Similarity in Computer Programs

Abstract

Recent debates over software patents raise questions about the nature of computer programs, algorithms and other kindred notions. In order for patents and other intellectual property mechanisms to function, principles of similarity are needed. In this paper I begin to explore several possibilities in this direction. The paper is intended as an introduction and "consciousness raising" about this area of possible research and no firm results are expressed.

Introduction

This paper consists of four principle sections. First, I explain the commercial and technological motivation as the origin of this topic. Second, I discuss philosophical relevance to the work, focusing on the concept of similarity. Third, I propose several principles of similarity and discuss their merits and demerits. No firm conclusions are reached but the directions I would like to take this work in are indicated in the fourth and final section.

Part 1 - Concrete Application and Motivation

This project initially grew out of an undergraduate presentation done many years ago for a philosophy of science seminar. Since then it has been resurrected by the continued interest in software patents (e.g., the conference "Software Patents: A Time for Change?" in 2006) , intellectual property, and the philosophical debates over the nature of computing (see, e.g., Davis 2006; Douglas 2003). It is my hope that this research will eventually yield some results which may bear on some of these concerns.

Concretely, recall the SCO vs. IBM lawsuits from a few years back (see, e.g., Raymond and Landley 2004). At issue was whether IBM, a licensee of some of SCO's source code for use in AIX, had illegitimately rolled into an open source project (Linux) some of SCO's code. Source code, the human-readable form of a computer program - one might say (I will return to this issue in section 2, below) - is bound to be similar in similar contexts. This judgement (used by IBM in their defense) is a prima facie one based on a principle something like "similar tools for similar jobs". But how much similarity is needed to prove infringement of intellectual property law, and in what respect?

Part 2 - Philosophical Relevance and Some Initial Philosophical Preliminaries

Many areas of exact philosophy have been used in computing. For example, AGM (e.g. Gärdenfors 1988) style belief revision work is being used in databases, the Wand-Weber tradition (Wand and Weber 1990) in data modeling stems from the exact metaphysics of Bunge (1977), and Turing's seminal 1936 paper which I trust is relatively familiar to all. Philosophy of computing itself is a growing field, with many conferences (e.g., the now annual Computing and Philosophy conferences) and publications (e.g., Bynum and Moor 1998). The current work lies at the intersection of these two lively areas and traditional metaphysics. It also has a connection to the work on analogy in cognitive science and AI (e.g. Hofstadter *et al* 1995) particularly in understanding our judgments of similarity. An area of rich possibilities indeed.

But first, some initial philosophical and methodological worries which must be addressed before delving into a few of the suggestions I have tried to use to solve the problem of this paper.

I am assuming that we all know correctly what computers are. They are, for purposes of this paper, those boxes that sit on our desks and floors, that we carry around like my iBook, etc - things that the so-called computer industry sells. There may be more things which are computers than those, but if there are I am stipulating that they are off topic in the interests of making this paper manageable.

However, life is not so simple when it comes to computer programs. What's a computer program? For purposes of this paper I am going to stipulate that a computer program is its source code and restrict my discussion to what might be called functional similarity. This decision is again in the interests of reducing the scope of consideration to start with. (As it happens, many of the concerns I raise apply also to object code, user interfaces [in spades!], what might be called feature similarity, and other potential areas of analysis.) By selecting this area as the object of my study I also do not mean to suggest that source code is some how the most important "part" (or not) of computer programs. It is here we can investigate the first area of genuine puzzlement.

Part 3 - To Work

Let's start by assuming also that proper parts[1] of computer programs can be meaningfully compared. (Again, if this is denied, life becomes more complicated.) So let's compare the two C functions[2] in Listing 1A and 1B, which are, I will assume, parts

[1] I am using "part" here informally. A mereology of programs would be one possible further development for the current work, but as we shall see it is of necessity quite far in the future.

[2] I have selected C for these examples for two reasons. One is that it is a widely known programming language. Second, as we shall see, there is a good philosophical (of sorts) reason to avoid pseudocode.

of two computer programs and that every other aspect of them is equivalent in some respect so that their similarities are otherwise maximal.

```
/* Listing 1A */
long fact (int n)
{
	int i;
	long total = 1;

	for (i = 1; i <= n; i++)
		total *= i;

	return (total);
}
```

```
/* Listing 1B */
long factorial (int n)
{
	int j;
	long tot = 1;

	for (j = 1; j <= n; j++)
		total *= j;

	return (tot);
}
```

A naive comparison of source code would probably simply complement the count of the places at which these fragments differ as a proportion of the length of the longer of the two programs. This yields (ignoring the comments): 1 - 16 / 105 x 100% = 84.8% similar. But, renaming identifiers - as that is all I have done - in some source code surely does not count as a different program. (Think of student work being analyzed for cheating, for example[3].) So, I think it is fair to say that most people would regard these two as identical, because of some appeal to "role" or "purpose" or "function" in a non programming sense. Similarity has as limiting case some sort of identity (or rather, similarity of a and b is maximal when a is identical to b), so this is good news.

[3] This suggestion does prompt the realization that similarity judgments are relative to a particular purpose. I do not consider this possibility further here.

But good news sometimes does not last, as is the case here. Consider listing 1C.

```
/* Listing 1C */
long factorial (int n)
{
       if (n == 1)
              return (1);
       else
              return (n * factorial (n - 1));
}
```

This function also returns the factorial of its parameter. But here the the function calculates by recursion, rather than by iteration. There are several possibilities here. One could take the approach that functions which compute the same function are equivalent, in which case the similarity between 1C and either 1A or 1B is 100%.

This would suggest the following equation:

$$S_{ab} = \frac{\overline{IO_a \cap IO_b}}{\overline{IO_a \cup IO_b}}$$

where S_{ab} is the similarity between a and b, IO_a is the set of ordered pairs <input, output> for function a, IO_b is the set of ordered pairs <input, output for function b> and the bar represents cardinality of a set. Thus the function ranges between 0 for no similarity to 1 for complete similarity.

There are two potential problems with this approach. Computing the identity of functions is a computationally (or recursively) unsolvable problem (Kozen 1997), so whatever one might think about human cognition, a computer can't calculate its programs' similarity by this measure. Also note that this makes quite clear the assumption that we are dealing with concrete implementations (and not algorithms understood "abstractly" or

"platonistically"[4]). Because of this, the source code then does not (by itself) determine the similarity! In the case of C programs, the size of the data types long and int are implementation defined, whence the equivalence of these programs is not merely a matter of source code. It is for these reasons of implementation that writing my examples in an actual language, not pseudocode, becomes important. Of course on any particular implementation the sizes are fixed, but I didn't say that we were fixing the implementation across source code listings (cf. footnote 1).

Alternatively, one could argue that since the overhead of 1C is much higher (invoking all those function calls, etc.) that it should be regarded as different in some way. But here we lose any obvious measure of similarity. Perhaps one should simply say that listings 1A - 1C simply differ in one place, and not worry about percentages. Fair enough, but note two things. One is that this assumption of difference again presupposes something beyond the source code. I suspect that a good optimizing compiler could "unroll" the loop in 1C and thus actually eliminate the potential inefficiency. Second, we run into another problem. Consider listings 2A and 2B (purists should also assume that

---

[4] Dean (2002) has argued that algorithms have to be viewed as implemented for our usual discussions of them in terms of complexity classes and other features to make sense. (For example, we assume fixed cost of access to memory.) Note, however, that even regarding algorithms as implemented does not fix the answers to the questions of current consideration because said answers only arise by further narrowing the class of implementations. That is, we not only assume single processor systems, with fixed access cost to memory, etc. but we need to know how the compiler turns source code into object code, etc. Cf. also Yanofsky (2006) where the analogy `program : software engineer :: algorithm : computer scientist :: recursive function : mathematician` is suggested.

appropriate typedef struct / typedef enum declarations are found somewhere else and are identical in both cases):

```
/* listing 2A */
void QuickSort (Array A, int m, int n)
{
     int i,j;

     if (m < n)
     {
          i = m;
          j = n;
          Partition (A, &i, &j);
          QuickSort (A, m, j);
          QuickSort (A, i, n);
     }
}

void Partition (Array A, int *i, int *j)
{
     Type Pivot, Temp;
     Pivot = A[(*i + *j) / 2]; /* This QuickSort uses middle item as pivot */
     do
     {
          while (A[*i] < Pivot) (*i)++;
               /* find leftmost i such that A[i] ≥ Pivot */
          while (A[*j] > Pivot) (*j)++;
               /* find rightmost j such that A[i] ≤ Pivot */
          if (*i <= *j) /* if didn't cross, swap */
          {
               Temp = A[*i]; A[*i] = A[*j]; A[*j] = Temp;
               (*i) ++;
               (*j) --;
          }
     } while (*i <= *j); /* while they haven't crossed each other */
}

/* Listing 2B */
void BubbleSort (Array A)
{
     int i;
     Type Temp;
     Boolean NotDone;

     do
     {
          NotDone = false;
          for (i = 0; i < n-1; i++)
          {
               if (A[i] > A[i+1])
               {
                    /* exchange A[i], A[i+1] to put them in sorted order*/
                    Temp = A[i]; A[i] = A[i+1]; A[i+1] = Temp;
                    /* since we swapped, need another pass */
```



```
                    NotDone = true;
               }
          }
     } while (NotDone);
}
```

Here we have a clear difference in (best / average case) running time (which cannot be optimized away) for the two functions which would produce the same I/O pairs[5]. They are, after all, both array sorting functions. Naively comparing source code still seems inapplicable. But perhaps one should compare running times. After all, that is what we are told when we first learn about algorithms and data structures. Quicksort runs in (best / average case) O(n log n) and Bubblesort O (n^2) where n is a fixed size of array. (I'll assume it as a parameter, since input to a program isn't at first glance part of its program.) We have a qualitative measure of similarity here, then. Intuitively, then, Bubblesort is more like Quicksort than it is like Stoogesort (the sorting algorithm which works by randomly permuting the elements until they are in order), since they are much closer in running time. We even can use the usual set theoretic inclusion operator on the complexity classes (see Cormen, Leiserson and Rivest 1996) and thus have an ordering relation usable in comparison of similarity. But this doesn't allow us to compare degrees of similarity, merely provides us with just an ordering for intuitively alike (parts of) programs. One might think one could use a chart like Figure 1, adapted from Standish (1995):

[5] Assuming, of course, that we only count the array A as the input in both cases; otherwise literally the functions have different inputs. This could be accomplished by using a "helper function" which would have no purpose other than to enforce this similarity. But that would probably be rightly regarded as a case of "moving the bump around under the rug".

Figure 1: Algorithm stops in f(n) microseconds

f(n)	n = 2	n = 16	n = 256	n = 1024	n = 1048576
1	1	1	1	1	1
$\log_2 n$	1	4	8	10	20
n	2	16	256	1024	1048576
$n \log_2 n$	2	64	2048	10240	20971520
n^2	4	256	65536	1048576	1.10×10^{12}
n^3	8	4096	16777216	1073741824	1.15×10^{18}
2^n	4	65536	1.167×10^{77}	1.80×10^{308}	6.74×10^{315652}

The idea would be to simply count how far away one function was from another on the chart. This suggestion fails for three reasons. One is that the asymptotic notation does not allow comparison between functions of the same class (e.g. Mergesort and Quicksort have the same asymptotic running time; see Standish 1995). Second, more importantly, is that in general no such chart is ever complete. Granted, any plausible approach to any particular function is likely to fall into a few fixed complexity classes, but it is difficult to determine what these are, *a priori*. Matrix multiplication makes this point rather nicely. The usual approach (form all the dot products of the row vectors of one matrix by the column vectors of the other) has $O(n^3)$ running time (for a suitable parameter n). Even so-called "block multiplication" (which reduces the number of multiplications of scalars) only reduces by a constant factor, so this approach is still in the same complexity class. Instead, with details I need not go into, it is possible using

the so-called Strassen algorithm to reduce the running time to $O(n^{\log 7})$ (Cormen, Leiserson and Rivest 1996).

Third, note that simply counting complexity classes fails to capture some similarity because of the extreme difference in actual running time. (This is why the base of the logarithms, normally ignored in complexity class calculations, is included in figure 1 - to make the running times concrete.) In other words, running times suggest that some complexity classes are "closer together" than others. Moreover, there are "gaps" which one cannot fill. It is easy to show that the exponential class includes (is "larger than") all classes of the form $O(n^k)$ for any finite, fixed[6] k. So if we wanted to somehow count (say by postulating that complexity similarity = complexity class i - complexity class j) the number of classes in between two classes as a measure of "complexity similarity" this would be infinite when comparing any polynominal to an exponential class. This is problematic for two reasons. First, since the same "infinite distance" would apply between multiples of logarithm functions and polynominals (after a certain size, anyway) comparison of sorting algorithms (a common task) would often be unhelpfully illuminated, as many of them are either in class $O(n^2)$ or O(n log n). Second, since the infinite distance would apply in several "places" in the complexity hierarchy, any comparison at least of the form I suggested across two such "places" would equal the difference across one, which seems a bit unhelpful.

[6] That is, k is not a nontrivial function of n.

Before concluding, I would like to anticipate one objection to the approach I have taken to my topic. Namely, that I should have taken similarity as primitive in some axiomatic system and then examined several postulates to constrain its behaviour. I agree that ultimately this should be worked on. I think such an approach is rather useless until we get a feel for what the axioms should capture. I do intend that one be ultimately able to make use of the principles, after all. Do we even know that an ordering is possible? I have assumed throughout that all programs are comparable. This assumption is found also in the brief presentation of similarity axioms in Smith (1998) which makes the otherwise plausible assumption that dissimilarity is "structurable" within a metric space. Perhaps this assumption, too, is false[7]. If it is false, an initial step in future investigations would be to figure out which classes of programs are comparable.

Section Four - Conclusion

I have now introduced several means of comparing for similarity of computer programs and found them wanting, without even leaving the discussion of source code. As I mentioned in passing earlier, there are many more such problems, in areas of human interface, features, and even as pertains to object code. Hopefully, then, we are now more enlightened about the potential difficulties and the rich possibilities for work in this

[7] Tversky (1977) has long argued that our similarity judgments fail to be describable in terms of the axioms of triangle inequality and symmetry. However, as with Smith's work, it is not clear to what extent this purely descriptive analysis of our judgments of similarity applies to any postulates for use in better **making** and **using** such judgments or to what extent the failure occurs in a restricted domain such as computer programs. Tversky's positive proposal uses various forms of feature matching, which also holds promise, but presupposes we can identify the different features of a program, and the difficulties arise here again.

area. My own inclination is to make use of some of the techniques discussed in Tversky (1977) to systematize the findings of various experts (software engineers, computer programmers, etc.) to see if any structure or regularity can be discerned in the no doubt largely tacit views of experts. But this, and many other matters, will have to wait.

References

Bunge, M. 1977. *The Furniture of The World*. Volume 3 of Treatise on Basic Philosophy. Dortrecht: Reidel.

Bynum, T. and Moor, J. (eds). 1998. *The Digital Phoenix: How Computers are Changing Philosophy.* Oxford: Basil Blackwell.

Cormen, T., Leiserson, C. and Rivest, R. 1996. *Introduction to Algorithms*. Cambridge: MIT Press.

Davis, M. 1964. *The Undecidable*. Hewlett: Raven Press.

Davis, M. 2006. "The Myth of Hypercomputation." In *Alan Turing: Life and Legacy of a Great Thinker.* Christof Teuscher (ed). Heidelberg: Springer-Verlag.

Dean, W. 2002 "What Algorithms Could Not Be". Paper presented at Computing and Philosophy Conference, Carnegie Mellon University.

Douglas, K. 2003. *Super-Turing Computation: A Case Study Analysis*. Unpublished thesis, Carnegie Mellon University.

Gärdenfors, P. 1988. *Knowledge in Flux: Modelling the Dynamics of Epistemic States*. Cambridge: MIT Press.

Hofstadter, D. et. al. 1995. *Fluid Concepts and Creative Analogies*. New York: Basic Books.

Kozen, Dexter. 1997. *Automata and Computability.* New York: Springer.

Raymond, E. and Landley, R. 2004. "OSI Position Paper on the SCO-vs.-IBM Complaint." Available at http://www.opensource.org/sco-vs-ibm.html. Accessed January 19, 2007.

Smith, E. 1998. "Concepts and Categorization" . Printed in Smith, E. and Osherson, D. (eds.) 1998. *An Invitation to Cognitive Science*, *Volume 3: Thinking.* Cambridge: MIT Press.

Standish, T. 1995. *Data Structures*, *Algorithms and Software Principles in C.* Reading: Addison Wesley.

Turing, A. 1936. "On Computable Numbers, With an Application to the Entscheidungsproblem." Reprinted in Davis 1964.

Tversky, A. 1977. “Features of Similarity”. Psychological Review 84, 327-352.

Wand , Y. and Weber, R. “Mario Bunge's Ontology as a formal foundation for information systems concepts”. In Weingartner and Dorn 1990.

Weingartner, P. and Dorn, G. (eds). 1990. *Studies on Mario Bunge’s “Treatise”.* Amsterdam, Atlanta: Rodopi.

Yanofsky, N. 2006. “Towards a Definition of an Algorithm”. Available at http://arxiv.org/pdf/math.LO/0602053.