SYMBOLIC EVALUATION OF LISP FUNCTIONS WITH SIDE EFFECTS FOR VERIFICATION

Dennis de Champeaux Jos de Bruin

Bedrijfsinformatica, Faculty of Economics, University of Amsterdam

ABSTRACT

In this paper we present a symbolic evaluator of LISP functions. It can handle data-altering functions of the RPLACA type, i.e. functions that change one datastructure by replacing parts of it by other structures that will themselves not be changed further, at least not permanently. The state description language uses first-order predicate calculus. It is argued that symbolic evaluation in terms of this language, although theoretically adequate, is not feasible in general, since it may require extremely complicated specifications for real-life functions with side effects. Examples are given of the specifications needed to verify several versions of SUBSTAD, a non-copying SUBST.

Keywords; LISP, symbolic evaluation, verification of data-altering functions, predicate calculus, frame problem.

1, Introduction

In 1978 we published SUBSTAD, a non-copying version of SUBST [1]. Comparison of these two functions in the context of a unification algorithm showed some very favorable results. Two years later we found out that the results were biased by a bug in our machine implementation of SUBST.

This experience increased our interest in verification, in particular of functions with side effects, such as SUBSTAD. These functions pose a challenge to verifiers. One simple RPLACA can have consequences for every datastructure around.

Very few practical, ready-to-use techniques are available at present. The theoreticians of program verification (for an overview, see [5]) are developing languages (Dynamic Logic e.g.) that abstract away from real application, concern toy-like programming languages and tend to be considered as interesting objects by themselves.

More promising seem concrete efforts like that of Topor [8], who verified the correctness of the Schorr-Waite marking algorithm, an algorithm somewhat similar to SUBSTAD. His proof by hand is reasonable to follow, but we are interested in actually automating the verification process as much as possible.

We developed a program that can keep track of the many details involved when checking all possible branches of computation trees. We have chosen the method of symbolic evaluation [3,6], because it guarantees that every branch is visited and that all preconditions to operations are considered.

Symbolic evaluation requires the addition of input/output specifications to the program code and of invariants to each loop in that code. The code is evaluated with symbolic input values that conform to the input specification, producing a symbolic output value for each branch through the code. The symbolic evaluator should embody the semantics of the operators used in the code, in our case (at least) the subset of LISP primitives used in SUBSTAD. For each of those operators it should be able to transform the description of the state in which this operator is called into a description of the state it creates.

It has to be verified that all of the output values produced are in accordance with the output condition. This, as well as checking entry and loop conditions, can be done "manually" or by a theorem prover. Although we have been experimenting with COGITO, our theorem prover (for results see [2]), our conoern here is the automatic updating concerning funotions with side effects, like RPLACA. For details on the actual proofs (by hand) see [2].

2. The state Description Language

In order to facilitate deduction, the state description language uses first-order predicate calculus. We start off with a countable domain of cells C and a countable domain of atoms A, where C and A are disjunct. Let D be their union: D = C U A. We will have the partial functions:

— car and cdr, with domain C and range D; and
— addr, with domain D and range N, the natural numbers.

We will have the partial predicate:

— atom with domain D, and which, where defined, coincides with the characteristic predicate of A.

Using the addr-function, we define the relation eqa with:

```
(d)(e){ eqa(d,e) <--> addr(d)=addr(e) },
```

for d, e in D where addr is defined.

<u>AXIOM 1</u>

```
(d)(e){ eqa(d,e) --> [ atom (d) --> d=e ] }, for d,
e in D.
```

<u>AXIOM 2</u>

```
(d)(e){ [ ~atom(d) & eqa(d,e) &
          car(d) = car(e) & cdr(d) = cdr(e) ] -->
d = e };
```

for d, e in D. Axiom 1 ensures that e is also non-atomic.

We define a <u>data object</u> D_r to be an element of the power set of D:

1) with D_r of finite size,
2) with C_r and A_r the elements of D_r respectively in C and A,
3) with car(C_r) and cdr(C_r) subsets of D_r, and
4) with a unique element r in D_r, the root of D_r, which has the property that all other members of D_r can be reached from r by finite car/cdr chains.

From now on we mention data objects by referring to their roots.

Recursive definitions on data objects run the risk of being undefined due to infinite regress, since data objects may contain cycles - a cell can reach itself along a car/cdr chain. The finiteness of data objects is the way out of this problem. Most recursive definitions that we will give in the sequel apply to data objects that have the special format of a tree. For generalizations to arbitrary data objects, see [2].

Recursive definitions on trees invoke in proofs an appeal to the so-called car/cdr induction. Whenever a formula P(x) reduces to a formula P(car(x)) and/or P(cdr(x)) then car/cdr induction allows the conclusion that P(x) has been inferred. This is justified by the observation that a well founded relation can be constructed (in most cases the number of cells reachable from x) that decreases on each recursive reference. Handled carefully, this also applies to recursive definitions with non-tree arguments.

Next we give definitions of the predicates partof and loopfree. The definition of partof works only on trees. (+ is the disjunction connective):

```
(d)(e){ partof(d,e) <-->
        [ partofcar(d,e) + partofcdr(d,e) ] }
(d)(e){ partofcar(d,e) <-->
        [ ~atom(e) &
          ( d=car(e) + partof(d,car(e)) ) ] }
(d)(e){ partofcdr(d,e) <-->
        [ ~atom(e) &
          ( d=cdr(e) + partof(d,cdr(e)) ) ] }
```

```
(d){ loopfree(d) <--> loopfree1(d,0) }
(d)(V){ loopfree(d,V) <-->
        [ atom(d) +
          { ~(d in V) &
            loopfree1(car(d),{d} U V) &
            loopfree1(cdr(d),{d} U V) } ] }.
```

The expression partof(d,e) signifies that the data object e contains a cell or atom identical to the root of d. Loopfree defines the property that a data object does not contain a cycle.

A <u>state description</u> is a conjunction of facts referring to a finite number of data objects, always containing the data objects nil and t, corresponding with NIL and T, members of A, for which holds: atom(nil), atom(t) and ~(t*nil).

A state description may refer to 'virtual* data objects, which existed during earlier states. Two data objects are compatible, if they can co-exist:

```
(d)(e){ compatible(d,e) <-->
        [ atom(d) + atom(e) +
          ( eqa(d,e) & d=e ) +
          ( ~eqa(d,e) &
            compatible(d,car(e)) &
            compatible(d,cdr(e)) &
            compatible(car(d),e) &
            compatible(cdr(d),e) )]}.
```

When two data objects are non-compatible at least one has to be virtual. The RPLACX operations are responsible for making data objects virtual.

DEFINITION: An <u>alist</u> is a finite list of pairs $((a_1,r_1),\ \ldots,\ (a_n,r_n))$ with a_i atoms unequal nil and r_i the roots of data objects, while for each pair r_i, r_j we have: compatible(r_i,r_j).

The alist contains the current bindings of the atoms. A data object is virtual with respect to an alist if it is non-compatible with an r^ from that alist. An atom may occur more than once as a first element of a pair, for Instance as a oonsequence of recursion. LISP functions retrieve and update leftmost occurrences. Side effects may propagate to the right in the alist. Extensions and contractions, as a consequence of entering a higher or lower stack level, also occur at the left.

DEFINITION: A s^ft oonflfiUriUofl is a Pair (AL,FL) with AL an alist and PL (the factlist) a state description. Atomicity of nil, t and all atoms a^ on the alist is implicitly assumed.

T. lufi SYrtQIlc. EYIIUftWr

When given LISP-code and a state configuration the symbolic evaluator generates a tree of state configurations, corresponding to all possible computation paths through the oode. The symbolic evaluator works like a real LISP evaluator. It has a code pointer, corresponding to a program counter, to that part of the code which has to be executed, it contains modules which correspond to built-in LISP funotions and it knows what to do with user defined functions.

A non-numerical atomic form is evaluated by retrieving the most recent (i.e. leftmost) binding from the current alist.

For built-in functions, the recipe consists of checking whether preconditions, parametrized for the current arguments, are fulfilled and, if the check succeeds, updating the state configuration. An exception is made for COND. The COND-module generates one or more bifurcations of the current state configuration. The correctness of a bifurcation (satisfiability of a test expression and its negation) is not proven by means of the deduction machinery but by constructing or having available two models that possess opposite truth values with respect to the test expression but are both consistent with the current state configuration. To construct these models one could ask the user to provide several examples, which are processed concurrently with the symbolic input specification for the code (not implemented). Testing by running examples and formal verification should not be seen as mutually exclusive, but should go hand in hand.

Modules are implemented for the following subset of standard LISP functions: ATOM, CAR, CDR, COND, CONS, EQ, EQUAL, GO, NOT, NULL, PROG, PROGN, QUOTE, RETURN, RPLACA, RPLACD and SETQ. The functions COND, GO, PROG, PROGN, QUOTE and SETQ are of type FSUBR, i.e. evaluation of their arguments is to their own discretion. The other functions have automatic - left to right - argument evaluation before module-specific actions are taken.

An essential requirement for the modules is that the compatability property of state configurations is preserved. Our only worry is RPLACA, RPLACD and SETQ beoause only those functions affect the alist. We will describe some of the modules.

ATOM
Let the argument of ATOM evaluate to x. A new symbolic value will be generated, say g1, which will be returned as the value, while the fact list will be expanded with:
{ g1=t & atom(x) } + { g1=nil & ~atom(x) }.

The implemented version deals immediately with the atomicity of x. It returns t or nil when atomicity or non atomicity of x can easily be derived from the given fact list, otherwise the user is asked to Indicate whether t, nil or both possibilities are to be pursued. In this last case, it generates a bifurcation of the current computation branch with t in one and nil in the other branch, adding either atom(x) or ~atom(x) to the respective factlist.

CAR (and analogously CDR)
Let the argument of CAR evaluate to x. In contrast with ATOM there is a precondition check for CAR: ~atom(x) should be derivable from the current fact list. If that derivation succeeds a new symbolic value, say g2, is generated and returned and g2«car(x) is added to the faot list.

COND
This function leads to bifurcation(s) of the current computation branch, as described for the implemented version of ATOM.

CONS
Let the arguments of CONS evaluate to x and y. A new symbolic value, say g3> is generated and will be returned, while the fact list will be extended with: ~atom(g3), car(g3)=x and edr(g3)*y.

GO
We assume only backward jumps. The loop invariant associated with the label to which GO refers, provided by the user and parametrized for the current bindings by the evaluator, should be derivable from the current fact list. A non-looping check, based on a well founded relation should also be performed. Because jumps are always backwards, we do not have to consider the current computation branch any further.

RPLACA (and analogously RPLACD)
Let the arguments of RPLACA evaluate to x and y. The precondition for RPLACA is ~atom(x). A new symbolic value, say g6, is generated and returned, while the fact list is extended with: eqa(x,g6), car(g6)*y and cdr(g6)=cdr(x).
Any non-atomic binding z1 on the alist, identical to x or •above* x, will be affected indirectly by the RPLACA operation and has to be replaced by a new binding z2 for which minimally holds: eqa(z1,z2). In general: when a RPLACX operation causes x1 to be replaced by x2 then each binding on the alist, y1, will be replaced by a fresh binding, y2, while the fact list will grow with: eqaupto(y1,y2,x1,x2), which says: y2 is identical with y1 unless there is a substructure of y1 that is identical with x1. The predicate eqaupto is defined as:

```
(y1)(y2)(x1)(x2){ eqaupto(y1,y2,x1,x2) <-->
[ eqa(y1,y2) &
  {y1=x1 --> y2=x2} &
  {[~(y1=x1) & ~atom(y1)] -->
   [eqaupto(car(y1),car(y2),x1,x2) &
    eqaupto(cdr(y1),cdr(y2),x1,x2) ]}]}.
```

Remark: When the original binding y1 is atomic then according to axiom 1 the new binding y2 is identical with y1.

<u>**LEMMA 1**</u>
{x1=x2 & eqaupto(y1,y2,x1,x2)} --> y1=y2.

<u>**LEMMA 2**</u>
{~(x1=y1) & ~partof(x1,y1) & eqaupto(y1,y2,x1,x2)} --> y1=y2.

These lemmas oan be used to curb updating activities. For proofs of these and other lemma's and theorems, see [2].

<u>THEOREM 1</u> Let y1 and z1 be old bindings which are respectively replaced by y2 and z2 due to an RPLACX-operation that caused x1 to be changed into x2, thus with eqa(x1,x2), then compatible(y1,z1), eqaupto(y1,y2,x1,x2) and eqaupto(z1,z2,x1,x2) implies compatible(y2,z2).

SETQ

Let the second argument evaluate to x. The precondition for SETQ is that the non-evaluated first argument is atomic, say A. The binding of the leftmost occurrence of A on the allst will be replaced by x. If A does not occur on the alist - i.e. when A is a global variable - then (A.x) will be added at the righthand side of the allst. Preservation of alist-compatability is ensured when the evaluation of the second argument yields a value compatible with the current bindings.

The modules not described trigger obvious updatings. (For the equal predicate needed by the EQUAL module, see [2].)

3.1, User Functions

Most LISP functions to be verified will contain functions other than the above mentioned primitive ones. These are provided either by the user or are built-in. They can be handled by the evaluator if they are accompanied by an input and an output condition.

The symbolic evaluator first asks for (and tries to assist with) a check that the input condition is fulfilled and then looks whether the user wishes this function to be verified. If so, she will have to provide its body. Recursive user functions will be opened at most once, for obvious reasons. A well-founded relation, user provided, should be used when verifying that arguments of a recursive call score strictly less with respect to that well-founded relation than the arguments at the top level call. This was not implemented.

An output condition should describe the resulting state in terms of the values used in the input condition to enable the symbolic evaluetor to update the state configuration in which the function was called. This updating is straigthforward when the function does not have side effects and just returns a value, but built-in and user functions of RPLACX-type need even more complicated alist updating schemes than the one given above for RPLACX.

Suppose we execute (NCONC LIS S1), where the bindings of LIS and S1 are respectively lis and s1. The rightmost leaf of SI, which must be NIL, will be replaced by a pointer to its second argument S1. Any datastruoture containing a pointer to lis or to a cell lying on its 'spine' (i.e. the cdr chain starting at lis) will be changed as a consequence of this NCONC operation.

We will describe an alist update scheme for a class of side effect generating functions, including NCONC, EFFACE and our SUBSTAD support functions SUBSTAD1 and SUBSTAD2. It applies to those functions which cause replacement of a cell, say x1, by a cell, say x2, (thus we have eqa(x1,x2)).

Every binding, z1, on the alist is replaced by a fresh binding, z2, and the fact list is expanded with: transf(z1,z2,x1,x2). The predicate transf and its supporting predicate tr1 and tr2 works by double recursion. First, it is checked whether z1 is identical with x1 or - using tr1 - with a cell reachable from xl. If the tr1-case applies the predioate tr2 is invoked to relate z1 and z2. Second, when z1 is not identical with x1 or a subcell of x1 then transf is called recursively to test whether subcells of z1 are affected by the x1-x2 replacement.

The predicate transf is defined as:

```
(y1)(y2)(x1)(x2){ transf(y1,y2,x1,x2) <-->
[ eqa(y1,y2) &
  {x1=y1 --> y2=x2} &
  {[~atom(y1) & ~(x1=y1) & tr1(y1,x1,x2)] -->
   tr2(y1,y2,x1,x2)} &
  {[~atom(y1) & ~(x1=y1) & ~tr1(y1,x1,x2)] -->
   [transf(car(y1),car(y2),x1,x2) &
    transf(cdr(y1),cdr(y2),x1,x2)]}]},
```

with tr1 defined as:

```
(y1)(x1)(x2){ tr1(y1,x1,x2) <-->
[ ~atom(x1) &
  eqa(x1,x2) &
  {y1=x1 +
   tr1(y1,car(x1),car(x2)) +
   tr1(y1,cdr(x1),cdr(x2))}]},
```

and with tr2 defined as:

```
(y1)(y2)(x1)(x2){ tr2(y1,y2,x1,x2) <-->
[{y1=x1 --> y2=x2} &
 {~(y1=x1) -->
  [{tr1(y1,car(x1),car(x2)) -->
    tr2(y1,y2,car(x1),car(x2))} &
   {tr1(y1,cdr(x1),cdr(x2)) -->
    tr2(y1,y2,cdr(x1),odr(x2))}]}]}.
```

The meaning of the transf(z1,z2,x1,x2) formula can be phrased as: let y1 be z1 or a subcell of z1, let u1 be x1 or a subcell of x1, while u1 has been replaced by u2 (so u2 is identical with x2 or with a subcell of x2), then, when y1 is identical with u1, there is a corresponding cell in z2, which is identical with u2.

In analogy with lemma 1 and lemma 2, we have:

<u>LEMMA 3</u>

{x1=x2 & transf(y1,y2,x1,x2)} --> y1=y2.

LEMMA 4

```
[(z){[z=x1 + partof(z,x1)] -->
     [~(z=y1) & ~partof(z,y1)]} &
 transf(y1,y2,x1,x2)] -->
y1=y2.
```

THEOREM 2 Let y1 and z1 be old bindings which are respectively replaced by y2 and z2 due to an side-effect operation causing x1 to be changed into x2, thus with eqa(x1,x2), then eoapatible(y1,z1), transf(y1,y2,x1,x2) and transf(z1,*2,x1,x2) implies compatible(y2,z2).

The limitations of this updating scheme can be seen from the function NC0NC2, defined as:

```
(NC0NC2(LAMBDA(LIS1 LIS2 S1)
     (NCONC LISKNCONC LIS2 S1))))
```

A binding referring to the *spine' of the input binding of LIS2 cannot be recognized and therefore will not be updated, although it is not up-to-date anymore.

We conclude that the user must be given the option to specify a specific, idiosyncratic alist update mechanism for any funotion having side effects. This will considerably increase the verification burden, since it will have to be shown that the compatibility requirement for the updated alist is fulfilled.

4. Evaluting SVBSTAP

To give an impression of the feasibility of the method of symbolic evaluation as introduced above, we will discuss our effort to verify SUBSTAD. This function is called with three arguments: S1, LAT and S3. It replaces al occurrences of LAT in S3 by S1. The value of LAT should be a non-numeric Atom. This is checked by SUBSTAD, which also handles the case that S3 is atomic. Otherwise it calls a support function with one argument, S3.

The support function published in [1] uses pointer reversal to avoid the use of a stack, as is done in garbage collectors. Before discussing this function, we will make some remarks on the verification of two simpler versions, to show how the method works and to illustrate how a slight modification in a program can lead to substantial differences in its verification.

4.1. SUBSTAD1

First of all, the recursive SUBSTAD1:

```
(SUBSTAD1(LAMBDA(S3)(PROG2
     (COND((ATOM(CAR S3))
           (COND((EQ LAT(CAR S3))(RPLACA S3 S1))))
          (T(SUBSTAD1(CAR S3))))
     (COND((ATOM(CDR S3))
           (COND((EQ LAT(CDR S3))(RPLACD S3 S1))))
          (T(SUBSTAD1(CDR S3)))) ))).
```

The preconditions are:

- the binding of S3, say vs3, is not atomic;
- the binding of LAT, say lat, is atomic; and
- lat is not a leaf of the binding of S1, say vs1. This last precondition is meant to prevent the introduction of cycles.

To simplify the proofs, we will assume that vs1 does not share substructure with vs3. Consequently, lemma 4 will apply and therefore updating of the S1 binding will never happen. (When vs1 does share structure we can still Invoke lemma 2, since lat is not a leaf of vs1.)

Since we assume the preconditions to hold, the faot list will (implicitely) contain: atom(lat) & ~atom(vs3) * ~partof(lat,vs1).
The input allst is: ((S1.vs1) (LAT.lat) (S3.vs3)).
Assume the output allst to be: ((S1.vs1) (LAT.lat) (S3.nvs3)).
The output assertion to be verified will be:
replacedd(vs1,lat,vs3,nvs3)»
with replaoedd (replacement with potential destruction of vs3) defined as:

```
(x1)(x2)(x3)(ot){replacedd(x1,x2,x3,ot) <-->
[ eqa(x3,ot) &
  {atom(car(x3)) -->
   [(x2=car(x3) --> car(ot)=x1) &
    (~(x2=car(x3)) --> car(ot)=car(x3)) ]} &
  {~atom(car(x3)) -->
replacedd(x1,x2,car(x3),car(ot)) } &
  {atom(cdr(x3)) -->
   [(x2=cdr(x3) --> cdr(ot)=x1) &
    (~(x2=cdr(x3)) --> cdr(ot)=cdr(x3)) ]} &
  {~atom(cdr(x3)) -->
replacedd(x1,x2,cdr(x3),cdr(ot)) }]}.
```

There are 9 different paths through the code. We will work our way along one of the paths.

Initially the fact list contains:
atom(lat) & "atom(vs3) 4 -partof(lat,vs1).
Assuming that (ATOM(CAR S3)) yields T we get in addition:
xarcar(vs3) & atom(xa).
Assuming that (EQ LAT(CAR S3)) yields T we get:
latexa.
RPLACA generates a new value, say nv1, adding:
eqa(nv1,vs3) & car(nv1)«vs1 6 cdr(nv1)scdr(vs3).
The allst update scheme for RPLACA generates a new binding for S3, say ivs3, so the allst becomes:
((SI.vsD (LAT.lat) (S3.ivs3)),
while the fact list grows with:
eqaupto(vs3,ivs3.vs3,nv1).
Assuming that (ATOM(CDR S3)) yields NIL we get:
xdscdr(ivs3) * ~atom(xd).
The next action concerns the recursive call on the CDR. Its parametrized and simplified input condition:
"atom(xd) & atom(lat) & "partof(lat,vsD,
is trivially satisfied. The function will not be opened, but instead the faot list grows with:

replaoedd(vs1,lat,xd,nxd) &
transf(ivs3,jvs3,xd,nxd),
while the allst changes into: ((S1.vs1) (LAT.lat) (S3.jvs3)).
The output assertion to be proven for this particular path is:
replacedd(vs 1,lat,vs3•Jvs3)•

We will not give proofs. The general strategy in this and following oases is a combination of subproblem recognition, case reasoning, expansion of recursive definitions and application of oar/cdr induction.

4.2. SUBSTAP2

The treatment of SUBSTAD1 as given above was slightly incorrect, although this did not affect the result. Upon entry of SUBSTAD1 the alist is in faot:
((S3.vs3) (S1.vs1) (LAT.lat) (S3.vs3)),
where the first occurrence of S3 cones from SUBSTAD1 and the second one from SUB3TAD. The output assertion of SUBSTAD1 did refer to the <u>second</u> occurrence of vs3- This more subtle treatment of the alist is essential for the half recursive half iterative support function SUBSTAD2.

```
(SUBSTAD2(LAMBDA(S3)(PROG(HH)
AGAIN
      (COND((ATOM(SBTQ HH(CAR S3)))
             (CONDUEQ LAT HHHRPLACA S3 S1))))
            (T(SUBSTAD2 HH)))
      (COND((ATOM(SETQ HH(CDR S3)))
             (COND((EQ LAT HH)(RPLACD S3 SI))))
            (T(SBTO S3 HH)
              (GO AGAIN)))
)))
```

Because of the assignment of the local S3 to its CDR Just before the loop, we no longer have a handle on the datastructure as a whole, to which we must be able to refer in order to specify the loop invariant and to enable a correct update of the calling environment after exiting SUBSTAD2. The problem is solved by refering to the <u>global</u> S3, the argument with which SUBSTAD2 is called. (In general a pre-prooessor should take care that all arguments given to user defined functions are expllcltely assigned on the alist.)

Verifying SUBSTAD2 requires deducing the loop invariant when oontrol reaches the label AGAIN upon entering the function, deducing the output assertion for six paths through the oode and deducing the loop invariant for three paths.

The input alist is as given above. The output alist, after exiting from SUBSTAD2 will be: US1.vs1) (LAT.lat) (S3.nvs3)).
The input and output assertion are the same as for SUBSTAD1. We have to provide a loop invariant with the label AGAIN. This loop assertion will refer to the current bindings of the variables, so we also have to specify an alist at the label:
((HH.vhh) (S3.1s3) (Sl.vsl) (LAT.lat) (S3.gs3))
The value ls3 is the local value of S3, and gs3 is the global value of S3. The loop assertion will be:
atom(lat) & ~atom(ls3) & ~atom(vs3) & ~partof(lat,vs1) &
spine(vs1,lat,vs3,gs3,ls3).

We will not give the definitions of spine and other support predicates. Giving a general description of the situation at the label is rather complicated, since it is not enough to say that every tree hanging off the spine above the local S3 has been checked and replaced if necessary. Structure sharing may have led to changes in the part of the tree that is still to be investigated. It may even have caused the replacement of the right most leaf of vs3 by a pointer to vs1, so S3 may eventually point to a cell for which there is no corresponding cell in the original vs3.

We will just give one definition as an example, for the others we again refer to [2]. The predicate sidefct is used to describe that xp and xq, which are parts of the not yet visited subtrees x3 and xl of the original (xo) and current (xn) incarnation, are the same unless structure sharing has led to side effects.

```
(xo)(xn)(x3)(xso)(xsn)(xp)(xq)
        { sidefct(xo,xn,x3,xso,xsn,xp,xq) <-->
[eqa(xp,xq) &
 {xsosx3 —>
  [{atom(car(xp)) —> car(xp)scar(xq)} &
   {~atom(car(xp)) —>
    sidefct(xo,xn,x3,xo,xn,car(xp),car(xq))} &
   {atom(cdr(xp)) —> cdr(xp)scdr(xq)} &
   {~atom(cdr(xp)) —>
    sidefct(xo,xn,x3,xo,xn,cdr(xp),cdr(xq))}]} &
 {~(xso=x3) —>
  [(car(xso)sxp —> car(xsn)sxq) &
   (~(car(xso)rxp) —>
    [ {tr1(xp,car(xso),car(xsn)) —>
      tr2(xp,xq,oar(xso),car(xsn))} &
     (~tr1(xp,car(xao),car(xsn)) —>
      sidefct(xo,xn,x3,cdr(xso),cdr(xsn),xp,xq)}]
```

Symbolic evaluation of SUBSTAD2 generates fact lists that are much longer than those generated for SUBSTAD1, since the alist in this case contains three arguments (HH, local S3 and global S3) that have to be updated after an RPLACX or a recursive call to SUBSTAD2. This, and the greater amount of predicates needed to specify the loop invariant, made verification of this function just barely feasible. The great difference in verification effort caused by a slight change in the code, challenges the claim that once a program is verified, modifications will require very little additional effort.

<u>4.3. SUBSTADP</u>

The disparity between amount of oode and amount of ad hoc definitions is even worse for SUBSTADP:

```
(SUBSTADP(LAMBDA(S3)(PROG(EX HH)
     (SETQ EX $)
L2
     (SETQ HH(CAR S3))
     (COND((NOT(ATOM HH))
          (MARK S3 1)
          (RPLACA S3 EX)
          (SETQ EX S3)
          (SETQ S3 HH)
          (GO L2))
         ((EQ LAT HH)(RPLACA S3 S1)))
L4
     (SETQ HH(CDR S3))
     (COND((ATOM HH))
         ((NOT(EQ EX $))
          (RPLACD S3 EX)
          (SETQ EX S3)
          (SETQ S3 HH)
          (GO L2))
         (T(SETQ S3 HH)
           (GO L2)))
     (COND((EQ LAT HH)(RPLACD S3 S1)))
     (COND((EQ EX $)(RETURN)))
L5
     (SETQ HH S3)
     (SETQ S3 EX)
     (COND((MARKB S3)
          (MARK S3 0)
          (SETQ EX(CAR S3))
          (RPLACA S3 HH)
          (GO L4)))
     (SETQ EX(CDR S3))
     (RPLACD S3 HH)
     (GO L5)
)))?end of the pointer reversal SUBSTADP?
```

In this version, the use of a stack is avoided by reversing pointers, i.e. when the car or cdr part of a cell is non-atomic, this part is saved, while the car or cdr is replaced by a pointer back to the parent cell immediately above it. Marking is used to indicate whether the car or the cdr part of the cell contains the reversed pointer. The tree is searched in a depth first manner.

The code contains three labels, so in addition to the input and output assertion we have to set up three loop invariants. Describing the situation at the various loops is extremely complicated because of the much greater number of (temporary) replacements.

We defined the predicates that are necessary to describe the situation at one label, L2, assuming that vs1 is atomic. Even with this drastic simplification, we needed a staggering amount of definitions: eleven predicates, several of them with seven arguments and totalling nearly 200 lines of text (see [2]). To get an impression of what is Involved, glance at the definitions of two predicates, lb2at3 and its support lb2at5. They describe the subtrees hanging off the spine above the inverted pointer chain.

```
(vs1)(lat)(ex)(l3)(ol)(nw)
     { lb2at3(vs1,lat,ex,l3,ol,nw) <-->
[eqa(ol,nw) &
 {onichain(ex,nw) --> lb2at5(vs1,lat,ex,l3,ol,nw)}
&
 {~onichain(ex,nw) -->
  [{atom(car(ol)) -->
    [{car(ol)=lat --> car(nw)=vs1} &
     {~(car(ol)=lat) --> car(nw)=car(ol)}]} &
   {~atom(car(ol)) -->
    lb2at3(vs1,lat,ex,l3,car(ol),car(nw))} &
   {atom(cdr(ol)) -->
    [{cdr(ol)=lat --> cdr(nw)=vs1} &
     {~(cdr(ol)=lat) --> cdr(nw)=cdr(ol)}]} &
   {~atom(cdr(ol)) -->
    lb2at3(vs1,lat,ex,l3,cdr(ol),cdr(nw))}]}]}.
```

This predicate mainly looks whether nw - which is already visited - is residing on the inverted pointer chain, which may be caused by structure sharing. If so the predicate lb2at5 will describe the situation.

```
(vs1)(lat)(ex)(l3)(ol)(nw)
     { lb2at5(vs1,lat,ex,l3,ol,nw) <-->
[eqa(ol,nw) &
 {ex=nw -->
  [{markb(nw) -->
    [replacedd(vs1,lat,car(ol),l3) &
     {atom(cdr(ol)) -->
      [{cdr(ol)=lat --> cdr(nw)=vs1} &
       {~(cdr(ol)=lat) --> cdr(nw)=cdr(ol)}]} &
     {~atom(cdr(ol)) -->
      replacedd(vs1,lat,cdr(ol),cdr(nw))}]} &
   {~markb(nw) -->
     {atom(car(ol)) -->
      [{car(ol)=lat --> car(nw)=vs1} &
       {~(car(ol)=lat) --> car(nw)=car(ol)}]} &
     {~atom(car(ol)) -->
      replacedd(vs1,lat,car(ol),car(nw))} &
     replacedd(vs1,lat,cdr(ol),l3)}]]} &
 {~(ex=nw) -->
  [{markb(nw) -->
    [{atom(cdr(ol)) -->
      [{cdr(ol)=lat --> cdr(nw)=vs1} &
       {~(cdr(ol)=lat) --> cdr(nw)=cdr(ol)}]} &
     {~atom(cdr(ol)) -->
      lb2at3(vs1,lat,ex,l3,cdr(ol),cdr(nw))} &
     (E icel){onichain(ex,icel) &
              lb2at5(vs1,lat,ex,l3,car(ol),icel) &
              [markb(icel) --> car(icel)=nw] &
              [~markb(icel) --> cdr(icel)=nw]}]} &
   {~markb(nw) -->
    [{atom(car(ol)) -->
      [{car(ol)=lat --> car(nw)=vs1} &
       {~(car(ol)=lat) --> car(nw)=car(ol)}]} &
     {~atom(car(ol)) -->
      lb2at3(vs1,lat,ex,l3,car(ol),car(nw))} &
     (E icel){onichain(ex,icel) &
              lb2at5(vs1,lat,ex,l3,cdr(ol),icel) &
              [markb(icel) --> car(icel)=nw] &
              [~markb(icel) -->
               cdr(icel)=nw]]}]}]}.
```

When nwsex, we can describe it with replaoedd, keeping in mind whether its car (markb) or its cdr ("markb) contains the back pointer.

If nw lies somewhere else on the Inverted pointer chain and the non-reversed pointer points to an atomic structure, describing this part is straigthforward. However, if it is non-atomic, we have to recursively Invoke Ib2at3, because structure sharing between that part of nw and the reversed pointer chain is again possible.

To describe the part originally pointed to by the now reversed pointer, we have to use existential quantification. We do not have a direct pointer to it, but we know were to start (at EX) and we know its unique identification: eqa(icel,car(ol)). This identification is part of Ib2at5.

Possible structure sharing similarly complicates the description of subtrees on the inverted pointer chain, under 13 or to the right of the inverted pointer chain.

5. Discussion

Although we were able to write a symbolic evaluator that can handle the functions we were interested in (and no doubt a host of others), it was not possible to give a completely general update algorithm to handle all RPLACX-type functions. We defined one for a common class, in which one datastrueture is changed by replacing certain subparts by other dataatructures that will not themselves be mutated before the function is exited (at least not permanently). To make the verifier a general one, it should allow the user to specify her own update procedures in other cases. Since compatibility will have to be proven by the user in those cases, this places a rather heavy burden on her.

The algorithm given is extremely oareful, replacing all bindings on the aliat after every call to an RPLACX-type function. This has its price. Updated bindings need potentially complicated proofs to show their invariance, even though it may be very obvious (to us) that in fact they could not have been changed at all. Of course one could keep the number of updated bindings down by incorporating the lemma's given above and other specific knowledge into the evaluator, but this would amount to pushing the problem around.

The attempt to give correctness proofs for several versions of SUBSTAD revealed that the method of symbolic evaluation - although theoretically adequate - flounders in some cases on a practical problem: formal description of input/output statements as well as loop invariants leads to a proliferation of ad hoc definitions. We expect this to hold for all currently available verification techniques. If so, verification specialists may be advlced to give more attention to the practical implications of their theories, instead of devoting all their energies to esoteric refinements, or even to the design of logics that become an end in themselves.

The bottle-neck lies in the necessity to specify in state-description terms what a function is supposed to do. Whether a funotion is recursive or not is not even explicitly expressible in such a specification. Somehow people feel more akin to a definition in procedural terms, such as "the terminals equal to lat will be <u>replaced</u> by vs1" and "the tree will be <u>visited</u> from left to right". Proving correctness of a function would then 'reduce' to showing that the function b<u>ehaves</u> according to expectations rather than that input/output description pairs conform to a certain relation.

The technique we have developed for describing evolving states using an aliat, a fact list and predictates like equaupto and transf that capture specific side effects, may be of interest to other areas of A.I.. The alist oan be considered a collection of individual concepts, where the bindings are the actual extensions. A new situation differs primarily in that some concepts have different extensions, which are described by fresh facts. Outdated facts do not have to be deleted but merely become invisible since they contain arguments not residing on the alist any longer.

This more procedural approach to the frame problem seems to have advantages over the strictly declarative method given in [7]. There is no need for wieldy axioms to express that when P(x,...,z,s1) holds in situation s1 and some conditions are fulfilled, the fact P(x,...,z,s2) can be inferred in s2. Instead we have a different problem. A fact may seem to be obsolete (since an argument has been removed from the alist) while an analogous fact can be inferred for a newly introduced extension. We have encountered this in lemma 1-4, where particular circumstances allow one to equate the old and new binding.

Since updatings and the recognition of identities are object centered and therefore may affect many facts simultaneously, this problem seems less obstructive than the original one, but more thinking and/or experimenting is needed to validate this suggestion.

Although we agree with De Millo e.a. [4] that the present verification tools do not lend themselves to practical use, we do not share their conclusion that the whole effort should be abandoned. Verifiers will probably always run into resource limitations, but to assume that they will never be able to use meohanisms similar to those that enable humans to circumvent some of these limitationa for certain tasks (without sacrificing preciseness) seems premature.

Finally, it pays to have a second look at one's program from a verification perspective.

Writing this paper forced us to reconsider the conditions under which the function SUBSTAD is applicable. The specification that we published five years ago turned out to be too liberal!

REFERENCES

[1] CHAMPEAUX, D. de, SUBSTAD: For Past Substitution in LISP, with an Application on Unification, Information Processing Letters, vol 7, no 1, January 1978, pp 58-62.

[2] CHAMPEAUX, D. de, Algorithms in AI, forthcoming thesis, Economische Paculteit, Universiteit van Amsterdam, 1981.

[3] DARRINGER, J.A. & J.C. KING, Applications of Symbolic Execution to Program Testing, IBM Report RC 6965, January 1977.

[4] DE MILLO R.A. et al, Social Processes and Proofs of Theorems and Programs, CACM, vol 22, no 5, May 1979, PP 271-280.

[5] HAREL, D., Proving the Correctness of Regular Deterministic Programs: A Unifying Survey Using Dynamic Logic, IBM Report 7557, March 1979.

[6] KING, J.C, Symbolic Execution and Program Testing, CACM, vol 19, no 7, July 1976, pp 385-394.

[7] MCCARTHY, J. & P.J. HAYES, Some Philosophical Problems from the Standpoint of Artificial Intelligence, in B. Meltzer & D. Michie (Ed), Machine Intelligence 4, Elsevier NY, 1969, PP 463-502.

[8] T0P0R, R.W., The correctness of the Schorr-Waite List Marking Algorithm, Acta Informatica, vol 11, pp 211-221, 1979.