CALCULATING PROPERTIES OF PROGRAMS BY VALUATIONS ON SPECIFIC MODELS

Michel Sintzoff
MBLE Research Laboratory, Brussels

Abstract

The proof that a program verifies some property is carried out by the valuation of the program in a model characterizing that property. Specific models are given for sufficient conditions of the correctness of types, locations and asynchronous computations ; a hypothetical programming language is used, which includes functions and locations and allows their recursive composition. The application of the method in studying termination or correctness problems is discussed on particular programs.

1. Introduction

A property of a program may be verified by computing this program in a model (i.e. interpretation, valuation, semantics) corresponding to that property. The values are then not all the normal values like numbers or data structures, but well objects abstracting the considered property ; similarly, the functions are then defined as mappings on these objects.

For example, a×a+b×b yields the value 25 when a is 3 and b is -4, and when + and × are the arithmetic multiplication and addition. But a×a+b×b yields always the object "pos" when a and b are the objects "pos" or "neg", and when the valuation is defined as follows :

```
pos+pos=pos               pos×pos=pos
pos+neg=pos,neg           pos×neg=neg
neg+pos=pos,neg           neg×pos=neg
neg+neg=neg               neg×neg=pos
V(p+q)=V(p)+V(q)          V(p×q)=V(p)×V(q)
V(0)=V(1)=...=pos
V(-1)=V(-2)=...=neg
```

The valuation of a×a+b×b yields "pos" by the following computations :

```
V(a)=pos,neg                 V(b)=pos,neg
V(a×a)=pos×pos,neg×neg       V(b×b)=pos×pos,neg×neg
     =pos,pos=pos                 =pos,pos=pos
V(a×a+b×b)=V(a×a)+V(b×b)=pos+pos=pos
```

This valuation proves that the result of a×a+b×b is always positive and hence allows to compute its square root without any preliminary dynamic test on its sign. On the other hand, the proposed valuation checks only sufficient conditions : it fails to prove the positive character of (a+b)×(a+b)-b×b, for which a more refined model is needed.

This paper applies this simple idea to the check of properties of programs written in the language introduced hereafter.

2. A hypothetical programming language

Values :

-true,false,0,1,2,...,-1,-2 ;
-routines ; locations.

Identification :

-identifiers may be defined : iden y
-a value v may identified : y is v ;
-the value of the identifier y is then v

Actions :

-standard operations and relations on numbers, Booleans ;
-abstraction, yielding a routine :
  λ.t, λx.t, λ(x,y) . t, etc. ;
-application:f( ),f(a),f(a,b),etc. ;
 if f is λz.t then f(a) equals
 (iden zi ; zi is a ; subst(zi,z,t))
 where zi is a new identifier and subst yields the result of substituting zi for z in t.
-to create a new location : loc
-to delete a location r : del(r)
-to assign a value v to a location r:r:=v
-to fetch the value assigned to a location r : fetch(r), which yields v if r:=v is still acting.

Composition rules :

-composition of action g to action f :
 f ; g
-selection between f and g according to the Boolean value b :
 if b then f else g fi ;
-any composition may be parenthesized
-any action yields a value, the last one computed in it.

Remarks :

-no data structures ; they may be represented by functionals ;
-no program points, jumps, parallelism or semaphores ;
-no action is implicitly defined by context ;
 x is loc ; y is loc ; y:=3 ;
 x:=y assigns the location y to x ;
 x:= fetch(y) assigns the number 3 to x .

3. Types

The valuation defined below checks the compatibility of the types of values through assignations, applications etc.

Values (v,v1,v2, ... denote values) :

-1,loc
-routines

Relationships (id, id' denote identifiers) :

id=v, id=id', id:=v

Valuation rules (Va is the result of V(a)) :

```
V(true)=V(false)=V(0)=V(1)= ...=1
V(iden x)=x
V(x is a)=(x=Va) if Va is not loc ,
         =(x=a) if a is(an identifier of) a loc
V(x)=v, when x=v exists
V(loc)=loc
V(del(r))=del(id(r))
 id(r)=r if r=loc or r:=v ,
      =id(s) if r=s  and s is an identifier
del(y) : delete all relationships using y
V(a:=b)=(id(a):=Vb)
v:=(v1,v2) equals (v:=v1, v:=v2)
(v1,v2):=v equals ((v1:=v,v2),(v1,v2:=v))
V(fetch(r))=t, when r:=t exists
```

```
V(λx.f)=λx.f
V((λx.f)(a))=V(iden xi;xi is a; subst(xi,x,f))
V(f;g)=V(f) before V(g)
V(if b then f else g fi)=(V(b;f),V(b;g))
```

Induction rule :

Compute the valuation as long as changes are obtained in the relationships.(A consistent substitution of identifiers or locations is not considered as a change.)

Example 1 (in all conditionals of all examples, b stands for any Boolean expression without side-effects, for simplicity) :

```
1. iden m,n,x,y,f,g ;
2. m is 105;
3. n is if b then m else 14 fi ;
4. x is loc ; y is loc ; g is loc ;
5. f is λu. if u≤0 then 1
            else f(u-1)×u fi ;
6. g:= λ(r,s).
      (1. iden z; z is loc ;
       2. z:= fetch(r) ;
       3. r:= fetch(s) ; s:= fetch(z)) ;
7. x:=m ;
8. y:=f(n) ;
9. (fetch(g))(x,y) ;
10.del(g);
11.g is λz.if b then f else  λx.(x+z) fi ;
12.x:=g(n)(m) ;
```

If we want to prove that, in this program,m, n are integers and x,y are locations of integers, we just have to check that the set

R=(m=1, n=1, x:=1, y:=1)

is not disproved during the computation of the valuation.

The steps of this computation are detailed below; the valuation of each statement is summarized by giving the number of this statement followed by the valuation of the statement. E.g. the valuation of statement 2, viz.m is 105, yields m=1, which may be read as "the identifier m is now an integer" ; after this statement, the state of the computation is represented by (m=1,n,x,y,f,g) which combines the effects of the statements 1 and 2 ; this state is of course compatible with the relationships R which are to be verified. The valuation of the statement 3 goes as follows : V(n is if b then m else 14 fi) yields n=V(if b then m else 14 fi)

```
        =(V(b;m), V(b;14))
        =(V(b)V(m), V(b)V(14))
        =(V(m),V(14)) because V(b) has no side-
         effect and because the value of V(b;m)
         is the last value computed in it,
         i.e. V(m)
        =(1,1)=1
```

Hence, after statement 3, the state is (m=1,n=1, x,y,f,g) which is compatible with R. The valuation proceeds on the subsequent statements, in the same automaton-like way ; note that the relationship x:=1 may be read as "the identifier x is now a location of an integer".

Valuation steps.

```
1   : m,n,x,y,f,g
2   : m=1 ; R is ok ;
3   : n=(1,1)=1 ; ok ;
4   : x=loc, y=loc, g=loc; ok ;
5   : f= λu.if u≤0 ... fi
6   : g:= λ(r,s).(iden z; ...)
7   : x:=1 ; ok ;
8   : y:=(iden u1; u1 is n ;
          if u1≤0 then 1
          else f(u1-1)×u1 fi)
i.e. y:=(1,f(1)×1), split   into (a) and (b) :
(a) y:=1 ; ok ;
(b) y:=f(1)×1 =(1,f(1)×1×1)
    no change, hence stop(induction rule)
9   : (0. iden r1,s1; r1 is x; s1 is y ;
       1. iden z ; z is loc ;
       2. z:= fetch (r1) ;
       3. r1:= fetch (s1) ; s1:= fetch(z))
9.0: r1=x, s1=y
9.1: z=loc
9.2: z:= fetch(x) i.e. z:=1
9.3: x:= fetch(y) i.e. x:=1 ; ok ;
     y:= fetch(z) i.e. y:=1 ; ok ;
10  : g:=λ... is deleted (cfr step 6)
11  : g = λ z.if b then ... fi ;
12  : x:=g(n)(m)
i.e. x:= (0. iden z1; z1 is n ;
          1. if b then f else  λx.(x+z1) fi)
          (m)
12.0: z1=1
12.1: x:=(λu ..., λx...)(m)
 (a)  x:=(λu...)(m)
      x:=1 (like step 8)
 (b)  x:=(λx.x+z1)(m)
      x:=(iden x1; x1 is m ;
         x1+z1)
      x:=(1+1)
      x:=1
12   : x:=1 ((a) and (b)) ; ok
```

4. Fetching correct locations

The goal is to check that each location is fetched only after   a value has been assigned to it.

Values :

-1,w1,w2,w3, ...,
-identifiers, routines

Relationships :

id=v, v:=v'

Valuation rules :

```
V(x is a)=(x=Va)
V(loc)=wi, a new one
V(del(r))=del(V(r))
V(a:=b)=(Va:=Vb)
V(fetch(r))=v, ok if r:=v exists
```

The other valuation rules and the induction rule are the same as in Section 3.

Example 2.

```
1. iden x,y,u,v,f ;
2. x is loc, y is loc, u is loc, v is loc ;
3. if b then u else v fi :=x ;
4. f is  λr.if b
     then u:=loc ; v:=fetch(u); del(u)
     else v:=r; f(y) fi
5. del(x) ;
6. f(y) ;
7. y:= fetch(v) ;

1 : x,y,u,v,f
2 : x=w1, y=w2, u=w3, v=w4
3 : (w3, w4):=w1, hence
    x=w1, y=w2, ((u=w3:=w1, v=w4),
              (u=w3, v=w4 := w1))
4 : f= λr.if b then ... fi
5 : y=w2, u=w3, v=w4
    because w1 is deleted
```

```
6 : f(y)=(y=r1=w2,v=w4:=w5),
        (y=r1=w2, u=w3, v=w4:=w2, f(y))
    the second part yields similarly
    (y=r1=r2=w2, v=w4:=w6),
    (y=r1=r2=w2, u=w3, v=w4:=w2, f(y))
    there is no new results (the differences
    between r1 and r2, w5 and w6 do not matter),
    hence the recursion stops ; the result is
    y=r1=w2, v=w4:=w5
7 : y=r1=w2:=w5, v=w4:=w5 ; ok
```

Remark. This kind of valuation may be used to answer the question : for any statement, which data items does it access? For example, the program above may be analysed as follows :

```
3 : access to u,v,x,w1,w3,w4
4 : λr ...  accesses u,v,y
6 : access to u,v,y,w3,w4, w2
7 : access to y,v,w2,w4,w5
```

5. Asynchronous computations

Another composition rule may be included in the hypothetical language, viz the parallel composition of f and g, denoted f//g. It is defined by the following rules ; u and v represent elementary actions (x:=4, or fetch(x)) which are supposedly not disrupted by other actions.

```
     u//v → u;v
  (u;f)//g → u;(f//g)
     f//g ⇆ g//f
(f//g)//h ⇆ f//(g//h)
```

This parallel composition is not synchronized, by assumption. It is then appropriate to check the absence of interferences like the one in ((x:=1)//(y:=fetch(x)). The idea is simply to verify that if a location is fetched in one branch of a parallel composition, no assignment to that location may be performed in any other branch.

Values:

-1,w1,w2,w3, ...
-identifiers, routines

Relationships : (A,F for assign, fetch)

v:=v', A(v),F(v)

Valuation rules :

```
V(x is a)=(x=Va)
V(loc)=wi, a new one
V(del(r))=del(V(r))
V(a:=b)=(Va:=Vb) and A(Va)
V(fetch(r))=v, when Vr:=v,
            and F(Vr)
V(f//g)=V(f;g), ok when, for any v, if F(v) occurs
        in V(f) then A(v) does not occur in V(g),
        and similarly for g and f.
```

The other valuation rules and the induction are again the same as before (Section 3).

Example 3. This one does not use recursions, because their treatment is the same as in the examples 2 and 3.

```
1. iden x,y,u,v ;
2. x is loc, y is loc, u is loc, v is loc ;
3. y:=v; u:=7; v:=2; x:=fetch(v) ;
4.1. (fetch(y):=10; y:=x)
4.2. //(x:=fetch(u); if b then u else x fi:=22)

1 : x,y,u,v
2 : x=w1, y=w2, u=w3, v=w4
3 : y=w2:=w4, u=w3:=1, v=w4:=1, x=w1:=1
    A(w2,w3,w4,w1),F(w4)
4.1 : v=w4:=1 , y=w2:=w1, u=w3:=1, x=w1:=1
      A(w4,w2), F(w2)
4.2 : ((x=w1:=1, u=w3:=1 ),(x=w1:=1 ,u=w3:=1))
       y=w2:=w4, v=w4:1
       A(w1,w3), F(w3)
4   : V(4.1), V(4.2) ; ok because w2 is not in
      (w1,w3) and w3 is not in (w4,w2)
```

6. Termination and correctness

The valuations given above allow to verify rather weak properties, but for all programs. The application of the method to the study of strong properties, like termination or equivalence, appears to be possible by defining a valuation for each particular program. We shall make such a study on three examples.

Example 4 (see |3|).

```
1. iden f,g,m1,m0,n1 ;
2. f is λx .
   if x=1 then 5
   else if x=0 then f(m1)
        else f(x-1)fi fi ;
3. f(m0) ;
4. g is λx . if x=1 then 1 else g(x-1)fi ;
5. g(n1) ;
```

We assume m1,n1≥1, m0≥0.

```
Values : 0,1,2, true, false
Valuation rules :
V(0)=0, V(1)=1, V(2)=V(3)= ...=2
V(a-b)=V(a)-V(b)
1-1=0, 2-1=(1,2)
Consequently, at 3 we have
f((0,1,2))=f(0),f(1),f(2)
f(0)=f(m1) i.e. f((1,2))
   =f(1),f(2)
f(1)=2
f(2)=f(2-1)=f(1,2))
   =f(1),f(2) = 2,f(2)
Hence f(0)=2, f(2)
      f(1)=2
      f(2)=2,f(2)
```

The only possibility of non-termination comes from f(2), i.e. from the non-termination of
f2= λx. if x≠1 then f2(x-1)fi
This is the same problem as the termination of g2 on line 5, where
g2= λx.if x≠1 then g2(x-1)fi.
Consequently, if g is known to terminate then f must terminate also.

Example 5 (see |6|) : a direct proof of non-termination, without the use of another auxiliary algorithm.

```
1. iden f,n; n is a number ;
2. f is  λx .
   if x<100 then f(f(x+11)) else x-11 fi ;
3. f(n);
Values: 1,2,3, true, false
V(0)= ...=V(88)=1
V(89)=...=V(99)=2
V(100)=V(101)=...=3
1+11=(1,2)             1<3=true
2+11=2.1               2<3=true
2.1-11=2             2.1<3=false
                       3<3=false
At the statement 3, we have :
f(n)=f(3),f(2),f(1)
f(3)=3-11
```

```
f(2)=f(f(2+11))=f(f(2.1))
    =f(2.1-11)=f(2)
f(1)=f(f(1+11))=f(f(1,2))
    =f(f(1)), f(f(2))
```

Consequently
f(3) does terminate ;
f(2)=f(2) does not terminate ;
f(1)=(f(f(1)), f(f(2))) does not terminate because the only possible end is through f(2) which does not terminate.

Example 6. All the previous examples of valuations show the usefulness of computing "static" information from a program. However there are situations where, on the contrary, still more "dynamic" information must be used in order to ensure correctness. The following example is such a case : the function f counts the descendant nodes from a given node, in a graph where loops are allowed. The recursive data structure is represented by functionals (see |8|).

```
1. iden omega, alpha,gr,pred1,pred2,f,a,b,c ;
2. omega is 0 ;
3. gr is λ(pred1,pred2,alpha). λcase.
        if case=1 then pred1
        else if case=2 then pred2
                       else alpha fi fi ;
4. pred1  is  λgr.gr(1) ;
5. pred2  is  λgr.gr(2) ;
6. alpha is λgr.gr(3) ;
7. f is λx.
   if x=omega then 0
   else if fetch(alpha(x)) then 0
        else alpha(x):=true ;
             1+f(pred1(x))+f(pred2(x))fi fi ;
8. a is gr(b,c,loc); b is gr(omega,c,loc);
   c is gr(c,b,loc);
9. alpha(a):=false; alpha(b):=false;
   alpha(c):=false ;
10.f(b) ;

10 : f(b)=1+f(pred1(b)+f(pred2(b));alpha(b):=true ;
     f(pred1(b))=f(omega)=0 ;
     f(pred2(b))=f(c)
        =1+f(pred1(c))+f(pred2(c)) ;
         alpha(c):= true ;
     f(pred1(c))=f(c)=0 ;
     f(pred2(c))=f(b)=0 ;
Hence
10 : f(b)=1+0+1+0+0=2 ; alpha(b):=true ;
     alpha(c):=true ;
```

The application of f terminates essentially because the function alpha establishes a very dynamic ordering in the data structure. At the beginning (step9), no node is alpha, but each node becomes an alpha as soon as it has been counted once : this ensures that f does not loop along a loop in the graph. A less dynamic situation occurs in the case of double linked lists with one head : during the counting this head permanently plays the part of alpha. The static case is of course that of numbers or non recursive data structures : then the normal well-ordering principles may be applied (see |7|).

7. Concluding remarks

Idea. We use the rule of signs in algebra, the casting out of nines in arithmetic and the parity check in hardware. Similarly, it is worth our while to compute static features of programs by computing on them with "subdata", before the actual executions on actual data. We must then define a valuation V, or subsemantics, by saying what are the values, relationships and valuation rules. Their definitions have the structure

$$V(f(t)=g(V(t)) \text{ if } P(f,t)$$

where t is a program expression, f is a basic action performed, g is the corresponding basic action in the considered valuation and P is a condition of validity. This structure is of course similar to that of the ordinary semantics.

The proposed technique may be used to verify sufficient conditions of truth or falseness of a property, even for second-order programs using routines yielding routines and locations of routines, or for still more dangerous programs which create and delete locations at will.

Advantage. The valuation processes are quite similar to the effective, normal computations. Hence the programmer could use the programming tools he knows, without learning yet another language for correctness purposes. Another consequence is that similar techniques may be used for implementing these valuations and the normal semantics : actually, in |1|, the process performing such a valuation on simple types is called "pseudo-evaluation", in order to emphasize the similarity with the run-time evaluation. Finally, such valuations may be defined for all programs of any complexity, although for rather weak properties up to now : this is the case e.g. in |2| for a kind of static type checking.

Problems. What must be studied is a general way of using the method for proving strong properties, like correctness or termination. A first approach which may be thought of, is to define valuations where relationships are generalized into predicate formulae (e.g. $a=b\times q+r$ & $r\geq 0$) and where the valuation rules produce changes in these formulae (yielding e.g. $a=b\times q+r$ & $r<b$). This is, in a sense, similar to the technique used in |5|, although not for second-order programs.

A further problem is the control of recursive data structures, for which a dynamic well-ordering must be established by the program. A general principle, based on the reasoning given in example 6, would be the following : if there is a proof that a dynamic ordering is always defined for each cycle in the data, then the algorithm working on these data and testing this ordering terminates.

Program design. The more static a program is, the more transparent it is and the more easily controlled it is. Hence, for the sake of correctness, features static at definition (identifiers, composition, fixed arrays, constants ...) are preferable to features static at application (identifiers created by recursions, recursion, dynamic arrays, parameters ...), and the latter are preferable to dynamic features (locations, program points in |8| , flexible arrays in |2|, assignations...). Of course, certain algorithms may require very dynamic structures, which are e.g. inherent to their logic : the extremely static program, without recursion, conditionals, jumps and assignations, is very easy to check but defines nothing more than a finite fixed composition of primitive functions.

It is also desirable that the valuations be defined in the programs themselves. In this way, not only existing standard checks (types, bounds, accessibility...) could be expressed in a unified way, but also new ones could be introduced at will by the user. Furthermore, the valuations themselves may then be classified according to their more or less static character : some are static at definition, e.g. models of properties of given functions, for all data ; some are static at application, e.g. for properties of given functions for given data ; others are dynamic, e.g. for dynamic recursive data structures.

Language design. If a programming language is considered as a system which defines program design rules and tools, then the language should satisfy all the requirements of program design, like those stated above : it should thus display the spectrum of static and dynamic computation structures, as well as the spectrum of static and dynamic checking techniques. Moreover, the language should also clearly distinguish and characterize the tools which are static at definition, static at application, or dynamic, in order that the programmer may consciously choose among them.

ALGOL 68 checks. It is interesting to compare the verifications of types, locations and asynchronous computations, as given in the Sections 3,4 and 5, with the checks defined in |2|. There the types are represented by "modes", and their valuation, or let us say the "emodulation", is static at definition. The basic valuation rules are the following ; u,v are types for elements and $\lambda u.v$ for routines :
$V(\lambda x.f)= \lambda u.v$ (this result must be explicitly defined in the program, for each routine);
$V((\lambda x.f)(a))=v$, ok if $V(\lambda x.f)= \lambda u.v$ and $V(a)=u$.
The latter rule is equivalent to the rule F in 9A3 of |4| and is expressed by the rule 8621 a in |2|. Dynamic mode checking is provided by the unions of modes, which specify possible dynamic modes, and by the conformity relations, which test the dynamic modes of data at given points of the program. The third class of type checking, viz static at application, is absent in |2|, but is the one used in Section 3 above : the types are checked at the application of functions, not at their definition. This gives stronger sufficient conditions of type correctness, but requires more computations during the valuation because the types must be constructed at each application.

The verifications concerning locations are related to the scope checking in |2| (and also to the checks on bounds of arrays and on locations containing data inside flexible arrays): these checks are dynamically performed at each assignation and verify sufficient conditions that no value may be used after its deletion. Such checks are required because of the stack-oriented and automatic deletions of data and because of the second-order programming features (routines delivering routines etc.). ALGOL 60 provides a similar automatic deletion system, but without second-order features ; the static control of identifiers is then sufficient to prevent the use of a value after its deletion. The valuation defined in Section 4 gives sufficient static conditions for the consistency of deletions ; the crucial check is made at the latest possible moment, i.e. the fetching, not at the assignations.

Finally, nothing in |2| is said specifically about the interferences between asynchronous computations. In the case of modes, a text with incorrect static modes is said to be syntactically incorrect. In the case of scopes, a program assigning a value to a location of smaller lifetime is said to have an undefined elaboration. In the case of asynchronous computations, a program yielding interferences is syntactically correct, semantically well-defined, but implicitly yields genuine ambiguities in the computations : these ambiguities are not tested. Section 5 above presents a valuation guaranteeing non-interference in non-trivial cases.

Formalization. A few remarks about the formalization of valuations definition should be made. The basic point is that the problems of formalization should not be mixed with the matters to formalize, although the mutual influence may be important. For example, in this paper, the valuation rules are expressed by recursive functions, which are also the heart of the hypothetical language : this follows the tendency of using a language as its own metalanguage. As another example, |2| describes the emodulation in a syntactical system using substitutions and superimposed context-free rules. Such a system may easily represent recursive equations, as shown by the following definitions of the exponential.

```
(O-rules :)
X : O.              H : .
X : X1.             H : HpX.
Y : X.              H : HsX.

(1-rules:)                          (recursive equations:)
    e : eXY.
HeXY1 : HpXeXY.                     e(x,y')=p(x,e(x,y))
HeXO  : HO1.                        e(x,0) =1
HpXY1 : HsXpXY.                     p(x,y')=s(x,p(x,y))
HpXO  : HO.                         p(x,0) =0
HsXY1 : HsX1Y.                      s(x,y')=s(x',y)
HsXO  : HX.                         s(x,0) =x
  X1  : X,1.                             x'=x+1
  O   :  .
```

References

1. Naur P., Checking of operand types in ALGOL compilers, BIT 5 (1965), 151-163.
2. Van Wijngaarden A. (ed.), B. J. Mailloux, J.E.L. Peck and C.H.A. Koster, Report on the algorithmic language ALGOL68, Num. Math. 14 (1969), 79-218.
3. Cooper D. C., Programs for mechanical program verification, in "Machine Intelligence 6", pp.43-59, Edinburgh Univ. Press, 1971.
4. Curry H. B., R. Feys, Combinatory logic, North-Holl., Amsterdam, 1958.
5. King J. C., A program verifier, Ph. D. Thesis, Carnegie-Mellon Univ., 1969.
6. Manna Z., Mc Carthy J., Properties of programs and partial function logic, in "Machine Intelligence5", pp. 27-38, Edin. Univ. Press, 1969.
7. Burstall R.M., Proving properties of programs by structural induction, Comp. J.12(1969), 41-49.
8. Reynolds J.C., GEDANKEN-A simple typeless language based on the principle of completeness and the reference concept, Comm. ACM 13(1970), 308-319.