Introduction to Quantum Programming and Semantics: Categorical semantics

Chris Heunen

Spring 2024

Suppose `F` and `G` are two fragments of code, and consider the two computer programs:

$$P = (\texttt{if 1 = 1 then F else G})$$
$$Q = (\texttt{if 1 = 0 then F else F})$$

Are P and Q the same programs or not?

Syntactically, they are clearly different: the code fragment P is different than the code fragment Q. But do P and Q compute the same? The answer depends on how fine-grained you are willing to look. From the perspective of machine *operations*, a (nonoptimising) compiler will make P and Q into different executables, because the machine will be storing `G` somewhere in memory when executing P, even though it is dead code that will never be reached, but not so when executing Q. But from perspective of the user, P and Q *behave* the same. They give the same output on every input. They *denote* the same algorithm, namely `F`.

This little analysis we have been doing is called *semantics*. We assigned to the two syntactical fragments P and Q their meaning, in the form of some mathematical objects $[\![P]\!]$ and $[\![Q]\!]$. On the first level, *operational semantics*, that mathematical object encoded all implementation details, so that we could retain the difference between the dead code. On the second level, *denotational semantics*, that mathematical object was a set-theoretical function that transforms input into output, and we didn't care how exactly the function performed that computation.

Denotational semantics is used to show that two programs implement 'the same' algorithm. For example, there are many ways to sort an array, but as long as we don't care about how fast the implementation is they all do the same. Quicksort might be faster than bubble sort, but they both sort. In other words, denotational semantics is used to prove that programs meet their specification, that they 'do what they should do'.

The example with P and Q above was insultingly simple, but you can imagine it gets a lot more complicated when for example recursion is in play. The important thing is that the assignment $P \mapsto [\![P]\!]$ *preserves structure*. For example, if we want to prove something about *concatenating* program fragments, we should have some operation that concatenates their denotations:

$$[\![\texttt{F; G}]\!] = [\![\texttt{G}]\!] \circ [\![\texttt{F}]\!]$$

Similarly, recursion requires us to talk about substitution (calling subprograms with some input) in the syntax, so the semantics had better have some notion of *function space*

$$[\![\texttt{F(X)}]\!] = [\![\texttt{F}]\!]([\![\texttt{X}]\!])$$

where $[\![\texttt{F}]\!]$ can live. For a third example, if the programming language describes concurrent computation, there should be some sort of *parallel composition* of denotations:

$$[\![\texttt{F par G}]\!] = [\![\texttt{F}]\!] \otimes [\![\texttt{G}]\!]$$

The trick is to choose the mathematical object in which the denotations $[\![\texttt{F}]\!]$ live cleverly. It should have the same structure as the programs you're analysing, but abstract away from all the unimportant details, so

that powerful mathematical theorems can be used. Popular choices are λ-calculus or partially ordered sets. We will use *categories* as our semantics, which subsumes both.

There is another reason we will use categories. Above we analysed code in an actual programming language. If we consider *quantum* processes, there is no such neat description. The systems are just black boxes that we cannot look inside. But we can still see how the boxes behave; that is precisely the empirical method of physics! Analogously, in computer science, if I give you two executables instead of their source codes, can you still say whether they implement the same algorithm? No, because then you could solve the halting problem.

But you can still analyse the structure of the two computations using denotational semantics. You can see how the *information flows* from input to output and how it is recombined in the process. Categories support this kind of bookkeeping in a beautiful way. As we will see, instead of boring algebra (like in λ-calculus or partially ordered sets), we can manipulate categories graphically completely rigorously, in a way that can even be automated.

In fact, in this way category theory itself becomes a programming language of sorts, where we can click together subroutines in various ways to compose larger programs. In fact, it is unclear what a quantum programming language should look like: if you cannot copy or delete information, you cannot push and pop things on the stack, how do you handle recursion, or even if-then-else statements? Therefore, investigating the categorical semantics first is in a sense the only way to inform the design of a good *quantum programming language*.