The What and the How

August 26, 2019

Logic Programming uses a logical system as a programming language. How is this different to using a logical system as a specification language? Can the two activities co-exist in one language with the same semantic theory?

1 Specification and Programming

There is an intuitive distinction between "the what" and "the how". The former informs us what is to be achieved: the goal of any enterprise. In contrast, the latter offers a plan, a way of achieving the goal. In computer science, this distinction reflects the difference between specification and programming: a specification tells us *what* is to be programmed and programming tells us *how* to do it. But how well does this distinction stand up to the various paradigms of programming. Is it always present and always so sharp?

2 Imperative Programming

Perhaps the paradigm where this distinction is most clearly present is the imperative one. A familiar and simple example concerns the greatest common divisor (GCD) of two numbers:

- A number z is the *greatest common divisor* of two numbers x, y if z divides both and is the biggest one that does.

This is a definition of the GCD: it does not tell us how to find the greatest common divisor; it only tells us what it is. The greatest common divisor relation

$$Gcd(x, y, z),$$

is more formally defined by the following predicate calculus expression:

$$Divides(z, x) \land Divides(z, y) \land \forall w \bullet Divides(w, x) \land Divides(w, y) \rightarrow w \leq z.$$

In contrast, the following program, which is written in the WHILE programming language encodes Euclid's algorithm for computing the GCD.

```
var     x,y,r: nat
begin
read x,y;
        r:=x mod y;
        while r notequal 0 do
            x:=y; y:=r; r:=x mod y
        od; write y
end
```

It is here that the distinction between "the what" and "the how" is the most explicit. How does it pan out with respect to other paradigms?

3 Functional Programming

The functional style moves us away from the imperative genre dictated by physical devices and moves us to a platform where everything resembles the definition and application of mathematical functions. To illustrate matters consider the following definition of Fibonacci.

```
Fib(0)=1
Fib(1)=1
Fib(n+2)=Fib(n)+Fib(n+1)
```

This is a definition of the function, and taken as a specification it tells us "the what". But it is not a program in a programming language until such mathematical expressions are given an implementation as part of a programming language with a semantic interpretation and an implementation. Now one might suggest that the mathematical reading provides its semantic interpretation. But consider the following implementation in Haskell.

```
fibs :: [Int]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

This is lazily evaluated. And generally, the user needs to know this in order to program in the language. This is not part of the mathematical interpretation which does not refer to order of evaluation. It is an axiomatic definition of a function. And this is the important distinction. Mathematical definitions of functions are to be read axiomatically and provide no mechanism of evaluation. Lazy evaluation allows the language run-time to discard sub-expressions that are not directly linked to the final result of the expression. It reduces the time complexity of an algorithm by discarding the temporary computations and conditionals etc. etc. None of this concerns the specification of the intended function which is defintional and axiomatically given. So there is still a fundamental distinction between specification and programming. Has the Logic paradigm manged to bridge the gap?

4 Logic Programming

Robert Kowalski at a special meeting of the Royal Society in London emphasises the relationship between logic programs and specifications. He observe that the only difference between a complete specification and a program is one of efficiency. As an example, Kowalski gives a specification of sorting.

$$Sort(x, y) \triangleq Perm(x, y) \wedge Sorted(y)$$

He then gives the following sorting program:

```
sort(X,X) :- ordered(X).
sort(X,Y) :- I<J, X[I] > X[J], interchange(X,I,J,Z), sort(Z,Y).
```

Given X, the program computes the sorted version Y by repeatedly exchanging some X[I] and X[J] that are out of order. The program is highly nondeterministic: the condition X[I] > X[J] is the only constraint on I and J. To regard this as a useful sorting program we must further constrain I and J, to reduce the search drastically.

Once again there is a fundamental distinction between specification and programming in that in order to write realistic programs, the user must know the operational semantics of the language and take advantage of it.

Is *Pure logic programming* , where there is no distinction between specification and code, practical feasible? And complexity issues are not just practical concerns but have epistemological significance: we must know the mechanisms of evaluation if we are to write realistic programs.