ON THE DEFINITION OP SPECIALIZATION HIERARCHIES FOR PROCEDURES

Alexander Borgida

Department of Computer Science
University of Toronto
Toronto, Ont. M5S 1A7, Canada

ABSTRACT

We highlight the growing body of systems in AI and outside where IS-A hierarchies of procedures co-occur with more traditional onea of objects, and we classify the various types of specialisations. We then give formal definitions which approximate their intended meanings and, finally, examine their utility using as a criterion the way in which they aid program verification.

I MOTIVATION

Inheritance (IS-A) hierarchies have been one of the trademarks of Semantic Network knowledge representations in AI. Although originally used for describing objects only, there is growing evidence both from within and outside AI that hierarchies of events/procedures are also useful [1,2,3,4,5,6]. It is therefore of some interest to take a brief look at possible formal foundations to the notion of specialization for procedures, and compare them according to some uniform criterion.

We beging by reviewing briefly the intended use of procedure specialization in some of the systems which currently support this idea. In SIMULA [1], SMALLTALK.76 [2] and PIE [3], the principal use of IS-A is for sharing code through inheritance; the goal is to save the programmer from the error-prone process of copying the material several times. In PSN [4], the IS-A hierarchy of programs provides the semantic basis for defining all other hierarchies (e.g. PERSON IS-A ANIMAL only if, among others, the To-add program of PERSON IS-A the To-add program of ANIMAL). In TAXIS [5,9,10], generalization/specialization is the principal abstraction tool of a methodology for Information System design and implementation. For example, as part of a university records system, the designer could introduce the many rules about what courses students can or must take by describing first the transaction for enroling any atudent in any course, and then specializing it to, among others, a transaction for enroling graduate atudenta into undergraduate courses, one for enroling part-time atudents, etc. In this caae apeclalization is used as a discipline for introducing the detaila of a system where consistency and completeneaa are at a premium. Finally, Rich [6] and others use hierarchiea to organize libraries of program plana in order to facilitate locating them In program synthesis/analysis tasks.

II HEE& OF Ol&zll HIERARCHIES

In this section, we will consider in more detail the varioua ways of defining the notion of specialization for activities. Probably the earlieat uae of inheritance hierarchies occurs in SIMULA-67 [1]. Here, one defines a class A by giving Its parameters PA and matching apeoiflcationa SA, declarationa DA, and a body of operatlona IA;inner;FA. One can then describe a subclass B by giving only the additional parts PB, SB, DB, IB and FB with the effect that claaa K with parameters PA and PB, ... and body IA:IB:<u>inngr</u>:FB:FA la created. We will call this <u>textual</u> inheritance and observe that its function is code sharing. This is a syntactic definition slnoe the claas SIMULATION can be considered to be IS-A the claaa LINKED-LIST, in contrast to the more standard AI view that all objects in a subclass must be viewable as objects of the superclass, at least in some way.

SMALLTALK-76 [2], and its descendant PIE [31, provide objects grouped into classes, where a claas is defined by the messages it recognizes and the methods (procedures) used to respond to each message. A subclass can add traits of its own or override those of its superclass by providing a new method for handling a message. SMALLTALK has a defau.lt version of inheritance: if a method for a message to a claas is not explicitly specified, one looks up the chain of superclasses for a method. Thla provided a great deal more freedom than in SIMULA, where one cannot modify the body of the auperclaaa, and in fact leads towards the opposite end of the spectrum where any two procedurea can be IS-A related. One can imagine Intermediate ayntactic versions of IS-A where, for example, one la allowed to specialize an IF-atatement only by replacing it with another IF-atatement.

Some researchera (e.g., [4,5,6,10]) adopt a stricter view of IS-A in which a subclass is a subset of its superclass (albeit one about which more la known); thua, if all EMPLOYEES must earn more than $10,000 then JANITORa must do so also, if JANITOR IS-A EMPLOYEE. Such a stricter interpretation for apeclalization is advocated for AI representations in [12], and In databases la motivated by the observation that when processing the elements of a claas in a loop, it la often uaeful to assume that they all satisfy the Integrity constraints stated for that claaa.

For reasons of symmetry, we are then lead to a different, more semantic definition of IS-A, one where the execution of a specialized procedure can be viewed in some sense as the execution of the more general one. On the basis of current experience (e.g., [4,5,6]), it seems that if B and A are procedures such that B IS-A A, then, ideally, (1) A should complete successfully in all situations where B does, (il) the final or intended effects of B should include at least those of A, and (iil) B should be allowed to have some additional effects. A typical example of this would be specializing the procedure which creates a new EMPLOYEE to the one which creates a new JANITOR.

For the remainder of this paper assume that all procedures are expressed in a "core" language, which allows simple variables, assignment and conditional statements, as well as a while-loop construct.* If we view the program as modifying machine states described by variables and their values, then we can define the semantics of a program A by, among others, RA, the set of initial/final state pairs connected by A, or PA, the set {(p,q)!p true in s, q true in s', (s,s') in RA) where p and q are formulas in some FOL over states.

We can start by defining (2,1): B IS-A A iff RB c RA ; this ensures (i) and (ii) above but unfortunately forces A and B to be identical whenever both are defined in the same state, thus contradicting (ill). The same holds for the other semantics of programs in [8], including requiring PA £ PB. To be more selective, one can define the difference A(s,s') between states as the set of changes from s to s', i.e., the set of pairs (x,e), where x is a variable with value e in state s' but with a different value in s. This leads to rule (2,2): B IS-A A iff for every (s,s^f) in RB there is (s,s") in RA such that A(s,s") c A(s,s') . This is the basis of the notion ofnet side effect, which is one of the underlying conditions of specialization in [7]. Alternatively, consider a procedure A to be "defined" by some particular pair of assertions (PreA, PostA) in PA and then let (U): B IS-A A iff PreB = PreA & p, PostB = PostA & q for some p, q; this is the surface notation for specialization in [6]. Another possibility is to let Free(f) stand for the set of free variables in a formula or program f, and let FA be {(p,q) in RA I Free(p), Free(q) c Free(A)}. One might then define (£*!): B IS-A A iff FA c FB , and thus obtain another characterization of IS-A which, like *(2.2)* and (2.3), captures conditions (i), (ii) and (iii) and yet constrains the additional effects of B so that they do not "oontradict" those of A.

Observe that all of the above definitions rely solely on the effect of the programs, not on their internal structure, and henoe the familiar notion of inheritance is missing. Both [5] and [6] attempt to oomblne into a hybrid definition the structural aspects of the procedure (parameters, statements, roles) with the semantic restrictions noted above in order to allow both inheritance of parts and a limited extension/modification of the more general procedure. In particular, in specializing a procedure one can usually specialize (a) the parameters, by imposing additional conditions on them, (b) the component statements, tests and primitive operations, and (c) one can extend the specialized procedure by adding new parameters and components. By using an FOL which allows procedures, etc. as domains to be quantified over, this can be stated rather elegantly ([6]) as B(x) = A(x) a r(x) where A is the characteristic predicate of procedure class A, which has already been specified axiomatically. One is lead to suppose that in "pure" PROLOG, where there are no side-effects, the condition for B IS-A A could simply be B«=>A; the reason for this is that PROLOG programs consist of clauses and a more specialized program would be "true" in fewer cases than the more general one, i.e., it would have additional or "stronger" clauses.

III "IS-A" tttPABCMSS *SL* PBQCEPVEBS AHE YMiriMTIPH

In addition to the various uses for the specialization hierarchy noted in section 1, one can observe that through inheritance common parts of procedures are factored out into higher classes. Now notice that these could, among others, be tested and verified independently, and this validation could then be "shared" by all the specializations of a procedure, i.e., presumably we need verify only the additions/modifications.* This could be a partial answer to the problem in program verification of how one breaks up in a motivated manner the proof of a large program into smaller, yet coherent parts. Consider, for example, verification using the standard Floyd-Hoare partial correctness assertions (pea's) p{A}q . An important application of this occurs in databases, where one would like to prove that all transactions maintain the integrity constraints invariant, so that the system would not have to check them after every update.•• It may therefore be of Interest to compare the various definitions of IS-A on independent grounds: how do they support such proof sharing.

The existence of a commonly used rule of inference: p{C)q, q{D}r k p{C;D}r makes textual inheritance, as in SIMULA, an attractive approach because it suffices to prove q{D}q in order to deduoe p{C;D}q from p{C)q. Unfortunately, this is not a necessary condition, as illustrated by C i

• Similar results hold in more general oases, although the definitions need to be more complex.

• The problem of re-validating programs which have been altered has also been considered in [11].

•* An extensive example of verifying a group of procedures organized in an IS-A hierarchy is presented in [10].

$x:=2$, $D \equiv y:=x$, $p \equiv q \equiv (y>0)$. More generally, if one has proven that p{E;F;G}q by showing p{E}r, r{F}s, and r{G}q, and then replaces E by E', as in default inheritence, then it is again sufficient but not necessary to show that r{E'}s.

Turning to the semantic definitions, (2.1) is ideal for our purposes because, at least for WHILE-programs, it gives $PA \subseteq PB$, which is equivalent to $p\{A\}q \vdash p\{B\}q$ for any p, q ([8]). Unfortunately, as we mentioned, definition (2.1) is too restrictive. In order to allow B to have additional effects, we might consider some definition of IS-A which required the pre- (post-) condition of the more specialized program to imply the pre- (post-) condition of the more general one, as in (2.3). Unfortunately, the following example suggests that no simple definition based on pca's will lead to the same ability to share proofs: let $A \equiv nil$, $B \equiv x:=2$, $p \equiv (x>0)$, $q \equiv (x<0)$; then p{A}p, q{A}q, true{B}p, and if r{B}q then r implies false, yet we want B IS-A A. The problem is fundamental and lies in the fact that conditions like p and q can make assertions about what a program has left unchanged, as well as its effects. It therefore seems that there will be no ideal definition for IS-A and we must content ourselves with only heuristic support for program verification.

Definition (2.4) provides the basis for two heuristic rules which together cover several situations occuring relatively frequently in practice:

- if p{A}q, and Free(p), Free(q) $\subseteq$ Free(A) then p{B}q;
- if Free(I) $\cap$ Free(B) = $\emptyset$ then I{B}I .

In addition, if B is obtained from A by interspersing extra statements then there is a simple heuristic for checking (2.4), namely Free(A) $\cap$ Free(B/A) = $\emptyset$, where B/A stands for all the statements of B which do not appear in A. The point is that the conditions of all these heuristics can be syntactically checked, and hence can be incorporated into a computer-aided development environment.

As a compromise, we therefore suggest (a) specialization of parameters by provision of additional constraints, and (b) modification and extension of components by textual or default inheritance, but subject to semantic constraints such as (2.4).

IV SUMMARY

We have considered a number of alternative ways of defining the specialization hierarchy of procedures. These definitions were motivated by actual uses of such hierarchies in current systems in software development and AI, and they range from purely syntactic ones, as in SIMULA, to purely semantic ones, with most recent versions of inheritance ([4,5,6]) being hybrid. We have sketched out formal definitions of IS-A for procedures and considered their utility for program validation, especially "inheriting" verification proofs.

ACKNOWLEDGEMENTS

My thanks to John Mylopoulos, Harry Wong and Hector Levesque for numerous discussions connected with this work. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1. Dahl, O.J., and Nygaard, K., "SIMULA -an ALGOL based simulation language", CACM 9, September 1966.

2. Ingalls, D.H. "The Smalltalk-76 programming system: design and implementation", Conf. Record of 5th Annual ACM Symp. on Programming Languages, January 1978.

3. Goldstein, I.P. and Bobrow, D.G. "Extending object oriented programming in Smalltalk", Proceedings of 1980 LISP Conference, Palo Alto, August 1980.

4. Levesque, H. and Mylopoulos, J., "A procedural semantics for Semantic Networks", in Associative Networks: Representation and use of knowledge by computers, N.Findler ed., Academic Press, 1979.

5. Mylopoulos, J., Bernstein, P.A. and Wong, H.K.T. "A language facility for designing interactive database intensive applications", ACM TODS 5, June 1980.

6. Rich, C. "Inspection methods in programming", Ph.D. Thesis, MIT, June 1980.

7. Mylopoulos, J. and Wong, H.K.T. "Some features of the TAXIS data model", Proc. of the 6th Annual Conf. on Very Large Data Bases, Montreal, Sept. 1980.

8. Greif, I. and Meyer, A.R. "Specifying the semantics of WHILE-programs", LCS Technical Memo - 130, MIT Lab for Computer Science, April 1979.

9. Borgida, A., Mylopoulos, J. and Wong, H.K.T. "Taxonomic Software Specification", submitted for publication.

10. Wong, H.K.T., "Design and verification of Interactive Information Systems", Ph.D. Thesis, Dept. of Computer Science, University of Toronto, 1981.

11. Shrobe, H., "Dependency-directed reasoning for complex program understanding", MIT AI Lab TR-503, MIT.

12. Israel, D. and Brachman, R. "Some remarks on the semantics of representation languages", IJCAI 7, Vancouver, B.C., August 1981.