{(< 20)} ∧ ∃inc{(< 20) ∧ [even] true}

Higher order path semantics can only be possibly consistent when domain constraints are quantified over for the whole expression. This includes any attempt to “restrict” or “control” the arguments in any way.

If you want to restrict the input in higher order path semantics, you need to do this on the argument type. For example, the following is valid:

∧f’ := \(f : nat ∧ (< 20) → nat) = ∃f ∧ ∃f{[even] true}

Notice that `∧f’` is not logically equivalent to `∧f`.