FIXED POINT AND INVARIANT

Introduction

In their discussion of Axiomatic Semantics Programming Language textbooks give an approach to handling assignment and decision statements which is straightforward. Consideration of simple unnested ***whiles*** and the ***invariants*** necessary for their proof however is not straightforward. What is required of an invariant is clearer, than what it is. Here we explore the assertion that an invariant for a *while* is equivalent to the fixed point of a related recursive definition. Those tail-recursive definitions equivalent to *whiles* in a program that have fixed points can usually be represented as assignments. Where it is possible to replace a simple *while* in a Program P with an assignment statement, proof of the consequence of larger parts programs may become feasible.

Relation Of Recursion And *while*

A recursive definition consists of a set of conditioned equalities. Each equality is asserted to hold under a condition involving the relation of its parameters. The fixed point of a recursive function definition, R, is a function, $g_{fp}(X)$ (***X*** is a set of parameters) which satisfies the equalities of R. The object of including the recursive definition facility in a programming language in which a desired function, $g_{fp}(X)$, is not directly available, is to be able to realize $g_{fp}(X)$. The programmer usually has in mind this closed (non-recursive) form version of the function $g_{fp}(X)$ when writing the recursive definition. Proof that recursive definition R realizes $g_{fp}(X)$ requires showing that when each parameter appearing on the left of an equality is substituted for its appearance on the right equality holds under the condition associated with that equality. Proof of termination is also required.

In a ***tail* recursive** function definition *g(X)* the form of non-terminal lines is *g(X) = g(h(X)), and* terminal lines g*(X) = t(X*). *h(X)* and *g(X*) are each a set of functions, on the parameters *X.* So the line

*f(n) = f(n-1) + f(n-2) :n > 1 (read **":"** as "if")*

could not appear in a *tail recursive* definition.

The *tail recursive* definition form is of interest because it can be evaluated by a simple ***while*** program, namely one whose body contains assignment and conditional statements. For example, the *tail recursive*

schema shown on the left of figure 1 represents a broad class of *tail recursive* definitions. The equivalent *while* is shown on the right. The notation is explained in the figure. Notice that on each pass through the *while* the components of *X* are transformed into a set of equinumerous components, ***oX***. The variable ***F,*** holding the eventual result of the *while* computation, is altered by a function ***pX***. In the simplest case *X* is a single variable. This is virtually the same way in which an equivalent *tail recursive* function would be evaluated .

Basic Form Of Tail Recursion for the Function *g(F,X)*

(X_T = terminal value of ***X)***

$F = F_0$

$X = X_0$

$g(F, X) = g[F \$ pX, oX]$ $\quad :X \mathrel{!}= X_T$

$g(F, X) = F$ $\quad :X == X_T$

Equivalent *while*

(X_T = terminal value of ***X)***

$F = F_0$

$X = X_0$

while $(X \mathrel{!}= X_T)$

$\{ F = F \$ pX;$

$X = oX; \}$

Assumed to be run in parallel

In this and other figures unsubscripted upper case letters, ***X*** and ***F,*** represent ordered sets of variables, i.e. $\langle X_1, \ldots, X_n \rangle$ and $\langle F_1, \ldots, F_m \rangle$ respectively. The lower-case letter ***o,*** in ***oX*** represents ***n*** functions each applied to the components of ***X*** i.e. $oX = \langle o_1(X_1,..X_n), o_2(X_1,..X_n), \ldots, o_n(X_1,..X_n) \rangle$, similarly $pX = \langle p_1(X_1,..X_n), p_2(X_1,..X_n), \ldots p_n(X_1,..X_n) \rangle$. *$* represents an infix operator with vector arguments, so $F \$ pX = \langle F_1\$P_1, \ldots, F_j\$P_j, \ldots, ..F_n\$P_n \rangle$ where $P_j = p_j(X)$.

figure 1

Note that the recursive definition in figure 1 defines a parallel computation i.e. the value of the parameters on the right depend only on their values on the left of the equality. They do not depend on the order in which the arguments occur. In the *while* however, the order in which the recomputation of ***X*** and ***F*** are written can make a difference. So if ***X = oX*** precedes ***F = F $ pX*** in the *while* the result would differ, though interchanging the parameters in the tail recursive definition would not. In our notation, unless otherwise specified, the assignments in the body of the *while* are assumed to be done in parallel. An equivalent iterative computation in the *while* will transform the invariant In the same way as the parallel computation. (Given an iterative computation in the *while* one can get a parallel version easily by listing all the variables which are changed in the body of the *while* as a post condition. The precondition for each of these give the computation which led to that post condition

Tail Recursion And The *while* Invariant

An Invariant of a *while* is a statement S which is true after the last instruction of the *while*'s body and which, together with the condition which allows the *while* to continue, is also the weakest precondition before the first instruction of that body. S is certainly related to the function performed by the *while,* but

the nature of that relation is not always clear. Most textbooks give the property that the invariant must have-but do not say what it is. This is probably because generally there are many statements which can serve as the invariant. As a concrete statement that will often serve as the invariant I propose using the statement "The invariant involves the fixed point "$g_{fp}(X)$" represented by **"$K == g_{fp}(L)$"**. ***K*** being constant within the body of the *while***.** Any property of "$g_{fp}(X)$" also will serve as the invariant and there are some *tail recursive* functions which do not have a simple "$g_{fp}(X)$". Showing that this Invariant has the correct properties by finding the weakest precondition involves a similar process as proving that $g_{fp}(X)$ is the fixed point of the equivalent *tail recursive* definition

In the recursive definition case, one simply substitutes for parameters on the left, the corresponding parameter on the right side of the conditioned equations and shows that the resultant right and left sides are equal under these conditions Now consider the *while* case. Assume the body of the *while* consists of a set of parallel conditioned assignments statements and S is the invariant after the body of the while. Then one substitutes in S with the right side of these assignments for the occurences of the left sides of the assignments found in S. These two forms of proof are virtually identical.

In the class of *whiles* in figure 2 sequential assignment is equivalent to parallel assignment. Whether parallel or sequential in the *while* the assignments, must be equivalent to the parallel substitution for parameters in the recursive definition case. So the results should be the same.

In figure 2 the tail recursive function is shown again with its equivalent *while.* This time with the invariant included together with its weakest pre-conditions.

Basic Form Of Tail Recursion for the Function g(F,X)	**Equivalent *while* with Invariant and Proof Scheme**
(X_T** = terminal value of **X)	
F =F_0	***F = F_0 ;***
X = X_0	***X = X_0***
a) ***g(F, X) = g[F$ pX, oX] :X != X_T***	***while(X != X_T)***
b) g(F, X) = F :X== X_T	***{*** { *K ==g_{fp}(F $ pX, oX) && X != X_T* }
g(F, X) = $g_{fp}(F,X)$) the fixed point	***F = F $ pX;***
if	{ *K ==* g_{fp}*(F, oX) && oX != X_T || oX == X_T* }
a) $g_{fp}(F,X)] == g_{fp}(F \$pX, oX)$ *:X != X_T*	***X = oX;***
b) $g_{fp}(F,X)] ==$ *F :X== X_T*	{ *K ==* g_{fp}*(F, X) && X != X_T || X == X_T* }
	}
	{ *K == F && X == X_T* }

figure 2

In figures 3a there is an example of the proof scheme of figure 2 a. The tail recursive definition defines the minimum Index *J* at which *Z* is found in array *X[1...N]* when for at least one *J* between *1* and *N, X[J] == Z.*

EXAMPLE

Tail Recursive definition for the minimum Index *J* at which *X[J] = Z* in array *X[1...N]*- given that for at least one *J* between *1* and *N, X[J] == Z.*
Fixed Point and Proof

a) g(F, <X,N,Z>) = g(F+1, <X,N,Z>) :X[F] != Z
b) g(F, <X,N,Z>) = F :X[F] == Z

g_{fp}***(F, <X,N,Z>) =*** *min J >= F* **3** *X[J] == Z*

if

a) g_{fp}*((F, <X,N,Z>) = min J >= F + 1* **3** *X[J] = Z* ***:X[F] != Z***
= min J >= F **3** *X[J] = Z*

b) g_{fp}*((F, <X,N,Z>) = min J >= F* **3** *X[J] = Z* ***:X[F] == Z***
= Z

Equivalent *while* with Invariant and Proof

Can be replaced by
F = min (1 >= F) **3** *X[J] == Z && X[F]== Z }*

K==min (1 >= F) X[J] == Z && X[F]== Z }
F=1; X=an array
while(X[F] != Z)
{
{ K==min J >= F+1) X[J]==Z && X[F] != Z }
*{ K==min J >= F** **3** *X[J] == Z }*
F = F + 1; **3**
{ K==min J >= F X[J] == Z }
} **3**
{ K==min (J >= F) X[J] == Z && X[F]== Z}
or **3**
{K==F}

*since *X[F] != Z*

figure 3a

In figures 3b another example of use of the proof scheme given in figure 2 is shown. There **a** tail recursive definition for the minimum Index *J* at which the value *Z* is found in array *.X[1...N]* given that for at least one *J* between *1* and *N, X[J] == Z.* The *while* realization, invariant and proof is included

EXAMPLE

Tail Recursion Definition for Integer Division of R by D, its Fixed Point and Proof

F= F_0 ***; R=*** R_0 ***D > 0***

a) g(F, <R, D>) = g(F+1, <R-D, D>) :R >= D > 0
b) g(F,<R, D>) = F :D > R >= 0, D > 0

g(F,<R, D>) = *F + [R /D]**

if

a) *F + [R /D] = F +1+ [R -D/D]* ***:R >= D > 0***
= F +1+ [R /D] -1
= F + [R /D] yes

b) *F + [R /D]* ***= F*** ***: R + D>=D>0, R>-0***
F + 0 ***=*** *F yes*

*[R /D] represents integer division of R by D.

Equivalent *while* with Invariant and Proof

Can be replaced by
F = F_0 *+ [*R_0 */*D_0*]*

{ K == F_0 *+ [*R_0 */*D_0*] }***
D= D_0***; F=*** F_0 ***; R=*** R_0***;***
while(R >= D)
{
{ K == F + 1 + [R /D] - 1,R >= D > 0, R >= D}
{ K == F + [R /D] }
F = F + 1
{ K == F + [(R -D)/D] R >= D > 0, R-D >= 0 }
{ K == F + [R /D] - 1 R >= D > 0, R >= D }
R = R-D
{ K == F + [R /D] R+D >= D > 0, R >= 0 }
}
{ K == F + [R /D] D > R > 0 }
or
{ K == F } }

Note that if, on the right, the instruction preceding *{ K ==* F_0 *+ [*R_0 */*D_0*] }* was *F_0 = 0 then***

{ K == [R_0 */*D_0*] }* must be true before that instruction. It follows that *{ K == F }* is true after the given code iff *K == [*R_0 */*D_0*] , i.e. the while can be replaced by the assignment F ==* F_0 *+ [*R_0 */*D_0*]*

figure 3b

General Form Of Tail Recursion

Figure 4 gives another general form of the *tail recursive* definition with the conditions which determine the computations at each pass through the *while* given explicitly. It has the property that all computation onthe right of a recursive line with ***g(X)*** on the left is done within the argument structure of ***g***. For each of the mutually exclusive conditions: $cond_1$,..., $cond_m$, the function that transforms the argument structure differ. Similarly for each of the mutually exclusive terminal condition, $condT_1$,..., $condT_m$, there is a terminal value $s_1X,...,s_m X$ Every one is tested, only one succeeds.

These conditions appear in the equivalent *while* function. They are, of course, still mutually exclusive there. This is likely to be an inefficient way of actually implementing the selection of cases. In an iterative representation there are many more efficient alternatives: a jump to the end of all decisions could be executed once a true outcome occurs, or a decision could be made initially which bifurcates the possible true decisions so one need not test every conditions or etc. Despite all these alternative possible iterative implementations: given a post condition at the end of the body of the ***while,*** the resultant precondition before the body of that ***while*** is independent of the choice of equivalent ways of determining which mutually exclusive conditions is true. So the invariance does not depend on these details of the iterative implementation. Figure 4 is then shows the same parallel representation of the ***while*** as in the recursive definition.

Basic Form Of Tail Recursion for the Function g(X)

$X_0 = < x_{01},..,x_{0n}>$

1) $g(X)=g(h_{11}(X),...,h_{1n}(X))$ $:X != X_T$ && $cond_1(X)$

⋮

n) $g(X)=g(h_{n1}(X),...,h_{nn}(X))$ $:X != X_T$ && $cond_n(X)$

Tm) $g(X)=s_1 X$ $:X==X_T$ && $condT_1(X)$

⋮

m_T) $g(X)=s_m X$ $:X==X_T$ &&$condT_m(X)$

$cond_1$,..., $cond_n$ are mutually exclusive

,$condT_1$,...,$condT_m$ are mutually exclusive

Solution:

$g(X) = g_{fp}(X)$ is the fixed point

if

1) $g_{fp}(X)=g_{fp}(h_{11}(X),...,h_{1n}(X))$:X != X_T && $cond_1(X)$

⋮

n) $g_{fp}(X)=g_{fp}(h_{n1}(X),...,h_{nn}(X))$:X != X_T && $cond_n(X)$

T1) $g_{fp}(X)=s_1 X$:X==X_T && $condT_1(X)$

⋮

Tm) $g_{fp}(X)=s_m X$:X== X_T && $condT_m(X)$

Equivalent *while* with Invariant with Proof Scheme

$X = X_0 = <x_{01},..,x_{0n}>$

while (X != $condT_k(X)$; k=1 to m)

{

{ K==$g_{fp}(h_{11}(X),...,h_{1n}(X)$ && X != X_T && $cond_1(X)$ }

⋮

{ K==$g_{fp}(h_{11}(X),...,h_{1n}(X)$ && X != X_T && $cond_n(X)$ }

if ($cond_1(X)$) {X = <$h_{11}(X),...,h_{1n}(X)$>;} |*

⋮

if ($cond_T(X)$) {X = < $h_{n1}(X),...h_{nn}(X)$>};

{ K == $g_{fp}(X)$ && X == X_T | X != X_T }

}

{ K == $g_{fp}(X)$ && X ==X_T }

⋮

{X= $s_1 X$}; if {$condT_1(X)$)

{X= $s_m X$}; if {$condT_m(X)$)

|* *Must be parallel because assignment on a true statement may make following statements also true. Can be made effectively parallel with use of "elses or with case statements.*

figure 4

Figure 5 is an example of a definition which involves two recursive equations which are invoked under mutually exclusive conditions and in the corresponding while two if statements under mutually exclusive conditions. These could alternatively be one if than else statement as shown. Also the conversion from that serial equivalent back to to the original parallel form is shown

EXAMPLE

Greatest Common Divisor

1) g(X,Y) = g(X-Y, Y) :X > 0 &&Y > 0 X > Y
2) g(X,b) = g((X, Y-X) :X > 0 &&Y > 0, X <= Y
T1) g(X,b) = X :X ==Y

$g_{fp}(X,Y)$ = gcd(X, Y)
if
1) gcd(X,Y) = gcd(X-Y, Y) :X > Y
2) gcd(X,Y) = gcd(X, Y-X) :X < Y
T1) gcd(X,Y) = X :X == Y

We show that

1) gcd(X,Y) = gcd(X-Y, Y) :X > Y

Assume X > b and g = gcd(X,Y) then g divides both X and Y (X/g and Y/g are integers). X > Y so X/g - Y/g = integer h
Therefore (X-Y)/g = h,
i.e., X-Y is divided by gcd(X,Y)=g A similar argument works for 2) and
T1) is obvious

```
while( X != Y )
  {
{ (X > Y)  => K == gcd(X-Y,Y), X-Y > 0, Y > 0
  (X < Y)  => K == gcd(X,Y-X), X > 0, Y-X >= 0 }

{ (X > Y) => K == gcd(X-Y,Y), X > Y, Y > 0 }
  (X < Y) => K == gcd(X,Y-X),  X > 0, Y > X }

  if( X > Y ) {X = X-Y; }
  if( X <  Y) {Y = Y-X; }      Must Be in Parallel

{  K == gcd(X,Y), X > 0,  Y > 0 }
 }
```

```
{ ( X > Y ) =>X = X-Y && ( X =<  Y) => Y= Y - X; } Parallel Equivalent
   if( X > Y ) X = X - Y  else  Y = Y - X;        Serial Alternative
{ X, Y )
```

figure 5

Conclusions

There is a tail recursive function corresponding to each simple *while* . The fixed point of the *tail recursive* function provides the basis for an invariant for the *while.* If *gfp(X)*" is the fixed point then "*K== gfp(L)*", with K being constant within the body of the *while* is an invariant. If this invariantis carried outside the *while* to take in the initial assignments it together with the terminating condition one can formulate an assignment statement which is equivalent to the *while.* In fact finding and proving the fixed point of the *tail recursive* equivalent already does the work of finding and justifying the invariant of the equivalent *while*. To establish an equivalent assignment statement one only needs the initial conditions of the variables involved in the

in the definition and equivalent while. This still leaves open proof of properties of the *tail recursion* definition *(*and *while)* which are not the fixed points, but rather properties of the fixed points. These also are often more easily handled directly using the definition rather putting them into the form of an invariant of the *while.*

REFERENCES

1. GRIES, DAVID: *The Science Of Programming*, Springer-Verlag, 1981

2. LOUDEN, KENNETH C.: *Programming Languages Principles and Practice, 2nd edition*, Thompson-Brooks/Cole 2003, p 163

3. SCOTT, MICHAEL L.: *Pragmatics Of Programming Languages, 2nd edition,* Morgan Kauffman 2006, pp 298, 623.

4. SETHI, RAVI: *Programming Languages, Concepts and Constructs, 2nd edition*, Addison Wesley, pp 186,190.

5. TUCKER, ALLEN and NOONAN, ROBERT: *Programming Languages Principles and Paradigms, 1st edition* McGraw Hill 2002, pp 60-71.

KEYWORDS

program correctness, axiomatic semantics

