A Weakest Precondition Model for Assembly Language Programs

Wilfred J. Legato

1.0 Acknowledgments

I am indebted to Prof. Edsgar Dijkstra for introducing me to weakest preconditions, Prof. J Moore for identifying early work of John McCarthy in transforming imperative programs to their functional counterparts, and to Dr. Frank Rimlinger for many fruitful discussions on the implementation of the ideas in this paper.

2.0 Overview

This paper describes a formal model, based upon Dijkstra's weakest preconditions [4], for reasoning about assembly language programs. The model applies more generally to any finite state machine. It extends earlier work of Floyd [5], Hoare [10] and Dijkstra [4], by automatically generating closed form expressions for the weakest precondition of arbitrary loops.

Section 3 provides an informal introduction to the subject matter, followed by a more rigorous treatment in sections 4 and 5. Finally, section 6 briefly compares this to other work.

3.0 Preliminaries

3.1 Functional Programs

Reasoning about computer programs requires that program variables have well defined meaning. An imperative program may associate different values with a variable **X** depending on the state of its execution. This is not true of purely functional programs, which model computation using a composition of state-to-state transformations. By way of illustration, consider a program which sums the first N natural numbers.

```
        X = 0
LOOP    X = X + N
        N = N - 1
        IF N>0 GOTO LOOP
```

A functional representation of the computation on **X** occurring within the loop body is

SUM(N,X) = IF N>0 SUM(N-1,X+N) ELSE X

Notice that the arguments in the recursive call of **SUM** are computed by applying the composition of all state to state transformations occurring within the body of **LOOP** to the formal parameters of **SUM**. Since every occurrence of **N** and **X** within the formula have the same meaning, we may treat the equality symbol "=" as a true logical equality and use it as a basis for both computation and reasoning. Properties about the functional program, such as **SUM(N,0) = N*(N+1)/2** may be directly stated and proved using the tools of mathematical logic.

McCarthy [13] showed that any flow chart program may be mechanically transformed to a purely functional form. One could, in theory, reason about imperative programs using their functional equivalents. In fact, this is routinely done both when using interpreter[1] models for program execution and when using the techniques of this paper. Our approach, based upon Dijkstra's weakest preconditions [4], blends the functional representation for state into the functional representation for predicates on state. We then derive the state valued functions from the predicate valued functions as the need arises.

3.2 Substitutions

Our basis for capturing programming language semantics is the *substitution* operator $[e_1/x_1, e_2/x_2, \ldots, e_n/x_n]$. When applied to a term (at the purely syntactic level), this operator simultaneously replaces each occurrence of x_i with e_i, $i = 1, 2, \ldots, n$. The x_i and e_i represent, respectively, state variables (registers, memory, flags, program counter, etc.) and expressions in the state variables. When used to model state transformations, e_i expresses the new value of x_i in terms of the old state. Applying a substitution $\sigma = [e_1/x_1, e_2/x_2, \ldots, e_n/x_n]$ to a function of state $f(x_1,x_2, \ldots ,x_n)$, transforms it to an equivalent function $\sigma\, f(x_1,x_2, \ldots ,x_n) = f(e_1,e_2, \ldots ,e_n)$ in the preceding state.

Substitutions σ and τ may be composed, so that $(\sigma\tau)f(x_1,x_2, \ldots ,x_n) = \sigma(\tau f(x_1,x_2, \ldots ,x_n))$. A substitution may also represent a state, in which case e_i describes the value of state component x_i. Composing several substitutions, each representing a state transformation, describes values of state components after the final transformation in terms of state components before the first (leftmost) transformation.

For example, applying the substitution [**(N-1)/N, (X+N)/X**] to the term

IF N>0 SUM(N-1,X+N) ELSE X

yields

1 An interpreter is a function which takes a state, a program, and a sequence of external (e.g. interrupts, input data, etc.) events as arguments, and returns a new state which represents the effect of running the program for a number of steps equal to the length of the event (also called an "oracle") sequence.

IF N-1>0 SUM((N-1)-1,(X+N)+(N-1)) ELSE (X+N).

Composing [**(N-1)/N, (X+N)/X**] with [**(N-1)/N, (X+N)/X**] yields

[**((N-1)-1)/N, ((X+N)+(N-1))/X**].

3.3 Modeling Computation

There are two conceptually different approaches to modeling program execution. The "natural" method is to move forward through the program, successively expressing each new state in terms of the original state. This is commonly called *symbolic execution*. Alternatively, one may move backwards through the program, expressing the final state in terms of successively earlier states. In the absence of branch instructions, both computations yield identical results. Forward execution is commonly used with interpreter models, whereas backward execution is typical of weakest precondition models.

We illustrates the similarities between forward and backward execution by way of an example. In the following table, we describe the semantics of each instruction using substitutions. The state component **PC** represents an instruction counter.

	Instruction	**Substitution**
	X = 0	[**(1+PC)/PC, 0/X**]
LOOP	**X = X + N**	[**(1+PC)/PC, (X+N)/X**]
	N = N - 1	[**(1+PC)/PC, (N-1)/N**]
	IF N>0 GOTO LOOP	[**(IF N>0 2 ELSE (1+PC))/PC**]

The following table shows the sequence of states obtained by symbolic execution beginning with the state [**0/PC,X/X,N/N**]. Each entry in column 2 is the state following execution of the corresponding instruction in column 1.

	Computation	**State**
	X = 0	[**1/PC, 0/X, N/N**]
LOOP	**X = X + N**	[**2/PC, (0+N)/X, N/N**]
	N = N - 1	[**3/PC, (0+N)/X, (N-1)/N**]
	IF N>0 GOTO LOOP	[**(IF N-1>0 2 ELSE 5)/PC, (0+N)/X, (N-1)/N**]

States are computed by applying the previous state to the substitution associated with the current instruction, e.g. [**1/PC, 0/X, N/N**] = [**0/PC, X/X, N/N**] [**(1+PC)/PC, 0/X**]. The intuition guiding these computations is that the state transformation associated with an instruction execution expresses the new state ("denominator") in terms of the old ("numerator"). The previous state expresses the old state in terms of the initial state. Composing the two, gives the current state in terms of the initial state.

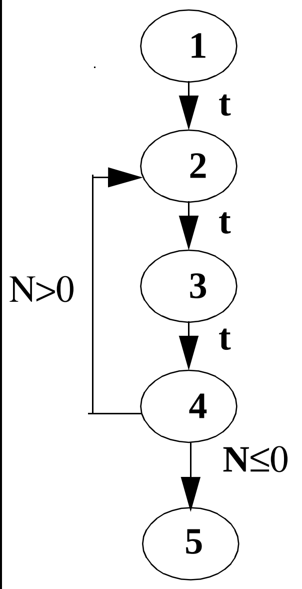
We do not continue the execution beyond instruction 4, since the state at each instruction no longer uniquely depends on the initial state. For example, the state at instruction 3 depends on how many iterations of the loop have occurred, whereas the state at location 5 is uniquely defined as a recursive function [2]of the initial state. We shall see that the backward execution provides a much more natural context for dealing with loops.

Consider now backward execution of this same program, beginning with the state at location 5. We omit **PC** from the state, since each state will be uniquely associated with an instruction. When performing a backward execution step over a branch instruction, we need the states at each of the branch targets. We compute these in a demand driven fashion. So, the computation of the state at location 4 demands that we know the state at location 2, which in turn demands that we know the state at location 3, which demands that we know the state at location 4. We are now back where we started, so we invent a function **S(X,N)** which represents the state at location 4. We use **S(X,N)** to complete the calculation, deriving the following equation for **S(X,N)**.

$$\mathbf{S(X,N) = IF\ N>0\ S(X+N,N-1)\ ELSE}\ [X/X,N/N]$$

The following graph shows this computation in greater detail.

Backward Computation for the Sum Program



$$S_1(X,N) = [0/X]\ S_2(X,N) = S(0+N,N-1) = S(N,N-1)$$

$$S_2(X,N) = [(X+N)/X]\ S_3(X,N) = S(X+N,N-1)$$

$$S_3(X,N) = [(N-1)/N]\ S(X,N) = S(X,N-1)$$

$$S(X,N) = \text{IF } N>0\ S_2(X,N) \text{ ELSE } [X/X,N/N]$$

$$[X/X,N/N]$$

2 For example, [**5/PC, SUM(N,X)/X, 0/N**].

Each node is labelled by its corresponding instruction. The arcs are labelled with predicates on the state which specify the conditions under which one instruction transits to another. $\mathbf{S_1}$, $\mathbf{S_2}$ and $\mathbf{S_3}$ define the state at location 5 in terms of the state at locations 1, 2 and 3 respectively.

Since $\mathbf{S_1(X,N)}$ returns a substitution, it may in fact be applied to an arbitrary function of state at location 5, transforming it into an equivalent function of the state at location 1.

3.4 Weakest Preconditions

Much of what we have done with backward execution, has been done previously in the context of Floyd-Hoare program semantics and Dijkstra's weakest preconditions. We review here briefly the foundations laid by John McCarthy[13], Robert Floyd [5], Tony Hoare[10] and Edsgar Dijkstra [4]. We begin with the notion of a correctness specification {P} C {Q}, where P (the *precondition*) and Q (the *postcondition*) are predicates about the state of a program, and C is a code fragment from the program. The specification is true, provided that whenever the program C begins execution in a state satisfying P, then if C terminates the resulting state will satisfy Q. This is called a *partial correctness* specification because it does not require that the program terminate. The notation [P] C [Q] is used to describe a *total correctness* specification which in addition to the above requires that the program C terminate. If P is the weakest predicate (meaning that it restricts the state the least) such that C terminates in a state satisfying Q, then P is called the *weakest precondition*. In the case of partial correctness specifications, P is called the *weakest liberal precondition*. When we use the term *weakest precondition*, we will in fact mean *weakest liberal precondition*.

Given an instruction with semantics (substitution) σ and postcondition Q, the *weakest precondition* is σQ. The intuition here states that if Q is a function of the state following the transformation σ, then replacing each state variable in Q by its representation in terms of the previous state, we derive an equivalent predicate in the preceding state. Dijkstra labelled such substitutions *predicate transformers*. Predicate transformers may be chained by repeatedly applying the substitution for an instruction to the weakest precondition of its successor.

A correctness specification for the sum program is

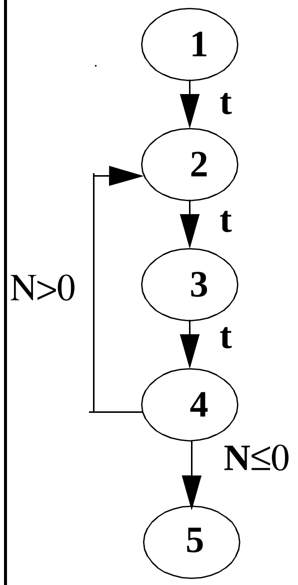
<u>Specification</u>

```
        {NS = N ∧ N>0}
        X = 0
LOOP    X = X + N
        N = N - 1
        IF N>0 GOTO LOOP
        {X = NS×(NS+1)/2}
```

where **NS** is a “ghost” variable (not part of the program state) used to reference the initial value of **N** within the postcondition $\mathbf{Q(X,NS)} \equiv (\mathbf{X = NS\times(NS+1)/2})$.

The following graph demonstrates the computation of weakest preconditions for the sum program.

Weakest Preconditions for the Sum Program



$$S_1(X,N,NS)=[0/X]S_2(X,N,NS)=S(0+N,N-1,NS)$$

$$S_2(X,N,NS)=[(X+N)/X]S_3(X,N,NS)=S(X+N,N-1,NS)$$

$$S_3(X,N,NS)=[(N-1)/N]S(X,N,NS)=S(X,N-1,NS)$$

$$S(X,N,NS)=\text{IF } N>0\ S_2(X,N,NS)\text{ ELSE } Q(X,NS)$$

$$Q(X,NS)$$

S(X,N,NS) represents the weakest precondition at location 4. $\mathbf{S_1}$, $\mathbf{S_2}$ and $\mathbf{S_3}$ are the weakest preconditions at instructions 1, 2 and 3 respectively. The correctness theorem is

$$(NS = N \wedge N>0) \Rightarrow S_1(X,N,NS)$$

which simplifies to

$$N>0 \Rightarrow S(N,N-1,N)$$

The important thing to bear in mind, is that the correctness statement for the program is totally embodied in a single predicate referring to its beginning state.

In summary, we compute the weakest precondition using the following algorithm. Given an assignment statement x = e with postcondition Q, the weakest precondition is simply [e/x]Q. For a branch instruction (which except for **PC** does not change the state) with branch conditions b and $\neg$b and associated postconditions Q_1 and Q_2, the weakest precondition is $(b \wedge Q_1) \vee (\neg b \wedge Q_2)$. Notice that this is logically equivalent to

IF b Q_1 ELSE Q_2. The invention of recursive functions defining weakest preconditions is treated in full generality later.

4.0 Transforming Programs to Directed Graphs

Given an assembly language program, we generate a forward control flow graph as follows. Associate with each instruction in the program a unique node. If the instruction n transits to the instruction n' upon the branch condition $B_{n,n'}$, then connect n to n' with a directed arc labelled with the predicate $B_{n,n'}$. Notice that sequential non-branch instructions are connected with an arc labelled **t** (true). Associated with each node n is a substitution σ_n which is the predicate transformer for the instruction at node n. Our execution model assumes that each node performs any state transformation actions prior to testing the branch conditions exiting the node. Thus the branch predicates share state with the target of the branch.

We illustrate the construction of control flow graphs using a simple program which multiplies two 8-bit numbers on the early Mostek 6502 microprocessor. The 6502 has a single 8-bit accumulator **A** and two 8-bit index registers **X** and **Y**. It has a carry flag **C** and zero flag **Z**, which remembers whether the last arithmetic calculation produced a zero result. In the following code **F1**, **F2**, and **LOW** represent single byte memory locations. Initially **F1** and **F2** hold the two numbers to be multiplied. At the end of the computation **LOW** will hold the low order 8-bits of the product, and **A** will hold the high order 8-bits. **LOOP** and **ZCOEF** are arbitrary labels used as targets of jump instructions. **F1SAVE** is a ghost variable which preserves the initial value of **F1**.

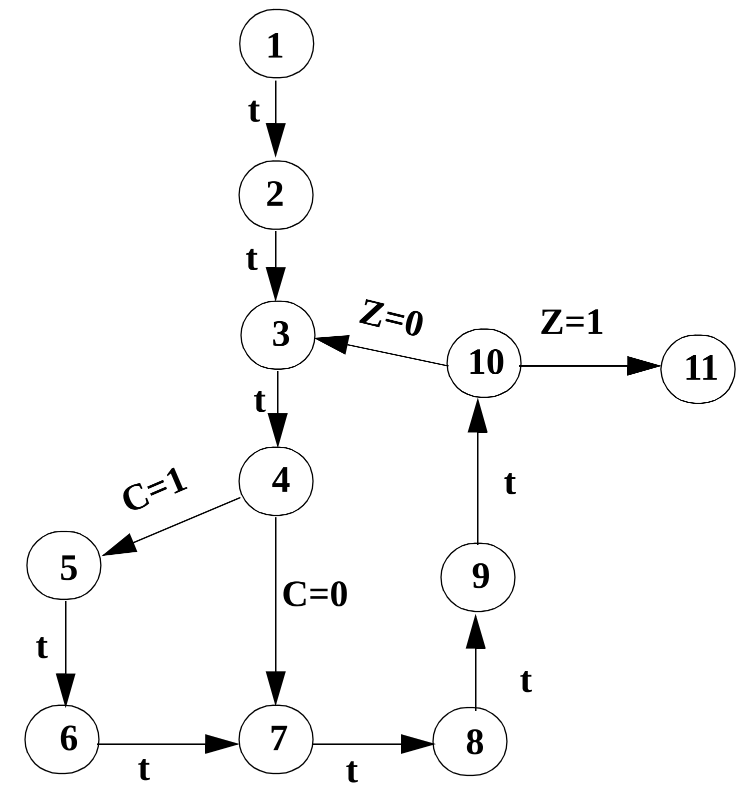
A Program to Multiply Two 8-bit Numbers

{ F1 = F1SAVE }

```
        LDX #8        ; load X immediate with the integer 8
        LDA #0        ; load A immediate with the integer 0
LOOP    ROR F1        ; rotate F1 right circular through C
        BCC ZCOEF     ; branch to ZCOEF if C = 0
        CLC           ; set C to 0
        ADC F2        ; set A to A+F2+C and C to the carry
ZCOEF   ROR A         ; rotate A right circular through C
        ROR LOW       ; rotate LOW right circular through C
        DEX           ; set X to X-1
        BNE LOOP      ; branch to LOOP if Z = 0
```

{ F1SAVE $\times$ F2 = 256 $\times$ A + LOW }

Directed Graph for the Multiply Program



5.0 Calculation of Weakest Preconditions

5.1 Overview

Rather than have a single postcondition for a block of code, we begin with a collection of predicates Q_n, each of which must be satisfied at their respective nodes n. Our goal is to find the weakest predicates P_n such that any state satisfying P_n at node n will, when at node n', satisfy $Q_{n'}$ for any node n' reachable from n (including n itself). We show how to construct such P_n by first deriving a set of equations that the P_n must satisfy, and then solving these equations for concrete representations of the P_n.

5.2 Deriving Equations

Let S(n) be the set of all successor nodes for n. If $S(n) \neq \emptyset$ we say n is an *internal* node, otherwise it is *terminal*. We say that an execution is *legal* beginning at node n provided: (1) for all nodes n' on the execution path *following* node n, the state immediately prior to execution of the instruction at node n' satisfies $P_{n'}$, and (2) the execution does not halt at an internal node. Notice that the definition of legal is relative to the set of predicates P_n. We now constrain P_n to be the weakest predicate implying Q_n which allows only legal executions beginning at node n. The following theorem shows that this constraint creates a (generally co-recursive) system of defining equations for the P_n.

Theorem: If for all n, P_n is the weakest predicate implying Q_n which allows only legal executions beginning at node n, then[3]

$$P_n = Q_n \wedge \sigma_n\left(\bigvee_{n' \varepsilon S(n)} (P_{n'} \wedge B_{n,n'})\right) \qquad \text{if n is internal} \qquad (1)$$

$$P_n = Q_n \qquad \text{if n is terminal}$$

Proof: We show first that $P_n \Rightarrow Q_n \wedge \sigma_n(\bigvee_{n' \varepsilon S(n)} (P_{n'} \wedge B_{n,n'}))$ for internal nodes n, and $P_n \Rightarrow Q_n$ for terminal nodes n. Since P_n is the weakest predicate implying Q_n, we obviously have $P_n \Rightarrow Q_n$ for all n. Suppose n is an internal node, and s is the beginning state for a legal execution starting at node n. Since $P_n \Rightarrow Q_n$, we need only show that s satisfies $\sigma_n(\bigvee_{n' \varepsilon S(n)} (P_{n'} \wedge B_{n,n'}))$. From property (2) of a legal execution, s has a successor state s', which must satisfy $P_{n'} \wedge B_{n,n'}$ for some node n'. Since $\sigma_n(P_{n'} \wedge B_{n,n'})$ is the weakest precondition for the postcondition $P_{n'} \wedge B_{n,n'}$ it must include the state s. Since substitutions on terms distribute over logical connectives, s indeed satisfies

3. The symbol $\bigvee$ represents the logical disjunction ("or") over the range described by its subscript.

$Q_n \wedge \sigma_n(\bigvee_{n' \varepsilon S(n)} (P_{n'} \wedge B_{n,n'}))$.

We show now that $Q_n \wedge \sigma_n(\bigvee_{n' \varepsilon S(n)} (P_{n'} \wedge B_{n,n'})) \Rightarrow P_n$ for internal nodes n, and $Q_n \Rightarrow P_n$ for terminal nodes n. If n is terminal, all executions beginning at node n are legal and therefore P_n is simply the weakest predicate implying Q_n, which is Q_n. Consider now the case where n is internal, and let s satisfy $Q_n \wedge \sigma_n(\bigvee_{n' \varepsilon S(n)} (P_{n'} \wedge B_{n,n'}))$. Distributing σ_n over $\bigvee$, s must satisfy $\sigma_n(P_{n'} \wedge B_{n,n'})$ for some n'. Furthermore $P_{n'} \wedge B_{n,n'} \neq$ false, otherwise s could not satisfy $\sigma_n(P_{n'} \wedge B_{n,n'})$. So there exists a state s' satisfying $P_{n'} \wedge B_{n,n'}$. Since $P_{n'}$ allows only legal executions, there is a legal execution beginning in state s' at node n' which satisfies $P_{n'}$. Prepending s to this execution sequence, we have a legal execution beginning in state s at node n. Since s satisfies Q_n, and P_n is the weakest predicate implying Q_n and allowing only legal executions beginning at node n, s must satisfy P_n. This completes the proof.

5.3 Solving the Equations

We describe an algorithm which uses the equations (1) to derive closed form expressions for the P_n in terms of the set of predicates Q_n. These expressions will in general involve co-recursive routines, which may be algorithmically converted (see below) to simply recursive routines. If termination properties can be proved for all simply recursive routines, then the existence of a unique solution to the system of equations (1) is assured. Our goal is to incrementally compute the weakest precondition WP(n) at node n, by accumulating (using logical "or") $Q_n \wedge \sigma_n(WP(n') \wedge B_{n,n'})$ for all successor nodes n'. We arrange our computation to process first those nodes with the fewest active successors. This avoids unnecessary introduction of new function symbols. In addition to WP(n), the algorithm makes use of counters M(n), which give the number of currently active successors of the node n, a list of yet to be processed nodes L, and the set A(n') of predecessors of the node n'. The algorithm proceeds as follows.

Initialize the variables.

M(n) = |S(n)|

WP(n) = Q_n if n is a terminal node, false otherwise

L = the set of all nodes

A(n') = { n : n' ε S(n) }

Iterate the following while L ≠ ∅.

(a) Choose an n' in L with minimal M(n').
(b) If M(n') = 0, then for each n ε A(n') accumulate (using disjunction)

$Q_n \wedge \sigma_n(WP(n') \wedge B_{n,n'})$ into WP(n) and then decrement M(n).

(c) If $M(n') > 0$, manufacture a new function symbol $f_{n'}$ whose arguments consist of all state variables $a_1, a_2, \ldots, a_k$,[4] then for each $n \in A(n')$ accumulate $Q_n \wedge \sigma_n(f_{n'}(a_1, a_2, \ldots, a_k) \wedge B_{n,n'})$ into WP(n), and finally decrement M(n).

(d) Delete n' from L.

Upon termination $WP(n) = P_n$. For each manufactured f_n set up the equation

$$f_n(a_1, a_2, \ldots, a_k) = WP(n)$$

These equations may be simplified by removing from the formal argument list of f_n any a_i that occurs within WP(n) solely in the i^{th} argument position of calls to f_n. The resulting equations define, in general, a co-recursive set of function definitions which explicitly represent the P_n in terms of the Q_n.

The co-recursive functions may be converted into simply recursive functions by collecting all co-recursive calls which are mutually dependent, and creating a simple recursion whose body performs a case split on a new parameter which identifies which of the co-recursive functions is being called. Once in this form, termination arguments may be rigorously pursued using, for example, the Boyer-Moore theorem prover NQTHM[1,2] or the Kaufmann-Moore theorem prover ACL2[11].

It is worth observing that the functions generated with the above algorithm are tail recursive. This follows from the fact that all occurrences of f_n within the WP(i) are at the outermost (logical) level. There are no pending operations that need to be performed on results returned from calls to f_n within any of its co-recursive routines. Panagiotis Manolios and J Moore have shown [12] that such functions may be safely introduced into NQTHM or ACL2 without affecting the soundness of the Boyer-Moore logic.

5.4 Weakest Preconditions for the Multiply Program

We show the output (after simplification using arithmetic and logic rules) of the above algorithm when applied to the multiply program. The syntax is that of NQTHM, which is similar to the lisp programming language. $(f\ x_1\ x_2 \ldots\ x_n)$ represents the function f applied to the arguments $x_1, x_2, \ldots, x_n$. “defn” plays the role of the common lisp “defun,” which defines a function. “defvar” defines lisp variables corresponding to weakest preconditions at various lines of the program. “equal,” “lessp,” and “zerop” are predicates which return true if their arguments satisfy the relations that their names suggest. “sub1,” “difference,” “plus,” “times,” and “remainder” also perform the operations that their names suggest on the natural numbers. “quotient” returns the integer part of a quotient. All functions are total, returning 0 where they would otherwise be undefined or not a natural number.

4. One may avoid listing all state variables by selecting only those occurring in the domain of substitutions applied to $f_{n'}(a_1, a_2, \ldots, a_k)$ in the course of computing the WP(n).

Notice that the postcondition is represented by the weakest precondition of a fictitious 11th instruction. In order to prove termination, the definition of “dec” was added as well as the final line of the definition of wp-zcoef, which instructs NQTHM to use (lessp (dec x)) as a decreasing measure function. The correctness statement for this program is

$$((f1save = f1) \wedge (f1 < 256) \wedge (f2 < 256) \wedge (low < 256)) \Rightarrow wp\text{-}1.$$

Output of the Equation Solving Algorithm

```
(defn dec (x)
  (if (zerop x)
      255
    (sub1 x)))

(defvar wp-1
  ‘(wp-zcoef (plus (times c 128) (quotient f1 2))
                8
                0
                low
                (times (remainder f1 2) f2)
                f1save
                f2)

(defvar wp-loop
 ‘(wp-zcoef (plus (times c 128) (quotient f1 2))
               x
               (times (remainder f1 2) (quotient (plus a f2) 256))
               low
               (plus (times (difference 1 (remainder f1 2)) a)
                     (times (remainder f1 2) (remainder (plus a f2) 256)))
               f1save
               f2))

(defvar wp-5
 ‘(wp-zcoef f1
              x
              (quotient (plus a f2) 256)
              low
              (remainder (plus a f2) 256)
              f1save
              f2))

(defn wp-zcoef (f1 x c low a f1save f2)
  (if (equal (dec x) 0)
      (equal (plus (times (plus (times c 128) (quotient a 2)) 256)
                    (plus (times (remainder a 2) 128)
                          (quotient low 2)))
             (times f1save f2))
     (wp-zcoef
       (plus (times (remainder low 2) 128) (quotient f1 2))
       (dec x)
       (times (remainder f1 2)
              (quotient (plus (plus (times c 128) (quotient a 2)) f2)
                        256))
       (plus (times (remainder a 2) 128) (quotient low 2))
       (plus (times (difference 1 (remainder f1 2))
                    (plus (times c 128) (quotient a 2)))
             (times (remainder f1 2)
                    (remainder (plus (plus (times c 128) (quotient a 2)) f2)
                               256)))
       f1save
       f2))
  ((lessp (dec x))))

(defvar wp-11
 ‘(equal (plus (times a 256) low)
         (times f1save f2)))
```

6.0 Concluding Remarks

When proving correctness theorems using weakest preconditions, it is often desirable to have available recursive definitions of state components. These may be derived mechanically from the weakest preconditions by simply replacing the exit predicates with the desired state variable. For example, to derive a recursive function for state component **X** from the weakest precondition at location 4 of the sum loop program

S(X,N,NS) = IF N>0 S(X+N,N-1,NS) ELSE Q(X,NS)

we define the function

X-S(X,N) = IF N>0 X-S(X+N,N-1) ELSE X.

It is a trivial verification effort (using induction) to prove[5]

S(X,N,NS) = Q(X-S(X,N),NS)

5 Notice that the term-valued recursive functions are in fact components of predicate transformers. For an example of a complete predicate transformer see **S_1(X,N)** at the end of section 3.3.

Direct representation of state components by recursive functions makes possible the creation of lemmas which rewrite expressions into normal form. This in turn contributes to higher degrees of proof automation.

We now compare the use of interpreters to weakest preconditions. An interpreter models all aspects of program execution directly within a formal system. It makes use of the same information as do weakest preconditions (σ_n and $B_{n,n'}$) to form a model for program execution. The model, when provided with a "clock" function steps forward from an initial state to a final state. The postcondition is then applied to the final state, which through the interpreter, becomes a function of the initial state. If the interpreted program contains loops, it is generally necessary to manufacture recursive functions similar to **X-S** above and prove that the state transformations effected by the loops is the same as that given by the recursive functions. This activity is repeated (if for example there are loops within loops) until one arrives roughly at the same point at which weakest preconditions begin.

The advantage of using interpreters is increased rigor.[6] Program execution semantics are captured entirely within a formal logic, whereas weakest preconditions embody the execution semantics in their calculation.

When implementing the algorithm for computation of weakest preconditions described in this paper, careful consideration must be given to program efficiency. Without due care, the size of the expressions P_n will grow exponentially with the length of the target program, and even with linear growth the size of the predicates generated on moderate size programs will exceed the capabilities of popular theorem provers which use "bottom up"[7] rewriting. Effective use of the techniques described in this paper may require redesign of currently available theorem provers.

Finally, it is interesting to note that in the context of finite state automata, this paper provides a concrete example of Church's Thesis, which states that any computation possible with a Turing Machine is also possible using the Lambda Calculus.

6 Some would claim that interpreters have the additional advantage of being executable, but the same holds true of weakest precondition models provided the system clock is part of the state. Each of the Q_n would simply state that the clock has the desired value. Each of the state components may then be derived by executing the term-valued functions referenced above.

7. "Bottom up" rewriting transforms the innermost terms of an expression prior to rewriting the encompassing expression. It performs a depth first exploration of the expression tree.

Bibliography

1. Robert S. Boyer and J Strother Moore, “A Computational Logic”, Academic Press, ACM Monograph Series, 1979. ISBN 0-12-122950-5.

2. Robert S. Boyer and J Strother Moore, “A Computational Logic Handbook”, Academic Press, Perspectives in Computing, volume 23, 1988. ISBN 0-12-122952-1

3. D. L. Clutterbuck and B. A. Carré, “The Verification of Low-level Code,” Software Engineering Journal, May 1988.

4. E. W. Dijkstra, “A Discipline of Programming,” Prentice-Hall Series on Automatic Computation, 1976.

5. R. W. Floyd, “Assigning Meaning to Programs,” Proceedings of the American Mathematics Society (Symposia on Applied Mathematics), volume 19, 1967, pp. 19-31.

6. G. C. Gannod and B. H. C. Cheng, “Abstraction of Formal Specifications from Program Code,” Proceedings of the 1991 IEEE International Conference on Tools for AI, San Jose, CA, Nov. 1991.

7. G. C. Gannod and B. H. C. Cheng, “Strongest Postcondition Semantics as the Formal Basis for Reverse Engineering,” Proceedings for the Second Working Conference on Reverse Engineering, Toronto, Ontario, pp. 188-197, July 14-16, 1995.

8. Allen Goldberg and Tie-Cheng Wang, “Integration of Linear Arithmetic and Goal-Oriented Resolution,” Frontiers in Combining Systems, (FroCos 1998), Amsterdam, Netherlands, Oct. 1998.

9. J. M. Hart, “Experience with Logical Code Analysis in Software Maintenance, Software Practice and Experience, 1995.

10. C. A. R. Hoare, “An Axiomatic Basis for Computer Programming,” Communications of the ACM, volume 12, number 10, October 1969, pp. 576-583.

11. Kaufmann, Manolios, Moore, “Computer-Aided Reasoning: An Approach,” Kluwer Academic Publishers, 2000, ISBN 0-7923-7744-3.

12. Manolios, Moore, “Partial Functions in ACL2,” (to be published) http://www.cs.utexas.edu/users/moore/publications, Feb. 2001.

13. John McCarthy, “Towards a Mathematical Science of Computation,” Proceedings of IFIP Congress 1962, North Holland Pub. Co., Amsterdam, 1963. Also available on John McCarthy’s web site at http://www-formal.stanford.edu/jmc.

14. A. Pizzarello, “A New Method for Location of Software Defects,” Peritus Software Services, Inc., 1993.

15. A. Pizzarello, “Formal Techniques for Understanding Programs,” Proceedings of the 8th International Software Quality Week, San Francisco, May 30 - June 2, 1995.