Programming Languages (CS302 2007S)

C.A.R. Hoare - Hints on Programming Language Design

Comments on:

Hoare, C.A.R. (1973). *Hints on Programming Language Design*. Stanford Artificial Intelligence Laboratory Memo AIM-224 / STAN-CS-73-403. Computer Science Department, Stanford University. (An extended version of a keynote address at the second SIGACT/SIGPLAN Symposium on Principles of Programming Languages.)

Comments.

Hoare mentions that simplicity of a language is much better than modularity in a language to keep programmers from invoking features of the language that they don't understand. However, modularity should be helpful if programmers stick to the parts of the language that they do understand and allow programmers with more experience and understanding use a broader selection of features. I feel like this is coming back to things we've already seen that try to limit the amount of screwups less experienced programmers can make by limiting functions for more experienced programmers.

Java is an example that has many characteristic/aspect against Hoare's options. such as splitting codes to files. This paper not only as Hoare says have many controversial points, but also some arguments are pool. e.g when he tries to argue P.L needs to be fast, listing the disadvantage of splitting codes doesn't make fast translation necessary. It only makes the method of splitting codes might not make the translation faster. Also, he compares programming language engineering to other engineering discipline. The first one has less than 50 years history, the second has much longer history. So the comparison might not be very helpful.

From my understanding nowadays C is not used to program anything other then Operating Systems, or other applications where getting every ounce of speed is absolutely critical. In almost every other application the advantages provided by the object-oriented paradigm outweighs their cost in slower performance and most complex languages.

The difference between the paper's conclusion and current programming seems to be caused by changes in the environment. Hardware became faster. This meant that the cost of slower performance due to abstraction, or inability to hand-optimize (such as in the story of Mel) brought on by programming concepts, like the object oriented paradigm, had less of an impact. Whether or not something is adopted depends on the costs and benefits of its adoption. Declining costs would mean that it is more likely the advantages outweighs the costs.

Another change in the environment seems to be that programming teams became larger. The author mentions that "the need for larger programming teams might provide the environmental discontinuity that languages like Smalltalk and C++ need." It seems that given the popularity of Java and C++ in modern times, this environmental change did happen.

Questions.

As a side note, the table data relating to the usage of different programming languages in 1992 seemed rather surprising to me. Is there a reason assembly is so common? I thought assembly was extremely hard to program with and that was why so much effort was spent developing "higher-level" languages.

Even in 1992, you coould worry enough about performance that you'd use assembly.

Would Scheme fit under Hoare's ideas of simplicity?

I think so. He seems to include LISP, and Scheme is an improved version of LISP.

On the last sentence of the second to the last paragraph of page 21, the author mentions "slave storage" what is he referring to?

I expect that it's a form of paging. We simulate larger memories than physical memory (that is viritual memory) by storing some things that seem to be in RAM on disk. Slave storage is one such technique.

Hoare says that one of the ways to enable fast translation of high-level programs is to allow machine code instructions interspersed with the lines of high-level code. After reading "Go To Statement Considered Harmful" by Dijkstra, I would object to this technique for the sake of readability. I envision a situation where even after programming language designers have done away with the go to statement, an obstinate programmer can still use the machine code version of go to at any point he deems fit resulting in convoluted code.

I expect that Hoare would not want you to inject code with non-local goto statements. Rather, the idea would be that you'd inject some hand-optimized code for efficiency's sake. But I agree with you, it does seem dangerious, particularly in light of the other arguments.

I thought the article was interesting and gave good ideas for programming design. Hoare also says that good language design should have simplicity, security, fast translation, efficient code and readability. All of the above seem fairly desirable to have in a programming language as Hoare suggest. However, Hoare did not in some case such as simplicity, security explain why programmers decided to deviate from these desirable traits. Am I right in guessing that programmers that did decide to deviate from the above traits would have weighed the advantages over the disadvantages?

Different decisions are made for very different reasons. I cannot say why people deviate, but it's not always because of reasoned work. For example, I talked to a student today who said "Sam, I know the code I've written is bulky and inefficient, but it's the first thing I could figure out."

Many of the constructs which C relies quite heavily on are cautioned against by Hoare for seemingly good reasons. Why is it then, that C style languages have become so popular?

Actually, C seems fairly simple, discourages use of things like goto, uses a Hoare-style case, limit coercion (but not as much as Hoare would like), and such. As Gabriel suggests, it's not clear why people choose languages, but it seems to rarely involve careful consideration.

Comments for classroom discussion.

Hoare states, "I fear that each reader will find some of my points wildly controversial". What points would have been considered "wildly controversial" at the time the paper was published.

From the hints that Hoare gives about, it seems that it would be almost unacceptable to have object-oriented programming languages around. He mentions that modularity does not mean simplicity, and readablity and program structures are important design considerations. All these seem to be against the object-oriented paradigm that was in an infant state when he wrote the paper. Yet, the OOP languages proved to be widely accepted in later years. Can this be explained by the acceptance model of Gabriel and the advantage of easy program debugging?

The second reading was very specific in its requirements for a good programming language. It was my impression, that he tought of the issue as too straight forward and cut dry. I always thought that there had to be some tradeoffs in designing a computer language. I understand his focus on simplicity, but is it not a tradeoff for the power of your language? I'm not sure what I'm trying to say here. But if everything was as simple as he said, why wouldn't people just do things his way? For example, what are the advantages of using pointers and references (they seem quite important and widely used in C)?

Followup Comments on Examples

I asked students to pick one of the code examples in this paper and then to explain or comment on it.

The Use of Whitespace in Fortran

```
DO 17 I = 1,10
```

and its possible misinterpretation as

```
DO17I = 1.10
```

In the syntax example from Fortran, Hoare uses a missing comma problem in a DO statement to prove how compliers need to detect and pinpoint errors accurately. Hoare shows how a missing comma might allow the compiler to treat the DO statement as a variable. Hoare argues that the position and length of each meaningful symbol should be known, allowing the compiler to immediately pinpoint errors.

Why would the compiler in Fortran allow spaces in variables? Is there a possible advantage to that? Doesn't that make interpretation of each line by the compiler and by humans much more difficult?

I haven't read enough of the history, but remember, people were figuring this out as they went along. My guess is that, when you're typing on a card punch, it's much easier not to have to replace a card when you've inadvertently typed a space. And, yes, it makes parsing much harder.

Switch vs. Case Statements

```
    switch SS = Ll, L2, L3;
      . . .
    go to SS[i];
Ll: Q1; go to L;
L2: Q2; go to L;
L3: Q3;
L:
```

vs.

```
case i of
{ Q1,
  Q2,
  Q3 }
```

By giving example of the structure for switch command, Hoare is trying to underline the importance of good structures in programming languages. He points out that the structure for switch in ALGOL 60 is inappropriate since it allows a programmer to jump into the structure from which has has already exited. In a way this also highlights the "harmfulness" of goto statements. Further he gives the example of his invention for switch structures to avoid the faults that he describes about the previous structures.

Hoare says "the control structure is not apparent from the form of the program, but can only be determined by a run-time trace" I am not sure what he means here.

Since the label array can be changed before the indexed jump, it's not really possible to figure outwhat statements are grouped in the jump. (Having watched C programmers for years, I would certainly expect someone to change the array of labels in different situations.)

Hoare he says the old switch allows programmer to jump back, because the switch is made by labels. He thinks programmers should not do it (vague pronoun). However, it's all up to programmers. I understand his suggestion also affects the art of programming.

The case statement enables the program to change the its flow based upon the value of i. If i == 1, then flow goes to Q1, if i ==2, then flow goes to Q2 and so on.

Since the case statement is a lot more concise than the switch statement, shouldn't programming language designers get rid of the switch statement?

Yes, I think Hoare would say that. Even C's switch statement is much more like the case than the earlier switch.

Special Operations

```
A.+B           instead of A :=A+B if A and B are matrices
L1.append(L2)  if L1 and L2 are lists
```

Hoare writes about how a programming language can help programmers deal with arithmetic expressions involving large operands. Using updating operations make programs with large operands more efficient since the space occupied by one of the operands can be used to hold the result. This is not much of a problem with small operands because they can be sufficiently held in a high speed register.

By using this example, Hoare shows how the presence of special notations in a language (such as the ones in the example) improves the efficiency of a program which deals with large operands.

Assignment to Reference Variables

`x:= y;` *always changes x*
but `x: = y+1;` *never changes x.*

The reason for this is because X is a location in memory, so rather then setting that location to an explicit space, as you do in the first case by specifying it with a variable, you are modifying the location it tracks in memory with the addition of that 1 to some other place entirely. This can be a serious problem because you don't know what's in that memory location. If you're lucky it's blank, if you're less lucky it has some important operating system information in it. (although now most operating systems don't give a person access to that memory).

Good try, but this analysis is incorrect.