Solvable Equations = Functions

by Sven Nilsen, 2017

One of the long standing conceptual gaps between path semantics and classical thinking is about the relation between equations in general and functions. The problem was whether equations are expressing a superset of the space of functions or not. Path semantics builds everything from the space of functions, so the answer to this problem got a lot to say about the power of path semantics. It turns out that all solvable equations, that is equations that has at least one solution for all variables involved, are statements about at least one function. Therefore, the power of path semantics scales to arbitrary solvable equations, and non-solvable equations are equivalent to proofs about non-existence. This closes the gap between path semantics and classical thinking and puts path semantics forward as a refined semantics of equations.

An equation is solvable when there exists some values for all variables that makes the equation `true`:

$$\text{solvable}_0 := \backslash(e : \text{equation}) = \exists\ a : A_{in}\ \{\ e_{in}(a)\ \}$$

$$A_{in} = A_0 \times A_1 \times \ldots \times A_{n-1} \times A_n$$
$$e_{in} : A_{in} \to \text{bool}$$

The definition above is what happens behind the scenes of classical thinking about equations.

In the path semantical version of equations, an equation is solvable when a higher order function $`e_{i \to n}`$ constructs a function returning a non-empty type:

$$\text{solvable}_1 := \backslash(e : \text{equation}) = \exists\ j : \text{nat} \wedge (< n)\ \{\ e_{i \to n}(j) : A_n\ \}$$

$$e_{i \to n} : \text{nat} \wedge (< n) \to () \mid A_n$$
$$A_n = A_0 \mid A_1 \mid \ldots \mid A_n$$

The behavior of the higher order function $`e_{i \to n}`$ must be such that the two definitions are the same:

$$\text{solvable}_0 <=> \text{solvable}_1$$

To describe the inner workings of $`e_{i \to n}`$ without circular reasoning, one needs to use a synchronized selection function `ɣ` with an injection product of the returned value that satisfies:

$$\forall\ j : \text{nat} \wedge (< n)\ \{\ \exists\ a : A_{in}\ \{\ e_{in}(ɣ(j, a) \times_j e_{i \to n}(j))\ \}\ \}$$

$$ɣ : \text{nat} \wedge (< n) \times A_{in} \to A_1 \times \ldots \times A_n \mid A_0 \times A_2 \times \ldots \times A_n \mid \ldots \mid A_0 \times A_1 \times \ldots \times A_{n-1}$$

The precise behavior of `ɣ` is such that:

$$\forall\ j : \text{nat} \wedge (< n)\ \{\ ɣ[id^n \times_j \text{unit} \to id^{n-1}] <=> id^{n-1}\ \}$$

This concludes the path semantical definition of a solvable equation.

This mechanism is a kind of erasure of all knowledge about any variable inside the equation and then asking which value we can give it. If the variable can have multiple values, then `$e_{i \to n}$` picks one of them. No matter what variable that gets erased, `$e_{i \to n}$` is capable of restoring the same or another value such that the equation is satisfied.

Instead of searching through every value of type `$A_{in}$`, the search is through every value of type `$A_{in\text{-}j}$`. Because this is a smaller type, it means that the path semantical version is a more efficient representation of the semantics of an equation compared to the classical one.

Even though `$e_{i \to n}$` is not unique for equations with multiple solutions, all possible constructions of `$solvable_1$` are logically equivalent.

It is not needed to include these complex sub-types in the `$solvable_1$` definition, because it is guaranteed by the construction of `$e_{i \to n}$` having the right properties. Since `e : equation`, one can choose to use `$e_{i \to n}$` to mean the full definition above.

Demonstration

In the classical thinking of mathematics, which evolved centered around the concept of equations, one is interested in finding out whether there exists some value for all variables such that the equation is true.

For example:

$$y = x + 1$$

The concept of a function emerged not until the 17^{th} century, and yet it took yet another 200 years to get a precise definition of a function. The invention of computers and functional programming has shown how powerful and useful functions are, because unlike solving equations, functions are easy to compute.

A mathematician thinking in terms of functions would write the equation as:

$$f(x) = x + 1$$

The value `y/f(x)` is uniquely determined from the function input. This is the case for all function outputs. However, in an equation it is not always the case that all variables are uniquely determined:

$$y^2 = x$$

Solving for `y` gives us two solutions:

$$y = sqrt(x)$$
$$y = -sqrt(x)$$

From a functional perspective one can construct two functions:

$$f_0(x) = sqrt(x)$$
$$f_1(x) = -sqrt(x)$$

In the classical thinking of equations, one must find a pair of values such that the equation evaluates to `true`, for example:

$2^2 = 3$	wrong	(2, 3)
$2^2 = 4$	right	(2, 4)
$3^2 = 9$	right	(3, 9)

In the path semantical thinking of equations, it is not required to look up a pair of values. In fact, one does not need to look up *any values at all*! Instead, it suffices to isolate one variable on one side and this works as a proof that the equation has at least one value per variable that satisfies the equation. Since the `x` variable is already on one side, it is already proved the equation has at least one solution:

$$y^2 = x$$

This is because `x` is a function of `y`:

$$y^2 = f(y)$$
$$f(y) = y^2$$

One can pick any value as the input of the function and it will produce a value, since this is proved by the *construction of a function*.

Some people might get confused by the path semantical definition, because did it not say that “for all `j`, the `$e_{i \to n}$` function must behave such as such”? Should this not mean you need to solve the equation for all variables to prove it? Take a look at this part of the definition:

$$\forall\ j : nat \wedge (< n)\ \{\ \exists\ a : A_{in}\ \{\ e_{in}(\gamma(j, a) \times_j e_{i \to n}(j))\ \}\ \}$$

It is true that `∀` is used here, but is NOT the definition of a solvable equation! The definition was:

$$solvable_1 := \backslash(e : equation) = \exists\ j : nat \wedge (< n)\ \{\ e_{i \to n}(j) : A_n\ \}$$

In the definition of ` $solvable_1$` there is only `∃`, so although `∀` is required for the total `$e_{i \to n}$`, the whole statement is proved by construction, simply by isolating a variable on one side.

The cool thing is that since one constructs a function, one can erase any input and still recover it.

$$f(2) = 2^2 = 4$$
$$f(y) = 4$$
$$y : [f]\ 4$$
$$4 : [\exists f]\ true$$

In path semantics the sub-type `[f] 4` type checks if the existential path returns `true` for `4`. However, since `4` was retried by evaluating `f` for some input, it means that the existential path must return `true`. Therefore, the sub-type will type check, and since it type checks one can recover any erased input. This means that the total `$e_{i \to n}$` is defined although only a partial one was constructed.

In practice, people tend to use this trick of evaluating the function to fill out the pair without realizing it, so one could say path semantics formalizes what humans already understand intuitively.