ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE - C.N.R.

A LOGICALLY BASED CLASS OF PROGRAM SCHEMATA AND ITS RELATION TO COMPLEX CONTROL STRUCTURES.

P. Degano , G. Levi

L75-29

Presentato al Congresso Annuale della Associazione Italiana per il Calcolo Automatico, Genova, 29, 30, 31 Ottobre 1975

Stampato in proprio

(C_1) is maximal iff it is not contained in any compatibility class.

-let λ_{C_i} denote the composition of all the substitutions λ_j which P_j is applicable to G with. Let G^i_{k-1} be the string of literals labelling the leaves of the graph s^i_{k-1} and $(C_j)=(P_h: M_h=N_h)$ a maximal compatibility class w.r.t. G. Furthermore let H be the substring of G^i_{k-1} which P_h is applicable to;

-the application of P_h to s^i_{k-1} can be defined in the following way:

i) every node belonging to H is connected to a new node Q labelled by (P_h,λ_{C_j});

ii) the node Q is connected to m new nodes labelled by the literals of P_h;

-applying the maximal compatibility class (C_j) to s^i_{k-1} means:

i) joining the graphs obtained applying all the applicable procedure declarations $P_h\varepsilon(C_j)$;

ii) applying the substitution λ_{C_j} to every leaf;

-a graph is terminal iff all its leaves are labelled by T;

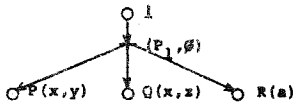
-a computation is k-th step terminal iff there exists a terminal graph $s^i_{k-1}\varepsilon(s_k)$.

Let us give an example of the execution of a schema, whose interpretation may be the following: let R denote a "read" process, P and Q two similar processes. If the control flow passes through P in presence of R, the output variables y,z have to assume the value g(x), if it passes through Q in presence of R, the output variables assume the value f(x). The schema is defined by the following equations:

P_1: $i=P(x,y),Q(x,z),R(a)$
P_2: $P(x,y),R(x)=S(x,y)$
P_3: $S(x,g(x))=T$
P_4: $P(x,f(x))=T$
P_5: $Q(x,z),R(x)=W(x,z)$
P_6: $W(x,f(x))=T$
P_7: $Q(x,g(x))=T$

(s_0) : O i

(s_1) : O i



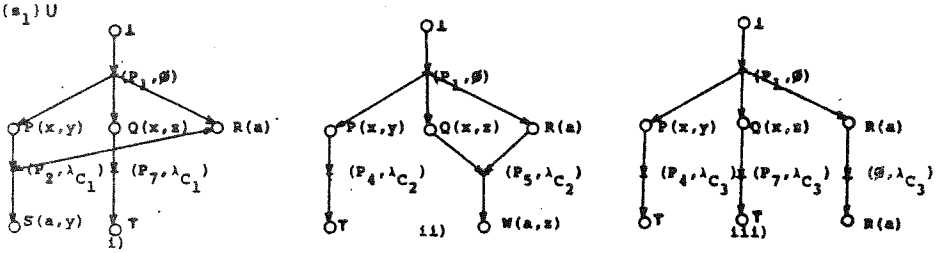
P_2 applicable with $\lambda_2=\{<x,a>\}$
P_4 " " $\lambda_4=\{<y,f(x)>\}$
P_5 " " $\lambda_5=\{<x,a>\}$
P_7 " " $\lambda_7=\{<z,g(x)>\}$
P_2 part-applicable " $\lambda'_2=\emptyset$
P_5 " " $\lambda'_5=\emptyset$

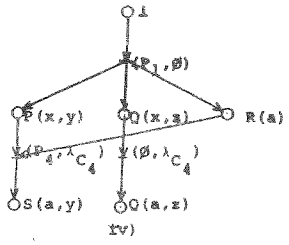
Maximal compatibility classes:

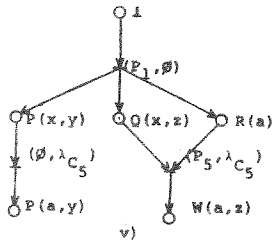
$C_1=\{P_2,P_7\}$ with $\lambda_{C_1}=\{<x,a>,<z,g(a)>\}$
$C_2=\{P_5,P_4\}$ " $\lambda_{C_2}=\{<x,a>,<y,f(a)>\}$
$C_3=\{P_4,P_7\}$ " $\lambda_{C_3}=\{<y,f(x)>,<z,g(x)>\}$
$C_4=\{P_2,P_5\}$ " $\lambda_{C_4}=\{<x,a>\}$
$C_5=\{P_2,P_5\}$ " $\lambda_{C_5}=\{<x,a>\}$

Note that C_4 and C_5 are two different classes because in C_4 P_2 is applicable and P_5 part-applicable while in C_5 P_2 is part applicable and P_5 applicable. The reason of having both this classes stands in the fact that we want to mantain the possibility of applying every procedure declaration.

(s_2) : $(s_1)\cup$



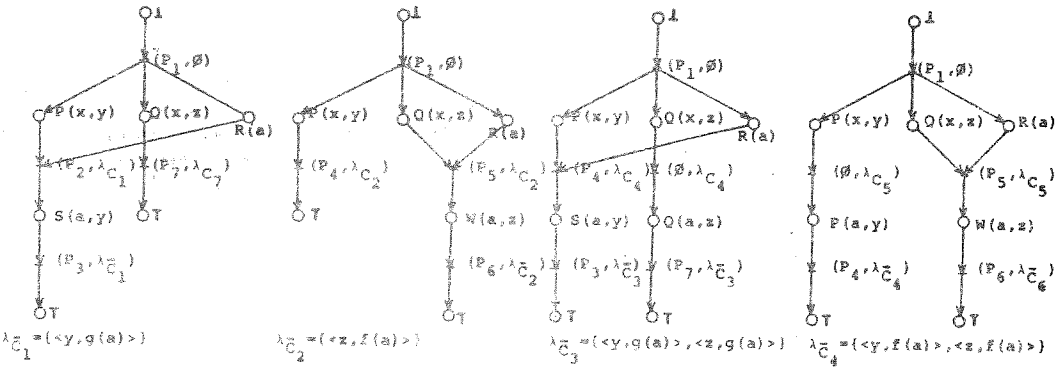




Note that no procedure name appears in the labels of the iii) graph, so it is a "dead end graph".

The next computation step leads to:

$(s_3) : (s_2) \cup$



Note that the graphs derived in step 3 are terminal.

It is worth noting that it is possible to give a simpler definition for the interpreter, applying only one equation at a time. However such a strategy will lead to an explosion of the number of the graphs built at each step of the computation. Besides it is easy to proof that our interpreter uses the highest degree of parallelism during the execution.

Note that it is possible to interpret our class of program schemata also in terms of problem reduction as expressed in [5]. Here we want to stress only that the set of the computations of our schemata contains properly the set of the computations of Kowalski's schemata, since the elements of the latter are AND-OR graphs, while ours are a generalization of the AND-OR graphs.

It is very easy to show [5] that the above defined interpreter is a "resolution-based" theorem prover for first order logic augmented with non standard conjunction. Furthermore, because of its logic basis, the formalism can be extended to express theorems about program schemata, whose proof can be attempted, provided that the interpreter has the ability to reason by induction.

4. Algebraic semantics

Let us characterize the set of the computations of a schema as a lattice (S) in order to give the semantics to the schema itself by means of the minimum fixed point of the associated continuous functional. Let $\sqsubseteq$ denote the patial ordering, $\top$ and $\bot$ the top and the bottom respectively, $\sqcup$ and $\sqcap$ the join and meet operators (on sets).

Let (s_1) and (s_2) be two sets of graphs of a computation, $(s_1) \sqsubseteq (s_2)$ iff (s_2) contains (s_1). Remark that $\bot$ is (s_0), that $(s_{k-1}) \sqsubseteq (s_k)$, $\forall$ k, and that the lattice is complete. The functional (ES) associated to the schema is the set of the equations itself, which is obviously a monotonic mapping onto the lattice, since applying it to the set of graphs (s_{k-1})(following the definition given in 3.) we obtain (s_k) and $(s_{k-1}) \sqsubseteq (s_k)$. Furthermore ES is continuous, in fact $\forall$ h,k

$$\sqcup ES(s_{k-h}),\ ES(s_{k-h+1}),\ \ldots,\ ES(s_k) = ES(\ (s_{k-h},\ s_{k-h+1},\ \ldots,\ s_k\ ))$$

So we can use the Knaster—Tarski theorem and find the unique set of graphs (s_ω) in the computation of our schema which characterizes it and such that: $(s_\omega) = \bigsqcup_{n=0}^{\infty} ES^n(\bot)$

Congresso A.I.C.A. 1973, Genova

P. DEGANO*, G. LEVI*,(0)

A LOGICALLY BASED CLASS OF PROGRAM SCHEMATA AND ITS RELATION TO COMPLEX CONTROL STRUCTURES

Abstract: The paper introduces a class of program schemata based on recursion and non-determinism whose expressive power overcomes the capability of the recursive non-deterministic schemata defined so far. The class contains complex schemata for which no equivalent standard recursive non-deterministic schema can be defined. As an example, we consider schemata which express in a purely syntactic way control constructs related to parallelism, such as cooperation, synchronization, co-routining etc.

1. Schema formalism

Our formalism, based on first order predicate calculus, uses a relational representation of non-primitive functions (i.e. f(x,y) is rapresented by the relation F(x,y,z), such that f(x,y)=z). Such a representation, in which a strict distinction between input and output variables is avoided, leads to a general and flexible formalism which is suitable for "reasoning about program schemata".

A program schema is a set of equations of the following form:

(1) $A_1, A_2, \ldots, A_m = B_1, B_2, \ldots, B_n$

where the A_i's, i=1,m, and the B_j's, j=1,n, are first order logic literals, whose constant and function symbols are uninterpreted. Let F be the set of formal variables (i.e. variable symbols which appear in the left-hand parts of the equations) and A be the set of actual variables (i.e. variable symbols which appear in the right-hand parts only).

It is worth noting that our formalism is an extension of Kowalski's Predicate Logic [3], since the latter does not allow equations to have a left-hand part consisting of more that one literal.

2. Logical and procedural interpretations

From a logical point of view, our equation (1) can be read as stating:

---

*) Istituto di Scienze dell'Informazione dell'Università degli Studi di Pisa

(0) Istituto di Elaborazione dell'Informazione del C.N.R.

(2) A_1 AND A_2 AND ... AND A_m if B_1 and B_2 and ... and B_n

where the AND operator has a non standard interpretation, i.e. the AND conjunct does not imply its conjuncts. Hence the hyper-clause* (2) allows only to assign a denotation to the single compound relation (A_1 AND A_2 AND ... AND A_m), while the individual conjunct denotations are undefined. On the other hand, the compuond relation cannot be rapresented by a single compound predicate, since the A_i's may occur in other clauses.

It is possible also to give a procedural interpretation to equations of the form of equation (1). If its left-hand set consists of the single distinct literal I,which stands for start, it is a main program schema; otherwise it is a procedure declaration schema. The left-hand literals are the procedure names, while the right-hand literals are the procedure body. An assertion is a procedure declaration schema, whose body consists of the single distinct literal T, which stands for stop. A right-hand side literal B_j can be seen as a call of a procedure whose name A_i "matches" B_j. A body consists of several procedures calls which have to be "executed" concurrently. Multi-name procedures (corresponding to equations having more than one left-hand literal) are activated by concurrent calls to all its names. The only basic control primitive is the parallel call of procedures by "matching". Standard constructs, like function composition, conditionals and "parallel if-then-else" can be easily expressed within the formalism. For example, the assignament y=f(g(a),b) is translated into two parallel procedure calls: G(a,x),F(x,b,y).

3. Operational semantics

In this section we give the definition of an operational semantics for our class of schemata, by defining a syntactic interpreter which can be regarded both as a schema evaluation mechanism, and as a first order logic theorem prover.

Let us first give the following definitions:

- let N_i be the string of the procedure name and B_i be the procedure body of the i-th equation P_i: $N_i=B_i$;
- let G be a string of literals** and G' a substring of G, P_i: $N_i=B_i$, then N_i is unifiable with G' iff N_i and G' can be unified by the first order logic unification algorithm and for every substitution <variable, term> one of the following conditions holds:
  - i) <variable>$\notin$F and no symbol of F occurs in <term>;
  - ii) <variable>$\in$A;
- P_i is applicable to G with substitution λ iff $(N_i)_\lambda=(G')_\lambda$;
- P_i is part-applicable to G iff at least one literal in P_i is unifiable with a literal in G.

Let us define now the computation of a schema:

- a computation is a sequence of sets of directed graphs $\{s_0\}$, $\{s_1\}$, ..., $\{s_m\}$ where
  - i) $\{s_0\}$is the set whose only element is the graph with a single node labelled by I;
  - ii) let s^i_{k-1} be the i-th graph of the set $\{s_{k-1}\}$ and let G^i_{k-1} be the string of the literals labelling the leaves of s^i_{k-1}. $\{s_k\}$ is the result of the union of $\{s_{k-1}\}$ and the graphs obtained applying to s^i_{k-1}, $\forall$ i, all the maximal compatibility classes, one at a time.

In order to formalize the concept of application, we need some more definitions:

- two procedure declarations P_i and P_j are non-compatible w.r.t. a string G iff one of the following conditions holds:
  - i) P_i and P_j are part applicable to the same literal(-s) in G;
  - ii) P_i and P_j are applicable to G with substitutions λ_i and λ_j respectively, such that $\lambda_i \circ \lambda_j=\emptyset$;
  - iii) $\exists P_k$ which is non-compatible w.r.t. G with P_i and P_j.
- let $\{P_G\}$ be the set of all the procedures part-applicable** to G. A subset $\{C_i\}$ of $\{P_G\}$is a compatibility class (w.r.t. G) iff no pair of non-compatible elements (w.r.t. G) belongs to it.

---

* Note that hyper-clauses are different fron non-Horn clauses, because their left-hand sides are not disjuntions.

** In G we do not take care of the literals T.

*** Note that an applicable procedure declaration is also part-applicable.

5. Program schemata and complex control structures

We will first consider how non-determinism is handled in the framework of our schemata. The most popular strategy used is backtracking, which unfortunately is not admissibile. Some A.I. languages (QA4, QLISP, NDLISP) use the state-saving technique of contexts; i.e. every computation is carried on in a context, while at a choice point a new context is created copying the actual state of the computation (memory and control). Let us show how it is possible to model such a strategy using our formalism.

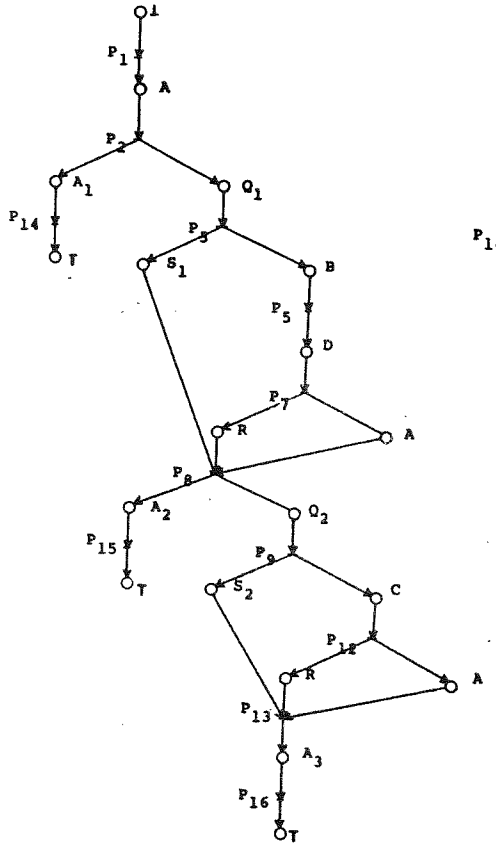
We define:

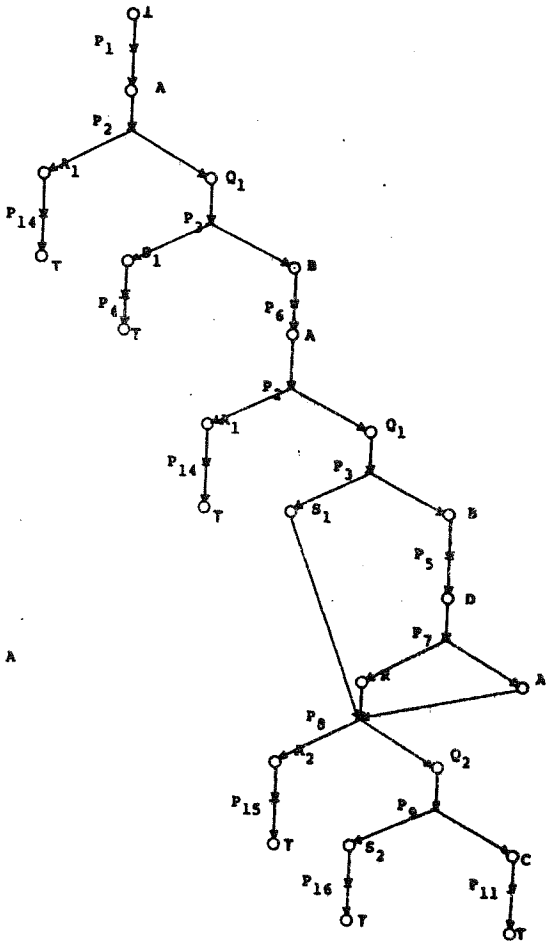
-a context is a sequence of graphs (context states) $s_0, s_1, \ldots, s_k$ such that s_{i+1} is obtained from s_i, $\forall i$, applying one maximal compatibility class;
-the memory state of a context is the substitution λ associated to it;
-a context is terminal iff it has a state whose leaves are labelled by T ;
-a computation is terminal iff it has a terminal context;
-a computation is non-deterministic iff it has more than one terminal context;
-a computation is deterministic iff it has at most one terminal context;
-a computation is determinate iff all its terminal contexts have the same memory state λ.

Another peculiar aspect of our schemata is their ability to formalise those complex control structures which are present in some new very high level programming languages and which allow non standard constructs such as co-routining and retention [7]. What actually makes it possible is strictly related to an explicit representation of continuation points which essentially relies on the compound names of the procedure declarations.

As an example of a schema which uses such complex control structures and which has no equivalent among Kowalski's schemata, we give the following variable free schema:
let A be a procedure whose execution can be resumed after a "return" has been performed; A can first return after computing A_1 and B, if resumed, A returns after computing A_2 and C; if A is resumed after such a return A_3 is computed. Resuming A is different from calling A, since the last command requires A to be executed starting from its first entry point. A is resumed by calling R,A, while the multiname procedure declarations P_8 and P_{13} are explicit representation of continuation points, which depend from the "sleeping state" (S_1 or S_2). The following graphs are two possible terminal contexts of the schema:

P_1: I=A
P_2: A=A_1,Q_1
P_3: Q_1=S_1,B
P_4: S_1=T
P_5: B=D
P_6: D=R
P_7: D=R,A
P_8: S_1R,A=A_2,Q_2
P_9: Q_2=S_2,C
P_{10}: S_2=T
P_{11}: C=T
P_{12}: C=R,A
P_{13}: S_2R,A=A_3
P_{14}: A_1=T
P_{15}: A_2=T
P_{16}: A_3=T





References

[1] Degano,P. , Una classe di schemi paralleli non-deterministici, Nota Int. I.E.I., B74-39.

[2] van Emden,M.H. & R. Kowalski, The semantics of predicate logic as a programming language, M.I.P.-R-103 School of A.I., University of Edinburgh.

[3] Kowalski,R., Predicate Logic as Programming Language, Proc. Information Processing 74.

[4] Levi,G. & F. Sirovich, Generalized AND-OR graphs and their relation to formal grammars, Nota Int. I.E.I., B73-15.

[5] Levi,G. & F. Sirovich, A problem reduction model for non independent subproblems, Proc. IJCAI4.

[6] Montangero,C., G. Pacini & F. Turini, Two level control structure for non-deterministic programming, Nota Int. I.E.I., B74-31.

[7] Montangero,C., G. Pacini & F. Turini, MAGMA-LISP: a machine language for A.I., Proc IJCAI4.

[8] Rulifson,J.F., Waldinger,R.J. & J.A.Derksen, QA4: procedural calculus for intuitive reasoning, A.I. Centre, Tech. Note 73, SRI.