USING TEMPORAL ABSTRACTION TO UNDERSTAND RECURSIVE PROGRAMS INVOLVING SIDE EFFECTS

Joachim Laubsch

Universitaet Stuttgart
Stuttgart
W. GERMANY

Marc Eisenstadt

The Open University
Milton Keynes
ENGLAND

ABSTRACT

This paper develops the notion of temporal abstraction, used originally for the automatic understanding of looping constructs, to account for a class of recursive programs involving side effects upon a relational data base. The programs may involve compositions of several side effects, and these side effects can occur either during descent or upon ascent from recursive calls.

I INTRODUCTION

The concept of 'temporal abstraction' was developed by Waters [5], and Rich & Shrobe [4] to describe the variables enumerated in loops as a set of objects which could be manipulated as a whole. We apply this principle to recursive procedures operating on threaded data structures and use it to understand compositions of several side effects. Our analysis is part of a larger project designed to help novice programmers understand their buggy programs ([2], [3]). The novices are students learning a LOGO-like language called SOLO [1], in which a procedure can only side effect a global data base which is a labelled, directed graph. Here is a sample problem, of the kind posed to our students: "Define a procedure INFECT which would convert a data base such as the one shown in Fig. 1 to the one shown in Fig. 2 if invoked as INFECT ANDY."

```
                  |---------> INOCULATED <-----|
                  |is                          |is
    kisses        |     kisses         kisses  |
ANDY-------->BARBARA-------->COLIN-------->DIANA
```

Fig. 1

```
                  |---------> INOCULATED <-----|
                  |is                          |is
    kisses        |     kisses         kisses  |
ANDY-------->BARBARA-------->COLIN-------->DIANA
|                             |
|gets                         |gets
|-----------> FLU <-----------|
```

Fig. 2

The primitives for storing, deleting, and retrieving relational triples are called NOTE, FORGET, and CHECK. CHECK provides for conditional branching depending upon its success, and also allows simple pattern matching, as the example below illustrates:

SOLUTION-1:

```
TO INFECT /X/
1 EXAMINE /X/
2 CHECK /X/ KISSES ?Y
 2A If present: INFECT *Y; EXIT
 2B If  absent: EXIT

TO EXAMINE /X/
1 CHECK /X/ IS INOCULATED
 1A If present: EXIT
 1B If  absent: NOTE /X/ GETS FLU; EXIT
```

INFECT recursively generates successive nodes along the thread of 'kisses' relations (steps 2 and 2A), and invokes EXAMINE at each node. EXAMINE conditionally side effects each node, i.e. asserting that /X/ GETS FLU only when the triple /X/ IS INOCULATED is absent.

This problem is typical of a large class of tasks given to beginning SOLO users. Students may adopt a variety of methods for tackling the stated problem, combining side effects to achieve the desired result. The next two sections describe how we cope with different varieties of recursion and how the accumulation of side effects is represented. Section IV then illustrates how deviant cases are handled.

II A CLASS OF RECURSION SCHEMATA

In a recursive procedure such as INFECT, a side effect (conditional or unconditional) may occur logically at any of five locations, depending on the juxtaposition of the side effect, the recursive call, and the termination test. Here is a skeleton of the INFECT procedure, with the five possible locations of side effect occurrences shown underlined (not all five can coexist, and the SOLO user must make careful use of the control-flow keywords CONTINUE and EXIT to obtain certain combinations):

```
TO INFECT /X/
   ...
   <initial>
   ...
   CHECK /X/ KISSES ?Y
    If present: <pre-rec>; INFECT *Y; <post-rec>
    If  absent: <termination>
   ...
   <final>
```

These occurrences are depicted schematically in Fig. 3, which shows a recursive procedure P partitioned into its possible constituents. The effect of each solid box in the figure, in accordance with the notion of temporal abstraction, is to add or delete some set of triples. The overall effect of P will be the composition of these, as described in section III. 'Rec' is short for 'recursion', 'self' is the actual recursive invocation of P, and 'get-next' is an implicit enumeration function which retrieves the next node in the thread (this happens automatically during pattern-matching in SOLO, e.g. when Y is bound to BARBARA, then COLIN, then DIANA).

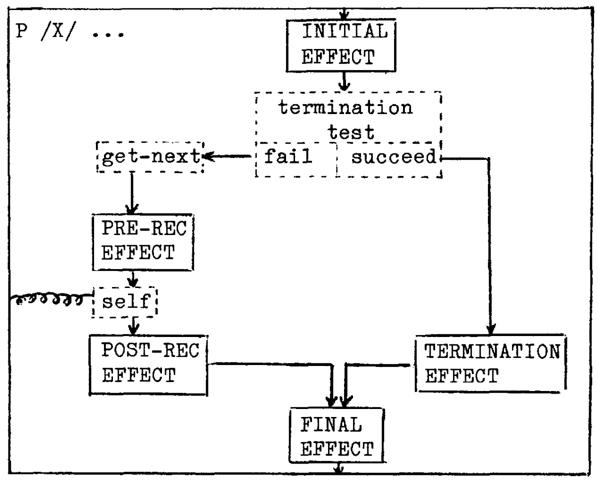

Fig. 3: The recursion schema for thread enumeration

Any of the effect steps may be conditional, composite, or missing altogether. The restrictions for recognizing a program as an instance of this schema are: (1) P has a parameter x which is the beginning of a thread; recursive invocations of P thus enumerate successive nodes along the thread; (2) the recursion and termination steps involve the enumerated node and relation R, where R is non-cyclic and one-to-one or many-to-one in the database (we don't deal with one-to-many mappings of R, which would exist if, say, the triples ANDY KISSES BARBARA and ANDY KISSES DIANA were both present in the data base); (3) the side effects do not alter the thread and the side effected node can only be reached from the enumerated node via one-to-one relations.

The effect steps can be classified according to time of execution and range of nodes on which the effect occurs:

RANGE OF ENUMERATION	TIME OF EXECUTION OF EFFECT: descent	ascent
entire thread	INITIAL EFFECT	FINAL EFFECT
butlast of thread	PRE-REC EFFECT	POST-REC EFFECT
last of thread	TERMINATION EFFECT	

The important insight of temporal abstraction is that the nodes enumerated during recursion can be dealt with as a set. Thus, any of these steps has an effect which can be represented as a set of triples (which we call 'db-set'). The termination step is the degenerate case of a singleton set. We describe db-sets as follows:

```
[db-set
  typical member: (<filter> => <side effect>)
  init: (<enumerated node> . <first value>)
  rec rel: <thread link>
  termination: (ABSENT [<ref> <thread link> ?])]

where
  <filter> ::= T ¦
               <simple filter> ¦
               (OR <conjunctive filter> ...)
  <simple filter> ::= (PRESENT <triple>) ¦
                      (ABSENT <triple>)
  <conjunctive filter> ::=
                   (AND <simple filter> ...)
  <side effect> ::=
           (+ [<enumerated node> <link> <node>]) ¦
           (- [<enumerated node> <link> <node>])
  <ref> ::= <enumerated node> ¦ (get <ref> <link>).
```

By (get <node> <link>) we denote the reference from <node> via <link>. Thus <ref> is the n-fold composition of the reference from the <enumerated node> (called e) along <link>. For example, the node BARBARA in Fig. 1 can be referenced by
  (get ANDY KISSES),
whereas the node DIANA can be referenced by the composition
  (get (get (get ANDY KISSES) KISSES) KISSES)
An instantiated schema does not, of course, refer to specific nodes in the data base, but rather to a generalised description of a typical node along the thread.

How, then, are db-sets derived from an instantiated recursion schema? We fill the slot 'typical member' from the effect of the step (e.g. a NOTE of some triple is a + with its first argument replaced by a <ref> involving e). If the effect is unconditional, the filter is T, otherwise the condition is taken as the filter. The 'init' slot is the first value that e will take. Enumeration stops when termination is true. Steps which work on the entire thread have

(ABSENT [e <thread link> ?])

as their termination condition, whereas those working on the butlast of the thread have

(ABSENT [(get e <thread link>) <thread link> ?])

as their termination condition.

Consider the instantiated schema for SOLUTION-1 above, which contains only an initial (conditional) effect. Its effect description is as follows:

```
[db-set
  typical member: ((ABSENT [e IS INOCULATED]) =>
                                   (+ [e GETS FLU]))
  init: (e . x)
  rec rel: KISSES
  termination: (ABSENT [e KISSES ?])]
```

III COMPOSITION OF RECURSIVE SETS BY SYMBOLIC EVALUATION

In general, a recursive procedure may comprise some combination of effect steps. To describe the net effect of the entire procedure, the individual effects must be composed. For instance, the addition of a set of triples followed by the conditional deletion of some elements should have the composite effect of asserting some elements of the set conditionally upon the negation of the condition for deletion (e.g. see SOLUTION-2 below). This simplification is done during symbolic evaluation.

Although the db-set for each step is a temporal abstraction (and hence ignores the order in which the nodes are enumerated), compositions of side-effects are sensitive to temporal order. Thus, we first compose those effects which occur on the descent and then those which occur on the ascent. The effects are still dealt with as db-sets, i.e. the inner details of the enumeration sequence are ignored. All we need to worry about is whether the db-set is of the 'descending' or 'ascending' variety, which we know from its postion in the instantiated schema.

Composition proceeds as follows: At a node in the symbolic evaluation tree (called S-node) where a db-set is to be asserted, we grow a branch with a description of the range of values that could be taken by the enumerated node. The range {e | e in R*(x)} says that e can take the value of all the nodes in the transitive closure of R starting at x. At the S-node at the end of this branch, the typical member of the db-set is asserted. If there is a (non-T) filter, we split into two branches, one with the condition and the other with its negation, and add the side effect to the S-node at the end of the branch where the condition is true.

The next db-set is dealt with in the same way at all terminal S-nodes grown so far. If it has a filter, we test it at each S-node: if the sets have the same range, it may be possible to show either that the condition must always hold or that it can never hold, thus saving the split. If the ranges overlap, we introduce one branch for the overlapping range and others for the non-overlapping parts. On the overlapping sets it is then possible to test conditions or apply rules for cancellation (i.e. a NOTE followed by a FORGET of the same triple has no net effect) and overwriting (i.e. a NOTE following another NOTE with the same triple has no further effect).

A special case arises if two consecutive assertions of sets are interleaving (i.e. they have the same time of execution of effect), and the second set's enumerated node (e2) 'runs ahead of' the first set's enumerated node (e1). We say that e2 'runs ahead of' e1 if e2 is an n-fold composition of the reference from e1 along the thread-link. Since the effect on e2 occurs before the effect on e1, we reverse the order of composition of both sets and apply the simplifications described above. Finally, starting from the terminal S-nodes we collect effects and conditions for all nodes with the same net-effect into one db-set with the appropriate filter. The result is the description of the program used for comparison with the ideal effect description.

IV EXAMPLES

Here are two alternative solutions to the problem posed in section 1. One has a bug.

SOLUTION-2 (Successive sweeps):

```
TO INFECT /X/
1 CONTAMINATE /X/
2 DECONTAMINATE /X/

TO CONTAMINATE /X/
1 NOTE /X/ GETS FLU
2 CHECK /X/ KISSES ?Y
 2A If present: CONTAMINATE *Y; EXIT
 2B If  absent: EXIT

TO DECONTAMINATE /X/
1 CHECK /X/ IS INOCULATED
 1A If present: FORGET /X/ GETS FLU; CONTINUE
 1B If  absent: CONTINUE
2 CHECK /X/ KISSES ?Y
 2A If present: DECONTAMINATE *Y; EXIT
 2B If  absent: EXIT
```

SOLUTION-3 (Ascending conditional side effect):

```
TO INFECT /X/
1 CHECK /X/ KISSES ?Y
 1A If present: INFECT *Y; CONTINUE
 1B If  absent: EXIT
2 CHECK /X/ IS INOCULATED
 2A If present: EXIT
 2B If  absent: NOTE /X/ GETS FLU; EXIT
```

In SOLUTION-2, the plan diagram for INFECT matches a schema for a conjoined effect. The plan diagram for the first of these, CONTAMINATE, matches the recursion schema for thread enumeration: an unconditional side effect occurring only in the 'initial effect' slot of the schema. It thus has the net effect:

```
for all {e ¦ e in KISSES*(x)}
  (+ [e GETS FLU])
```

The second of the conjoined effects, DECONTAMINATE, matches the same schema, except that the side effect in the 'initial effect' slot is conditional. This yields the following net effect description:

```
for all {e ¦ e in KISSES*(x)}
  (PRESENT [e IS INOCULATED]) => (- [e GETS FLU])
```

During symbolic evaluation, the two S-nodes are recognized as having the same range, so the evaluator simplifies their combined effect to be:

```
for all {e ¦ e in KISSES*(x)}
  (ABSENT [e IS INOCULATED]) => (+ [e GETS FLU])
```

which is precisedly the intended effect of the ideal INFECT procedure.

SOLUTION-3 has a plan diagram which matches that of the recursion schema with just a (conditional) 'post-rec' effect. The reader may wish to verify that this conforms with the skeleton depicted at the beginning of section II and the schema shown in Fig. 3 (the CONTINUE at step 1A in effect places step 2 in the 'post-rec' position). Notice that in this solution the first thing that normally happens is the recursive invocation of INFECT (step 1A), which means that the side effect only happens when ascending. Our schema knows that a 'post-rec' effect on its own fails to reach the final node in the thread (in this case it is due to the EXIT at step 1B). This is instantiated from the schema's canned effect description as follows:

```
for all {e ¦ (get e KISSES) in KISSES*(x)}
  (ABSENT [e IS INOCULATED]) => (+ [e GETS FLU])
```

That is, the conditional side effect is perpetrated only on the butlast of the thread running from x via KISSES. For the example of Fig. 1, SOLUTION-3 happens to work. However, a counter-example can be generated for the student in the following way: generate a thread in which the final node (f) of the thread satisfies the condition specified in the schema's effect description, i.e. (ABSENT [f IS INOCULATED]). For Fig. 1, this would amount to deleting the IS link between DIANA and INOCULATED. In such a case, SOLUTION-3 will fail. This counter-example can be used to point out the inherent flaw in the student's solution.

V CONCLUSION

Temporal abstraction provides us with a powerful mechanism for reasoning about side effects on sets of data objects. Our method of composing several such sets lets us analyse recursive procedures involving side effects on threaded data structures. Static set descriptions, which are the essence of temporal abstraction, can be combined using simplification rules derived from symbolic evaluation techniques. A library of recursive schemata enables us to analyse a range of students' programs, to combine set descriptions in a sensible way (e.g. composing descending and ascending effects in the right order), and to generate tailor-made counter-examples for programs which might 'work' by accident.

REFERENCES

[1] Eisenstadt, M. Artificial intelligence project. Units 3/4 of Cognitive psychology: a third level course. Milton Keynes: Open University Press, 1978.

[2] Eisenstadt, M. & Laubsch, J. Towards an automated debugging assistant for novice programmers. Proceedings of the AISB-80 conference on Artificial Intelligence, Amsterdam, 1980.

[3] Laubsch, J. & Eisenstadt, M. Domain specific debugging aids for novice programmers. Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81). Vancouver, BC, CANADA, 1981.

[4] Rich, C. & Shrobe, H. Initial report on a LISP programmer's apprentice. IEEE Transactions on Software Engineering, SE-4:6, 1978.

[5] Waters, R.C. A method for analyzing loop programs. IEEE Transactions on Software Engineering, SE-5:3, 1979.