Editorial

Vladimir Lifschitz

Reasoning about programs and reasoning about actions are two areas of research that do not seem to interact with each other. One is a part of the theory of programming; the other, a part of artificial intelligence. A computer scientist who works today in one of these two fields usually does not know much about what is happening in the other.

In the sixties, the intellectual atmosphere in computer science was different. John McCarthy described the operational approach to the semantics of programming languages in the following words:

> *The meaning of a program is defined by its effect on the state vector...* In the case of ALGOL we should have a function $\xi' = algol(\pi, \xi)$ which gives the value ξ' of the state vector after the ALGOL program π has stopped, given that it was started at its beginning and that the state vector was initially ξ ... A machine is described by a function $x' = machine(p, x)$ giving the effect of operating the machine program p on a machine vector x.[1]

Compare this with McCarthy's other invention—the "situation calculus" notation $s' = result(a, s)$ for the situation that results when the action a is carried out starting in the situation s. The situation calculus was first introduced in a 1963 memo; it seems that McCarthy came up with these two ideas at about the same time.

From this perspective, it is interesting to consider the old notion of a "strategy," presented by McCarthy and Hayes in terms of an analogy with ALGOL: "Actions can be combined into strategies... We shall combine actions as though they were ALGOL statements, that is, procedure calls."[2] A sequence of actions that solves the Monkey and the Bananas problem is taken as an example; it is written as

begin *move(Box,Under-Bananas); climb(Box); reach-for(Bananas)* **end**.

[1] John McCarthy. A basis for a mathematical theory of computation. In *Proc. IFIP Congress 1962*, pages 21–28. North Holland, 1963.

[2] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969.

The next example written in that paper in the ALGOL notation involves loops: It is a strategy "that consists of walking 17 blocks south, turning right, and then walking till you come to Chestnut Street."

The authors go on to discuss how methods developed in the theory of programming can be used for proving the termination and correctness of strategies like this. In the discussion of the frame problem, they refer to the paper on the operational semantics quoted above, and conjecture that the problem can be resolved by representing actions as assignments and by using "the general properties of assignment statements."

Few traces of this line of thought can be found in more recent research. The work by Zohar Manna and Richard Waldinger on the synthesis of imperative programs is a rare exception:

> Plans are closely analogous to imperative programs, in that actions may be regarded as computer instructions, tests as conditional branches, and the world as a huge data structure. This analogy suggests that techniques for the synthesis of imperative programs may carry over into the planning domain. Conversely, we may anticipate that insights we develop by looking at a relatively simple planning domain, such as the blocks world, would then carry over to program synthesis in a more complex domain...[3]

Most other authors do not seem to think that the theory of actions can learn anything from a closer look at the theory of programming, or the other way around.

How did these two areas become as distant as they are today? Were there any good reasons for this divorce?

What happened was apparently a gradual shift of emphasis that made it difficult for programming theorists and for AI researchers to see any relationship between their respective concerns.

First of all, the operational approach to the semantics of programs is not very popular in computer science today—unlike the situation calculus, which remains, along with one or two alternatives, a favorite formalism for representing properties of actions. Most authors "adopt a postulational method

[3]Zohar Manna and Richard Waldinger. How to clear a block: A theory of plans. *Journal of Automated Reasoning*, 3:343–377, 1987.

because its technical advantages, as compared with operational methods, are so overwhelming."[4] As Bertrand Russell observed in connection with the problem of defining real numbers, "[t]he method of "postulating" what we want has many advantages; they are the same as the advantages of theft over honest toil."[5] The possibility of proving the usual postulates for reals using Dedekind cuts is traditionally viewed as a fundamental mathematical fact, an important part of a student's education. Computer scientists somehow fail to be impressed by the possibility of proving Hoare's postulates.

On the other hand, central for the theory of programming is the study of the programming constructs that are used to form programs from atomic statements. In AI, on the contrary, almost all attention is given to the effects of individual actions; the idea of a strategy is all but forgotten.[6] The theorists of programming may be puzzled by the fact that their colleagues in artificial intelligence find an atomic action an interesting object to study. Is there anything nontrivial that one can say about the effects of an atomic action? Apparently, yes—this is where AI researchers found the notorious "frame" and "qualification" problems!

In terms of programming, the frame problem is the problem of describing which variables do not change their values when a procedure is called. There are several reasons why this is not usually perceived as a difficulty.

First, some of the representational mechanisms employed in the theory of programming do not seem particularly rich in comparison with the temporal formalisms used in artificial intelligence. Hoare triples and predicate transformers appear to be, roughly, on the same level of expressiveness as STRIPS operator descriptions, which were developed as a "scaled down" variation of the situation calculus.

Second, programming theorists do not share AI researchers' interest in "modular" axiomatizations that would represent each fact about a change that an action may cause by a separate axiom—as well as each fact about a circumstance that would make an action nonexecutable. This is what motivated the invention of formal nonmonotonic reasoning.

---

[4] Edsger Dijkstra and Carel Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[5] Bertrand Russel. *Introduction to Mathematical Philosophy.* Allen & Unwin, 1919.

[6] A welcome exception was mentioned by Raymond Reiter in his recent invited talk at KR'92: Hector Levesque, Fangzhen Lin and Raymond Reter, *Complex Events*, in preparation.

Third, in the theory of action we have to deal with "ramifications," or indirect effects. The direct effect of moving an object x to a location l is to make the assertion $at(x, l)$ true. From the fact that x cannot be located in two places at once we can conclude, furthermore, that the action has also an indirect effect: The assertion $at(x, l')$, where l' is the original location of x, becomes false. In some cases, ramifications make an action nondeterministic. As an example, consider the same action of moving x to l, and let us not require that l be clear when the action is executed. Since two objects cannot be in the same place, the object that currently occupies l will be nondeterministically forced to some other location.

Ramifications make the frame problem more difficult. Something similar to this phenomenon seems to occur in the theory of programming when pointers are introduced. But nondeterministic ramifications do not appear to have a counterpart in programming.

There are also pragmatic differences between planning and programming. Plans, unlike programs, are supposed to be constructed automatically, and, in comparison with programs, they usually have an extremely simple structure. Manna and Waldinger observed in an early draft of the paper quoted above:

> Planning systems are typically applied to very specific situations rather than to general classes of situations. We are told that block a is on block b, that a is clear, and that b is on the table. We are rarely asked to deal with a more general class of situations in which block a may or may not be clear. In program synthesis, on the other hand, we are almost always dealing with a general situation. No one would be impressed with a system that wrote a program to divide 7 by 3; we would want a program to divide m by n, for arbitrary integers, say. But in planning, solving specific instances of general problems is the norm. Perhaps for that reason, planning systems often fail to develop plans with tests. After all, if the situation is always fully specified, the plan has no need to test whether block a is clear or not; the planner has already been told the answer.

Moreover, "most plans are constructed to be executed once and, if the plan isn't exactly right, it can be modified on the spot to take the unexpected into account. Programs are usually made to be executed many times, and it

is not usually feasible to reprogram whenever a bug is discovered" (Richard Waldinger, personal communication).

To sum up, the lack of interaction between programming theory and artificial intelligence can be explained, to some extent. But it is very well possible that developing a common language and a better understanding of the similarities and differences between the two subjects can be beneficial for both communities—especially now, when the interests of AI researchers are beginning to move toward to the study of the two problems that have been discussed in the programming literature for years: nondeterminism and concurrency.