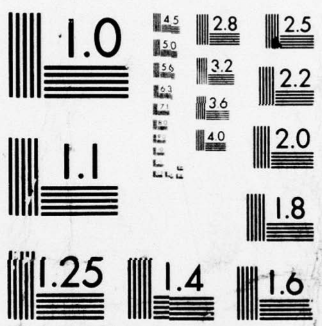
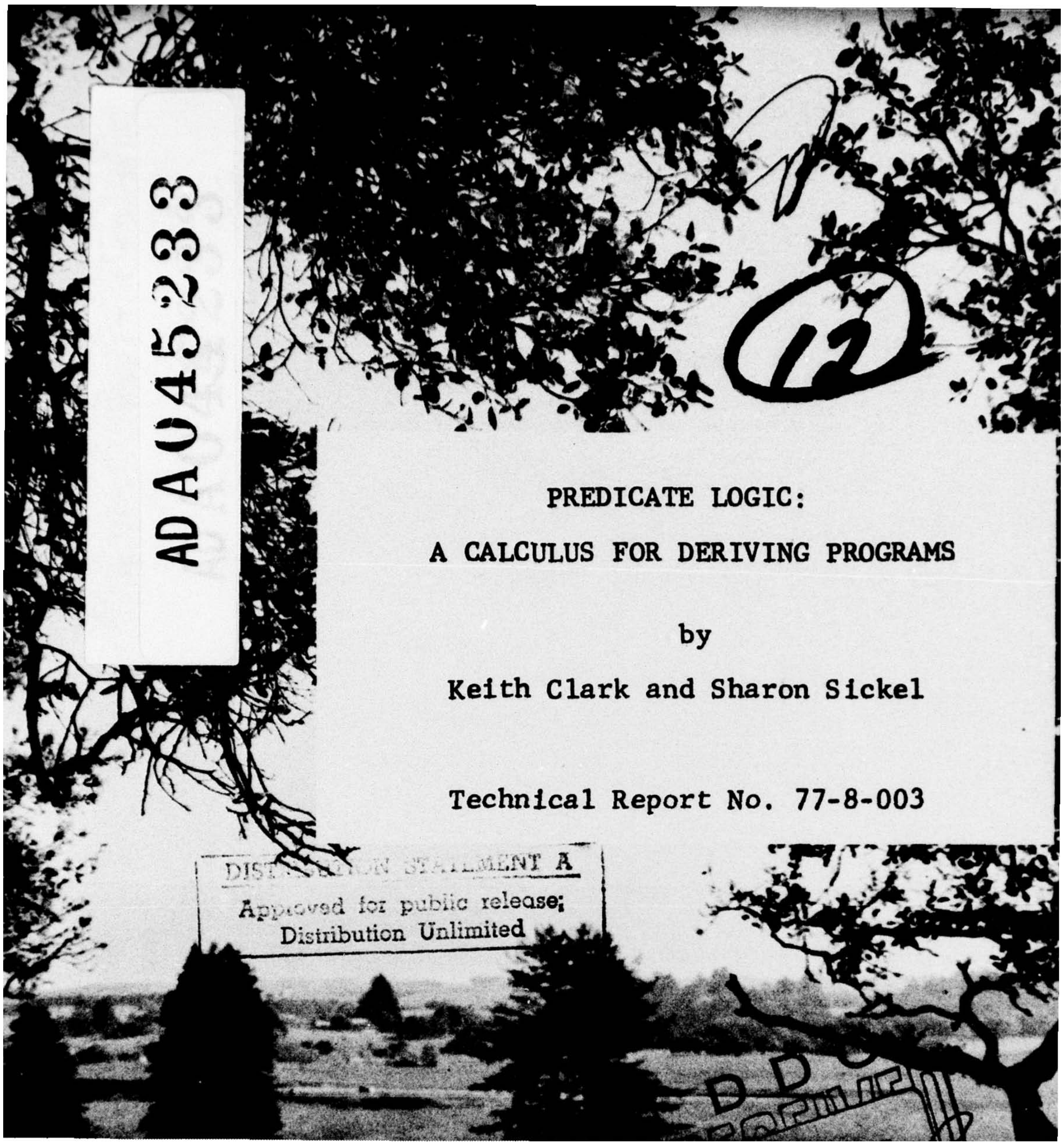
AD-A045 233 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
PREDICATE LOGIC: A CALCULUS FOR DERIVING PROGRAMS.(U)
MAY 77 K CLARK, S SICKEL N00014-76-C-0681
UNCLASSIFIED TR-77-8-003 NL

1 OF 1
AD
A045 233

END
DATE
FILMED
11-77
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0



12

PREDICATE LOGIC:
A CALCULUS FOR DERIVING PROGRAMS

by

Keith Clark and Sharon Sickel

Technical Report No. 77-8-003

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER 9
4. TITLE (and Subtitle) 6 PREDICATE LOGIC: A CALCULUS FOR DERIVING PROGRAMS.		5. TYPE OF REPORT & PERIOD COVERED Technical rept.,
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Keith Clark and Sharon Sickel		8. CONTRACT OR GRANT NUMBER(s) 15 N00014-76-C-0681
9. PERFORMING ORGANIZATION NAME AND ADDRESS Information Sciences University of California Santa Cruz, California 95064	14 TR-77-8-003	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, Virginia 22217	11 31	12. REPORT DATE May 1977
		13. NUMBER OF PAGES 12 5p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research University of California 553 Evans Hall Berkeley, California 94720		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Predicate calculus, programming methodology, program synthesis

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We show how predicate logic ~~can be used~~ is shown to derive programs from axiomatic specifications. ~~We also show how~~ its proof theory can be used to analyze, and re-characterize, the computations of a program.

410350

PREDICATE LOGIC: A CALCULUS FOR DERIVING PROGRAMS

Keith Clark
Computing & Control
Imperial College
London, England

Sharon Sickel
Information Sciences
University of California
Santa Cruz, California

May 1977

Published in the Proceedings of the International Joint Conference on Artificial Intelligence, Boston, August 1977.

Supported in part by the Office of Naval Research under Contract N00014-76-C-0681.

Abstract

We show how predicate logic can be used to derive programs from axiomatic specifications. We also show how its proof theory can be used to analyse, and re-characterize, the computations of a program.

1. Programs as computationally useful theorems

We start with a set of axioms that give an intuitively correct characterization of some input-output relation we wish to compute. Under the procedural interpretation of logic [5,6] these axioms can be used to 'compute' the relation. So we 'symbolically execute' the axioms, qua program, for various forms of input. From each symbolic execution we distil a theorem that can double for the axioms in the business of computing the relation. Our set of derived theorems is a logic program whose procedural use *is* computational.

Example

The following axioms are a specification for the input-output relation, mem-test, of a program to test if an element u is a member of a list z. The output is to be T if u∈z, F if not. All free variables (lower case) are implicitly universally quantified.

Specification

$$\sim u\in NIL \quad (\text{or } u\in NIL \leftrightarrow \textit{false})$$

$$u\in v.x \leftrightarrow u=v \vee u\in x$$

$$\text{mem-test}(u,z,t) \leftrightarrow t=T \,\&\, u\in z \vee t=F \,\&\, \sim u\in z$$

In several respects this axiomatisation is incomplete. We should really axiomatise the equality relation for lists, and list elements, and express the finiteness condition for lists by an induction schema [3]. However the absence of explicit equality axioms is an implicit assumption that things are equal only if they are identically named, which is what we intend, and we shall not need the induction schema for the program synthesis.

Synthesis

We 'evaluate' the definition of mem-test for the cases z=NIL and z=v.x .

z=NIL

$$\text{mem-test}(u,NIL,t) \leftrightarrow t=T \,\&\, u\in NIL \vee t=F \,\&\, \sim u\in NIL$$

Using the axiom u∈NIL ↔ *false* the 'call' u∈NIL evaluates to *false* , giving

$$\text{mem-test}(u,NIL,t) \leftrightarrow t=T \,\&\, \textit{false} \vee t=F \,\&\, \sim\textit{false}$$

The definiens now reduces to t=F using only logical evaluation rules. In effect we have proved

$$\text{mem-test}(u,NIL,F) \qquad (1)$$

z=v.x

$$\text{mem-test}(u,v.x,t) \leftrightarrow t=T \,\&\, u\in v.x \vee t=F \,\&\, \sim u\in v.x$$

This time we evaluate the 'call' u∈v.x by substituting the equivalent expression $u=v \vee u\in x$.

$$\text{mem-test}(u,v.x,t) \leftrightarrow t=T \,\&\, (u=v \vee u\in x) \vee t=F \,\&\, \sim(u=v \vee u\in x)$$

We now bring the components of the substituted expression to the surface by distributing connectives. We do this in order to throw together formulae such as P& P that can be logically evaluated. But, more importantly, we 'multiply out' in the hope of eventually 'factoring out' an expression that is just another instance of the mem-test definiens. If we can do this we have found a recursive use of the mem-test definition from which we can infer a recursive theorem. Distributing gives

$$\text{mem-test}(u,v.x,t) \leftrightarrow t=T \,\&\, u=v \vee \boxed{t=T \,\&\, u\in x \vee t=F \,\&\, u\neq v \,\&\, \sim u\in x}$$

The boxed disjunction very nearly matches the mem-test definiens. The 'difference' is the extra condition u≠v that appears in its right disjunct. We could factor this out if it also appeared in the left disjunct. So we introduce it!

$$\text{mem-test}(u,v.x,t) \leftarrow t=T \,\&\, u=v \vee t=T \,\&\, u\neq v \,\&\, u\in x \vee t=F \,\&\, u\neq v \,\&\, \sim u\in x$$

But note that the "↔" has been down-graded to "←". Introducing u≠v destroys the equivalence. However, since t=T & u≠v & u∈x implies t=T & u∈x, we still have the if-half of the iff. We now factor out u≠v.

$$\text{mem-test}(u,v.x,t) \leftarrow t=T \,\&\, u=v \vee u\neq v \,\&\, (t=T \,\&\, u\in x \vee t=F \,\&\, \sim u\in x)$$

Substituting mem-test(u,x,t) for its definiens gives

$$\text{mem-test}(u,v.x,t) \leftarrow t=T \,\&\, u=v \vee u\neq v \,\&\, \text{mem-test}(u,x,t)$$

which we expand as the pair of theorems:

$$\text{mem-test}(u,u.x,T) \qquad (2)$$
$$\text{mem-test}(u,v.x,t) \leftarrow u\neq v \,\&\, \text{mem-test}(u,x,t) \qquad (3)$$

Theorems (1),(2), and (3) are the statements of our derived program. With minor syntactic changes, they are in fact a PROLOG program [10], PROLOG being essentially a 'top-down' resolution theorem prover. A request to refute

$$\sim\text{mem-test}(2,(4.(3.(5.NIL))),t)$$

is a call of the program. It will generate the recursive computation one expects. This computation is a constructive proof that binds t to F.

Correctness

A logic program that comprises a set of theorems about the relation it is supposed to compute is, in the computational sense, (partially) correct. (Computing an instance of the relation is then *proving* it is a correct instance.) Thus, a logic program is *verified* by *checking* that each of its statements are theorems; it is *synthesised* (and verified) by *finding* each of its statements as theorems. This approach to verification and synthesis is elaborated in [2].

2. Proof theory analysis of computation

The computations of logic programs are resolution proofs. We can characterize such proofs as paths through an interconnectivity graph [8], the unifications that appear on each path being the essential steps of the proof computation. This conceptualization of what constitutes a proof gives us a tool for analysing, and reformulating, a logic program.

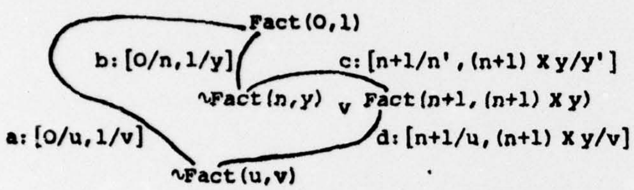
Example

The logic program

Fact(O,1)

Fact(n+1,(n+1) X y) ← Fact(n,y)

is used to compute the factorial function by asking *for a refutation of a conjecture of the form* ~Fact(u,v) where u is some numeral input. Below is an interconnectivity graph for the general theorem proving task in which the conditional statement has been expressed in clausal form. Unifiable complementary literals are connected with an edge labelled by the unifying substitution.



A proof of Fact(u,v) is given by any path through the graph that connects ~Fact(u,v) with *Fact(O,1)*. In this case the set of all possible paths can be succinctly described by the regular expression a | bc*d . In effect, this is an iterative characterization of the set of compositions of unifications *that constitute a proof. Taking into account* the intended use of the logic program, i.e. that u is to be input and v output, it is a compact notation for the iterative program:

1) a: if u=O then v←1
2) b: initialise u'←O; v'←1
   c*: repeat (zero or more times)
        u'←u'+1;
        v'←u' X v'
   d: terminate above loop when u'=u
        v←v'

General method

The above example was simple enough for us to read off directly from the graph a regular expression. For more complex examples we may first need to characterize the set of proofs by a context free attribute grammar. This we can always do [9]. The productions of such a grammar reflect the ground structure of the problem, taking into account unifiable pairs of literals, but ignoring the necessary substitutions. The attributes carry the substitution information. Temporarily ignoring the attributes we try to re-express the language generated using regular expressions. To the extent that we are successful, we then re-introduce the substitution constraints as refinements of the regular expressions. Thus, in the Fact example, the regular expression bc*d would be refined by the constraint that c is applied the number of times to satisfy the substitution. The refined expression is therefore $bc^{(u-1)}d$.

Domain and range

The attributes are consistency checks on the variables; a production can be applied only if its associated substitution constraint is consistent with the substitution constraint of all previous steps in the derivation. The refined regular expression therefore gives us restrictions on each of the variables that must be satisfied in any proof. The restrictions on the input variables determine the domain, those on the output variables the range. For Fact, this analysis gives us O+(+1)* as the domain, i.e. the natural numbers, and, for n in the natural numbers, n X ((n-1)..X(2 X 1)..) as the range.

Termination

If we are able to describe the set of computations as a regular expression we can use the attributes to replace the *'s with specific integer functions of the arguments. If we can do this for every *, that is for every implicit iteration, we have proved that every computation terminates.

3. Final remarks

So far we have only investigated the hand synthesis of logic programs. However it is intended to implement an interactive system which becomes more autonomous as the synthesis methodology is refined and understood. The idea of synthesing a recursive program from the recursive use of a specification first appeared, independently, in [1] and [7]. Indeed the reader may have noticed the similarity between our approach and that of Darlington and Burstall[1,4]. Like them we use the same formalism for both specification and program (they use enriched recursion equations), and like them we symbolically execute the specification. We have derived much from their work.

The proof theory analysis of computation is also in its beginning stages. It is in fact an application of more general work, currently in progress, on the analysis of resolution proofs. We believe it provides a useful conceptualization, and will provide a useful tool.

References

[1] R.M.Burstall & J.Darlington, *Some transformations for developing recursive programs*, Proc. Int. Conf. on Reliable Software, Los Angeles (1975)

[2] K.L.Clark, *Synthesis and verification of logic programs*, Research report, CCD, Imperial College (1977)

[3] K.L.Clark & S-Å Tarnlünd, *A first order theory of data and programs*, Proc. IFIP Congress (1977)

[4] J.Darlington, *Application of program transformations to program synthesis*, Colloques IRIA on Proving and Improving Programs, (1975)

[5] P.J.Hayes, *Computation and deduction*, Proc. MFCS Conf., Czech Academy of Sciences (1973)

[6] R.Kowalski, *Predicate logic as programming language*, Proc. IFIP Congress (1974)

[7] Z.Manna & R.Waldinger, *Knowledge and reasoning in program synthesis*, Art. Int. Journal, 6(2), (1975)

[8] S.Sickel, *A search technique for interconnectivity graphs*, IEEE Trans. on Computers, Aug. (1976)

[9] S.Sickel, *A linguistic approach to automatic theorem proving*, Proc. CSCSI/SCEIO Summer Conf (1976)

[10] D.Warren, L.Pereira & F.Pereira, *PROLOG-the language and its implementation compared with LISP*, SIGPLAN/SIGART Prog. Lang. Conf., Rochester (1977)

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0681

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Code 102IP
Arlington, VA 22217
6 copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area Office
715 Broadway - 5th Floor
New York, NY 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, DC 20375
6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (CodeRD-
Washington, D. C. 20380
1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Cent
Computation and Mathematics Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-911G)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy