The Empty Space of Mind

by Sven Nilsen, 2018

In this paper I want to discuss ideas related to something that is usually taken for granted in general reasoning, but which is actually made as a hidden assumption that has become so common that it is very hard to point to it out for those lacking the mathematical background to analyze it. The phenomena in question is how some mind can think of something requires a symmetry. For every thing that can be thought about, there must be possible to assign the mechanism to another thing to think about. The very meaning of defining what one thinks about depends on the ability to think about other things. I show that a such general ability is the construction of arbitrary functions of type `nat → T` where `T` is some arbitrary type. Hence, the ability to think about things can be thought about using the framework of path semantics. This means the concept of an "empty mind" can be studied formally.

The framework of path semantics puts the interpretation of functions as the central theme of understanding how and where mathematical meaning comes from. Functions have certain properties and there are different kinds. For example, there exists deterministic functions and indeterministic ones. One can say various things about functions and, thanks to probabilistic paths, formalize any claim about functions with a well defined meaning in probability theory.

However, from the theory of path semantics alone, it might not become clear *why* it works. The motivation for using functions, instead of equations, as the central mathematical object, seems rather mysterious to the outsider. In this paper I will try to explain exactly what motivates the use of functions. Hopefully this will make it easier for people to understand path semantics in more depth.

A modern computer consists of bits of memory which can be turned on/off. Each bit can be addressed, which makes it possible to manipulate the memory of the computer using program instructions. The ability to change bits using algorithms and preserve their value over time is *the* fundamental property of a computer.

Still, a computer would not be very useful if it could not represent symbols with bits. The only reason we use computers is because the information that it processes has relevance for the real world. When a human watches a computer screen, the pixel colors are interpreted by the human brain. For example, an action movie shown on a screen would have no meaning without being seen by a human. Or, a program controling an industrial process, has meaning which depends on the actual process happening out there.

It turns out that even there are countless ways of defining meaning of symbols, there exists a general ability of assigning such meaning that is isomorphic to a single concept: The ability to create arbitrary functions of type `nat → T` for some arbitrary type `T`. From an intelligent reasoning perspective, it is assumed that one can apply general knowledge of path semantics. Under most practical applications, this is an unrealistic assumption, but it provides a way of grounding beliefs about semantics.

The type `nat` stands for natural number. The smallest natural number is 0, the second smallest natural number is 1, and so on up to, but not including infinity. Since this is an infinite type, one can not construct abitrary function of type `nat → T`, because a such decision procedure would require infinite amount of space and the instructions would contain all information needed to describe the function. With other words, if a such system existed, it would be useless and not actually doing anything.

However, by pretending a such system exists while being useful, one can imagine this as a sort of ability. In the theory of Zen Rationality, a such ability is formalized in Naive Zen Logic as imagining a smarter version of your own mind. Instead of assiging equal weights to every function, one tends to "focus" the ability on relevant information for one's current mind. There exists a such mind for every purpose, so while it is impossible to construct a general system for arbitrary functions, every *purpose* has the analogue of a smarter mind and therefore the general ability of constructing functions of type `nat → T` is used relatively for arbitrary minds.

Hence, the address space of a computer starts out empty, not necessarily in the configuration sense of the word, but how meaning is assigned relative to some mind. It is sufficient to describe a such state as totally undefined:

undefined := \(_ : nat) = "undefined"

undefined : nat → string

The mind has some internal type system which represents the analogue of interpreting the symbols stored in computer memory. Philosophically, this type system might exist separately from the computer, because in the real world meaning is not defined in some formal language, but by the inner workings of an agent. One reason to use a type system in this abstract way is to permit reasoning about how the agent reasons about symbols, since the behavior of most agents are too complex to describe. Instead of using an implementation of an agent to study how symbols are interpreted, the mathematical isomorphism based on functions of type `nat → T` is used.

Once this connection is realized, it becomes possible to study meaning through the framework of path semantics. A predictions that the agent does corresponds to a function `h` determined by a function map generator `$g_{i \to n}$` of a function executed by the computer `f`:

$f[g_{i \to n}] <=> h$

$f : nat \to nat$

For example, if the agent reasons about whether the user presses a button, a function can be defined which returns `true` if the button is pressed and `false` otherwise. This function takes the whole computer memory as input, represented as a natural number, and returns a type `bool`:

$g_0 : nat \to bool$	Returns `true` if a button is pressed, `false` otherwise

Another function might return `true` if a pixel on the screen shows a blue color:

$g_1 : nat \to bool$	Returns `true` if a pixel shows a blue color

For some state of change in the executing program `f`:

$f[g_0 \to g_1] <=> h$

This means that `h` predicts the future state of the blue pixel from the press of the mouse button. A such prediction might be impossible to make, in which case one says "the path `h` does not exists".

Probabilistic paths on the other hand always exist, but computes the likelihood of the pixel showing a blue color when the user presses a button. This requires a sophisticated formula which I will not demonstrate in this paper.

Path semantical notation can be translated into an equation:

$f[g_0 \rightarrow g_1] <=> h$	Path semantical notation
$g_1(f(x)) = h(g_0(x))$	Equational form

The reason path semantics uses functions so heavily instead of equations is because of readability and the equational form is much less constrained. It might be very hard to interpret what an equation says. Path semantical notation makes it easier to understand proofs and avoid mistakes.

In principle, anything that a computer does might be analyzed by a mind this way. The mind is thinking about what the computer does. Since this happens by using functions to study functions, a mind can also replace parts of its own with computation. This means that a computer can implement a mind thinking about what is being computed.

The number of paths is too large for any practical computer to analyze. Therefore, an agent must select what to think about.

It follows that an empty space of mind is an agent not thinking about anything. It does not define any functions predicting future outputs of its thoughts.

Strangely, a computer can still compute without its output being analyzed. An agent can let its mind run habitual programs without reflecting on them. One says that the mind of the agent is empty, even though lot of computations might be performed.

The difference between an empty mind and a non-empty one is that a non-empty mind requires reflection having potentional of leading to self-modification of the running computer program.

Formally, an abstract reflection operator `<?>` and an abstract modification operator `=|>`:

$$(f[g_{i \rightarrow n}] <?> h) => (f =|> f')$$

Where `=>` means here something represented in the decision algorithm of the agent. If this reflection happens by testing some predictions, the agent will self-modify in a certain way. The next time the agent thinks about itself in the same way, `f'` has taken the place of `f`:

$$f'[g_{i \rightarrow n}]$$

All minds with no potential for self-modification are empty. Such minds have only one function `f` that they compute over and over until something external disrupts the computation, or the computer breaks down.

From this it follows that it is strictly harder to think about agents who can self-modify, because one would have to predict the result after several modifications. An empty mind can not think about a non-empty mind in great detail. There exists a natural hierarchy of complexity of such agents.