Generic Theories as Proof Strategies: A Case Study for Weakest Precondition Style Proofs

Wilfred J. Legato

1.0 Abstract

This paper presents several techniques, motivated by the study of weakest preconditions, for structuring proofs about recursive functions using generic theories. The theories can be implemented on a variety of theorem provers that support introduction and instantiation of partial functions (PVS, HOL, ACL2, NQTHM). The focus here is on the Boyer-Moore (NQTHM [1,2]) and Kaufmann-Moore (ACL2 [6]) theorem provers.

2.0 Background

The automated generation of weakest preconditions described in [7] introduces new recursive functions with no known properties other than their definitions. Interesting properties of these recursive functions must be established using induction, and induction is complicated by the fact that many of the required properties are for specific applications of the recursive functions. A standard means for overcoming this difficulty is to generalize the required properties, perform induction, and then instantiate the result. We illustrate this approach with the following simple example.

Consider a program loop that sums the integers from 1 to n. It employs an accumulator, A that is initially set to 0, and a variable X that is initially set to n. On each iteration of the loop, X is added to A and then decreased by 1 provided it is not 0. When X is 0, control exits from the loop. Our goal is to prove that A = (n * (n + 1))/2 upon exit. We ignore for now constraints placed upon A and n due to finite precision arithmetic. The weakest precondition for the program loop is

$$wp(A,X) = ((X > 0) \wedge wp(A + X, X - 1)) \vee ((X \leq 0) \wedge (A = (n * (n + 1))/2))$$

Backing this predicate up over the loop initialization instructions replaces A by 0 and X by n, yielding wp(0,n). Since n is a variable, we must use induction to prove wp(0,n). The standard induction suggested by NQTHM and ACL2 is patterned after the recursive definition of wp, and its effectiveness relies upon both A and X being variables. ACL2 and NQTHM's induction heuristic will generate the two cases:

$wp(0,0)$	base case
$n > 0 \wedge wp(0,n - 1) \Rightarrow wp(0,n)$	induction step

The base case proves easily, but the induction step presents a problem. The only property available is the definition of wp. Replacing wp(0,n) by its definition yields the proof goal

$$n > 0 \wedge wp(0,n - 1) \Rightarrow wp(n,n - 1)$$

where the hypothesis does not contribute to the proof of the conclusion. The standard way out of this predicament is to draw upon human insight, realizing that the final value of the accumulator is its original value plus (X * (X + 1))/2. This generalization leads to the theorem $wp(A,X) \equiv A + (X * (X + 1))/2 = (n * (n + 1))/2$. Now induction patterned after wp yields

$$wp(A,0) \equiv A + (0 * (0 + 1))/2 = (n * (n + 1))/2 \qquad \text{base case}$$

$$X > 0 \wedge (wp(A + X,X - 1) \equiv A + X + ((X - 1) * X)/2 = (n * (n - 1))/2)$$

$$\Rightarrow (wp(A,X) \equiv A + (X * (X + 1))/2 = (n * (n - 1))/2) \qquad \text{induction step}$$

The base case follows from expanding wp(A,0), and the induction step follows from expanding wp(A,X) using the identity (X * (X + 1))/2 = X + ((X - 1) * X)/2. Finally, we instantiate this theorem replacing A by 0 and X by n.

With simple programs such as this, the human insight comes easily. In situations where it does not we offer several alternative approaches using generic theories.

3.0 Generic Theories

NQTHM and ACL2 each have means to soundly introduce new functions that are only partially specified. Within NQTHM one uses the ***constrain*** event, and within ACL2 the ***encapsulate*** event. Using these facilities, we introduce functions that capture the relevant properties of weakest preconditions, loop invariants, postconditions, substitutions and other objects of interest. We then prove properties of these functions, that can later be instantiated[1] using ***functionally-instantiate*** (NQTHM) or ***:functional-instance*** (ACL2). We include three generic theories: (1) a theory for Floyd-Hoare style proofs using loop invariants, (2) a theory to convert tail recursive functions to primitive recursive functions, and (3) a theory for functional equivalence and alternative inductions. In the following we use the lisp syntax of NQTHM and ACL2 for function definition and application.

3.1 Loop Invariant Theory

In its most general setting the weakest precondition algorithm will generate a co-recursive set of functions to represent a collection of intertwined loops. This set may be mechanically converted to a simply recursive function of the following form[2]

1. Only those properties introduced with the defining event (e.g “constrain” or “encapsulate”) need be proven when instantiating the theory.
2. Notice that we have not sacrificed any generality here, since multiple recursive calls within a function body can be combined into a single transformation, sigma, whose components embody the conditionals governing the recursive call.

(wp **s**) = (if (b **s**)
(qp **s**)
(wp (sigma **s**))

where **s** represents state, b the loop exit predicate, qp the postcondition and sigma the state transformation for the body of the loop. This generic function is introduced into NQTHM or ACL2 via the following two axioms (using constrain or encapsulate, respectively).

(b **s**) ⇒ (wp **s**) = (qp **s**)

(not (b **s**)) ⇒ (wp (sigma **s**)) = (wp **s**)

Since these axioms are treated as rewrite rules, the orientation of the equalities is important. We restrict the loop invariant theory to total functions by requiring that sigma decrease under some measure[3].

(e0-ordinalp (measure **s**))

(not (b **s**)) ⇒ (e0-ord-< (measure (sigma **s**)) (measure **s**))

Next we introduce a loop invariant r

(not (b **s**)) ∧ (r **s**) ⇒ (r (sigma **s**))

(b **s**) ∧ (r **s**) ⇒ (qp **s**)

From which we prove by an induction patterned[4] after wp

(r **s**) ⇒ (wp **s**) (1)

Since wp satisfies all the properties of a loop invariant, this characterizes wp as the weakest loop invariant. This result is important because it enables weakest precondition proofs to be conducted in the classical Floyd-Hoare style.

A typical application of this theory occurs when (wp $\mathbf{s}_0$) is within the proof goal and (sigma $\mathbf{s}_0$) is not an instance of (sigma **s**).[5] This prevents wp from being used effectively as an induction pattern. Instead of generalizing the proof goal so that wp is a suitable induction pattern (see section 4.1 for an example of this), one can instead look for a loop invariant r such that (r $\mathbf{s}_0$) is provable in the context in which (wp $\mathbf{s}_0$) occurs. The above theorem (1) may then be functionally instantiated to establish the proof goal. In practice r

3. See reference [6] for a description of e0-ordinalp and e0-ord-<. Basically, e0-ordinalp recognizes an ordinal number, and e0-ord-< is a well founded relation on the ordinals. These are used to show termination of functions introduced with the definitional principle. The NQTHM counterparts to these functions are ordinalp and ord-lessp.
4. An inductive proof patterned after wp of a predicate P splits into the base case (b **s**) ⇒ (P **s**) and the induction step ¬(b **s**) ∧ (P (sigma **s**)) ⇒ (P **s**).
5. For example, let s be the pair (A, X), $\mathbf{s}_0$ = (0, X), (sigma $\mathbf{s}_0$) = (0, X-1) and (sigma **s**) = (A+X, X-1).

will be chosen such that it involves only previously known functions that are either built into the logic or part of one of the standard proof libraries.

3.2 Tail Recursion Theory

This theory transforms a tail recursive function into a primitive recursive function on a subset of the arguments of the original function. If this subset is chosen properly, then the primitive recursive function may be used as an induction pattern where the tail recursive pattern fails. We constrain the tail recursive function

```
(g a s) = (if (bb s)
              (qt a s)
            (g (rho a s) (tau s)))
```

using the axioms

(bb **s**) ∧ (rt **a** **s**) ⇒ (g **a** **s**) = (qt **a** **s**)

¬(bb **s**) ∧ (rt **a** **s**) ⇒ (g (rho **a** **s**) (tau **s**)) = (g **a** **s**)

¬(bb **s**) ∧ (rt **a** **s**) ⇒ (rt (rho **a** **s**) (tau **s**))

where rt is an invariant used to capture assumptions underlying the theory. For example, rt may require that **a** be a natural number and **s** be a list of natural numbers. As before, we constrain a measure function with the following axioms

(e0-ordinalp (measure-g **s**))

¬(bb **s**) ⇒ (e0-ord-< (measure-g (tau **s**)) (measure-g **s**))

We define the function

```
(a-g a s) = (if (bb s)
                a
              (a-g (rho a s) (tau s)))
```

which is known to terminate because of measure-g. a-g plays the role of a state valued counterpart to a recursive weakest precondition described in section 6 of reference [7]. It is identical to g, except that it returns **a** in the final state instead of (qt **a** **s**).

We constrain the primitive recursive function

```
(h s) = (if (bb s)
            (id)
          (rhoh (h (tau s)) s))
```

using the axioms

(bb **s**) ∧ (rt **a s**) ⇒ (h **s**) = (id)

¬(bb **s**) ∧ (rt **a s**) ⇒ (rhoh (h (tau **s**)) **s**)) = (h **s**)

We constrain a function op that relates g to h using the axioms

(bb **s**) ∧ (rt **a s**) ⇒ (op **a** (id) **s**) = **a**

¬(bb **s**) ∧ (rt **a s**) ⇒ (op (rho **a s**) (h (tau **s**)) (tau **s**)) = (op **a** (rhoh (h (tau **s**)) **s**) **s**)

from which we prove[6]

(rt **a s**) ⇒ (a-g **a s**) = (op **a** (h **s**) **s**)

We constrain a function that computes a bottom object under tau

```
(hs s) = (if (bb s)
             s
             (hs (tau s)))
```

using the axioms

(bb **s**) ⇒ (hs **s**) = **s**

¬(bb **s**) ⇒ (hs (tau **s**)) = (hs **s**)

Finally, we prove

```
(rt a s) ⇒ (g a s) = (if (bb s)
                         (qt a s)
                         (qt (op a (h s) s) (hs s)))
```

This theory allows g to be replaced by an expression whose only "new" functions are h and hs, both of which have induction patterns depending only on **s**. It finds use when the **a** component of state has been instantiated in a way to block an induction patterned after g, but the **s** component has not. We will see an example of this in section 4.3.

3.3 Function Equivalence and Alternative Induction Theory

This theory can be used to prove equivalence between alternative representations of the same function or in developing induction patterns different from that suggested by a recursive function definition. We constrain two tail recursive functions[7]

(fn1 **s**) = (if (b1 **s**)

6. Notice that the above axiom states that (op **a** (h **s**) **s**) is invariant under the recursion satisfied by a-g.
7. Notice that **s** is strictly a formal parameter in definitions of fn1 and fn2, and the state spaces for the two functions may indeed have different structure.

(q1 **s**)
(fn1 (sigma1 **s**))

(fn2 **s**) = (if (b2 **s**)
(q2 **s**)
(fn2 (sigma2 **s**))

using the axioms

(b1 **s**) ⇒ (fn1 **s**) = (q1 **s)**

¬(b1 **s**) ∧ (p **s**) ⇒ (fn1 (sigma1 **s**)) = (fn1 **s**)

(b2 **s**) ⇒ (fn2 **s**) = (q2 **s)**

¬(b2 **s**) ∧ (p **s**) ⇒ (fn2 (sigma2 **s**)) = (fn2 **s**)

¬(b1 **s**) ∧ (p **s**) ⇒ (p (sigma1 **s**))

where p plays the role of rt above. We constrain the measure function

(e0-ordinalp (measure1 **s**))

¬(b1 **s**) ⇒ (e0-ord-< (measure1 (sigma1 **s**)) (measure1 **s**))

The mapping id-alt between the domains of fn1 and fn2 is constrained by

¬(b1 **s**)) ∧ (p **s**) ⇒ (id-alt (sigma1 **s**)) = (sigma2 (id-alt **s**))

(p **s**) ⇒ (b2 (id-alt **s**)) = (b1 **s**)

(b1 **s**) ∧ (p **s**) ⇒ (q2 (id-alt **s**)) = (q1 **s**)

From the above properties we prove

(p **s**) ⇒ (fn1 **s**) = (fn2 (id-alt **s**))

id-alt plays the critical role in this theory. It may be used to restructure the state space of fn1 by combining state variables into a single variable (yielding an alternative definition of fn1), or to create an alternative induction (in which case fn1 and fn2 are the same). It is the most flexible of the theories, in that it provides the greatest freedom in choosing an induction pattern. We will see an example in section 4.4 where a variable that remains constant throughout the recursive call forms the basis for the induction.

4.0 Examples

We work through a simple example using Robert Krug's September 2003 modified version of ACL2 and his arithmetic-4 proof library. A more complex example that

presents a comparison between Krug's modified ACL2 and NQTHM, using this author's modularithmetic-98 proof library, is included in the appendix. Both examples uses the early Mostek 6502 microprocessor, which has an 8-bit accumulator A, two 8-bit index registers X and Y, a carry flag C, a test for 0 flag Z, and various variables representing single bytes in memory. The simpler example sums the integers from 1 to N.

```
        LDA #0      ; load A immediate with the constant 0
        CLC         ; clear the carry flag
LOOP    ADC N       ; add with carry N to A
        DEC N       ; decrement N
        BNE LOOP    ; branch if N is non-zero to LOOP
```

The weakest precondition at LOOP for postcondition A = (**nsave***(**nsave**+1))/2 is[8]

```
(defun wp-loop (n a c nsave)
 (declare (xargs :measure (dec n)))
 (if (equal (dec n) 0)
    (equal (mod (+ c (+ a n)) 256)
           (floor (* nsave (+ 1 nsave)) 2))
  (wp-loop (dec n)
           (mod (+ c (+ a n)) 256)
           (floor (+ c (+ a n)) 256)
           nsave)))
```

where (dec n) is defined by

```
(defun dec (n)
  (if (zp n)
     255
   (+ -1 n)))
```

The weakest precondition at the beginning of the program is

```
(defun wp-1 (n nsave)
 (wp-loop n 0 0 nsave))
```

The proof goal is

```
(defthm wp-loop-is-correct
  (implies (and (not (zp n))
                (equal nsave n)
                (< (floor (* n (+ 1 n)) 2) 256))
           (wp-1 n nsave)))
```

8. **nsave** is commonly called a "ghost variable" in the Floyd-Hoare parlance. It is used to refer to the initial value of **n** in the postcondition. One assumes **n** = **nsave** in the precondition.

4.1 Proof by Generalization

This is the simplest of the proofs, requiring only two support lemmas:

```
(defthm equal-transpose-constant
  (equal (equal (+ -1 a) 0)
         (equal a 1)))

(defthm wp-sum-loop-generalization
  (implies (and (not (zp n))
                (< (+ a (floor (* n (+ 1 n)) 2)) 256)
                (natp a)
                (equal c 0)
                (natp nsave))
           (equal (wp-loop n a c nsave)
                  (equal (+ a (floor (* n (+ 1 n)) 2))
                         (floor (* nsave (+ 1 nsave)) 2)))))
```

where the predicate natp recognizes a natural number.

```
(defmacro natp (a) '(and (integerp ,a) (<= 0 ,a)))
```

The former lemma is needed because of a bug in arithmetic-4. The latter accomplishes the generalization step. It relates wp-loop on arbitrary arguments to an expression comprised entirely of previously known functions. The discovery of this generalization is currently a user assisted activity, but not without structure. The postcondition states that the final value of the A register is (**nsave***(**nsave**+1)/2 when the initial value of A and C are 0. Since all operations on the A register behave linearly, we may express its final contents when A is not initially 0 as **a**+(**nsave***(**nsave**+1)/2. Since **nsave** represents the initial value of **n**, we can replace **nsave** with **n**. Finally, one needs to check whether the resulting generalization satisfies the recursive definition of wp-loop. In this case it does, since incrementing **a** by **n** and decreasing **n** by 1 preserves the value of **a**+(**n***(**n**+1)/2. The proof of wp-loop-is-correct follows automatically from these lemmas.

4.2 Proof Using the Loop Invariant Theory

The generic theories require that the variables **n**, **a**, **c** and **nsave** be represented as components of a single state variable **s**. The following definitions accomplish this

```
(defun n (s) (car s))
(defun a (s) (cadr s))
(defun c (s) (caddr s))
(defun nsave (s) (cadddr s))
```

This lemma instantiates the generic loop invariant theory.

```
(defthm wp-sum-loop-invariant
```

```
 (implies (and (not (zp (n s)))
               (< (+ (a s) (floor (* (n s) (1+ (n s))) 2)) 256)
               (natp (a s))
               (equal (c s) 0)
               (natp (nsave s))
               (equal (+ (a s) (floor (* (n s) (1+ (n s))) 2))
                      (floor (* (nsave s) (1+ (nsave s))) 2)))
          (wp-loop (n s) (a s) (c s) (nsave s)))
:hints
(("Goal"
  :use
  ((:functional-instance
    wp-is-weakest-invariant
    (b (lambda (s) (equal (dec (n s)) 0)))
    (qp (lambda (s) (equal
                     (mod (+ (c s) (+ (a s) (n s))) 256)
                     (floor (* (nsave s) (1+ (nsave s))) 2))))
    (wp (lambda (s) (wp-loop (n s) (a s) (c s) (nsave s))))
    (measure (lambda (s) (dec (n s))))
    (sigma (lambda (s) (list (dec (n s))
                             (mod (+ (c s) (+ (a s) (n s))) 256)
                             (floor (+ (c s) (+ (a s) (n s))) 256)
                             (nsave s))))
    (r (lambda (s) (and (not (zp (n s)))
                        (< (+ (a s) (floor (* (n s) (1+ (n s))) 2)) 256)
                        (natp (a s))
                        (equal (c s) 0)
                        (natp (nsave s))
                        (equal (+ (a s) (floor (* (n s) (1+ (n s))) 2))
                               (floor (* (nsave s) (1+ (nsave s))) 2)))))))))
```

Notice that all functional instances can be automatically derived from the definition of wp-loop except for the loop invariant r. It is no accident that the form of r bears a close resemblance to the theorem wp-sum-loop-generalization. In fact any proof by generalization may be recast into a proof using the loop invariant theory. This is true because the hypotheses of wp-sum-loop-generalization must be a loop invariant in order for induction patterned after wp-loop to be effective. The conclusion of wp-sum-loop-generalization states an equivalence between wp-loop and an expression not involving wp-loop. Such an expression must be a loop invariant because wp-loop is the weakest loop invariant. Therefore the conjunction of this expression and the hypotheses is a loop invariant. This is precisely the form of r. The loop invariant theory does not require that r be chosen in this fashion, and offers the flexibility to choose stronger invariants when the preconditions allow.

This lemma restates wp-sum-loop-invariant in terms of the “flat” state space.

```
(defthm wp-sum-loop-invariant-flat
```

```
  (implies (and (not (zp n))
                (< (+ a (floor (* n (1+ n)) 2)) 256)
                (natp a)
                (equal c 0)
                (natp nsave)
                (equal (+ a (floor (* n (1+ n)) 2))
                       (floor (* nsave (1+ nsave)) 2)))
           (wp-loop n a c nsave))
 :hints
 (("Goal"
   :in-theory (disable wp-sum-loop-invariant)
   :use (:instance
         wp-sum-loop-invariant
         (s (list n a c nsave))))))
```

The proof of wp-loop-is-correct follows automatically from this lemma.

4.3 Tail Recursion Theory

In this theory we choose to represent the **a** component of state directly as a natural number and the **s** component as a list with the following accessors.

```
(defun n (s) (car s))
(defun c (s) (cadr s))
(defun nsave (s) (caddr s))
```

We define the primitive recursive instantiation of h from the generic theory.

```
(defun wp-loop-h (s)
  (declare (xargs :measure (dec (n s))))
  (if (equal (dec (n s)) 0)
      0
    (+ (n s)
       (wp-loop-h (list (dec (n s)) (c s) (nsave s))))))
```

We now instantiate g=h from the tail recursion theory

```
(defthm wp-loop-g=h
  (implies (and (not (zp (n s)))
                (natp (nsave s))
                (natp a)
                (equal (c s) 0)
                (< (+ a (floor (* (n s) (+ 1 (n s))) 2)) 256))
           (equal (wp-loop (n s) a (c s) (nsave s))
                  (if (equal (dec (n s)) 0)
                      (equal (mod (+ a (n s)) 256)
```

```
                              (floor (* (nsave s) (+ 1 (nsave s))) 2))
                        (let ((a (+ a (wp-loop-h s)))
                              (s (list 1 (c s) (nsave s))))
                          (equal (mod (+ a (n s)) 256)
                                 (floor (* (nsave s) (+ 1 (nsave s))) 2))))))
  :hints
  (("Goal"
    :use
    ((:functional-instance
      g=h
      (bb (lambda (s) (equal (dec (n s)) 0)))
      (qt (lambda (a s) (equal (mod (+ a (n s)) 256)
                               (floor (* (nsave s) (+ 1 (nsave s))) 2))))
      (g (lambda (a s) (wp-loop (n s) a (c s) (nsave s))))
      (measure-g (lambda (s) (dec (n s))))
      (tau (lambda (s) (list (dec (n s)) (c s) (nsave s))))
      (rho (lambda (a s) (mod (+ a (c s) (n s)) 256)))
      (rhoh (lambda (a s) (+ a (n s))))
      (h (lambda (s) (wp-loop-h s)))
      (rt (lambda (a s) (and (not (zp (n s)))
                             (natp (nsave s))
                             (natp a)
                             (equal (c s) 0)
                             (< (+ a (floor (* (n s) (+ 1 (n s))) 2)) 256))))
      (id (lambda () 0))
      (op (lambda (a x s) (if (equal (dec (n s)) 0)
                              a
                              (+ a x))))
      (hs (lambda (s) (if (equal (dec (n s)) 0)
                          s
                          (list 1 (c s) (nsave s))))))))))
```

Once the user has stated this theorem, all functional instances may be derived from previously defined functions. The user involvement is confined to identifying the primitive recursive function wp-loop-h, and relating it to wp-loop through the instantiation of op. We flatten the state space using the following lemma.

```
(defthm wp-loop-g=h-flat
  (implies (and (not (zp n))
                (natp nsave)
                (natp a)
                (equal c 0)
                (< (+ a (floor (* n (+ 1 n)) 2)) 256))
           (equal (wp-loop n a c nsave)
                  (if (equal (dec n) 0)
                      (equal (mod (+ a n) 256)
                             (floor (* nsave (+ 1 nsave)) 2))
```

```
                (equal (mod (+ 1 a (wp-loop-h (list n c nsave))) 256)
                       (floor (* nsave (+ 1 nsave)) 2)))))
 :hints (("Goal"
          :use (:instance wp-loop-g=h
                          (a a)
                          (s (list n c nsave))))))
```

Having transformed the tail recursive function to a primitive recursive function, we need to prove that the primitive recursive function has the desired closed form.

```
(defthm equal-transpose-constant
  (equal (equal (+ -1 a) 0)
         (equal a 1)))

(defthm wp-loop-h-closed
  (implies (not (zp (n s)))
           (equal (wp-loop-h s)
                  (+ -1 (floor (* (n s) (+ 1 (n s))) 2)))))
```

As with the proof by generalization, derivation of this closed form can be motivated from the postcondition. The presence of -1 in the closed form is due to wp-loop exiting when **n** is 1 rather than 0. The remainder of the proof follows automatically.

4.4 Alternative Induction Theory

This theory uses the same representation for state as does the loop invariant theory.

```
(defthm wp-loop-fn1-as-fn2
  (implies (and (not (zp (n s)))
                (not (zp (nsave s)))
                (equal (c s) 0)
                (< (+ (a s) (floor (* (n s) (+ 1 (n s))) 2)) 256)
                (natp (a s))
                (<= (nsave s) (a s)))
           (equal (wp-loop (n s) (a s) (c s) (nsave s))
                  (wp-loop (n s)
                           (- (a s) (nsave s))
                           (c s)
                           (+ -1 (nsave s))))))
  :hints
  (("Goal"
    :use
    (:functional-instance
     fn1-as-fn2
     (fn1 (lambda (s) (wp-loop (n s) (a s) (c s) (nsave s))))
     (fn2 (lambda (s) (wp-loop (n s) (a s) (c s) (nsave s))))
     (b1 (lambda (s) (equal (dec (n s)) 0)))
```

```
(b2 (lambda (s) (equal (dec (n s)) 0)))
(q1 (lambda (s) (equal (mod (+ (c s) (a s) (n s)) 256)
                       (floor (* (nsave s) (+ 1 (nsave s))) 2))))
(q2 (lambda (s) (equal (mod (+ (c s) (a s) (n s)) 256)
                       (floor (* (nsave s) (+ 1 (nsave s))) 2))))
(sigma1 (lambda (s)
          (list (dec (n s))
                (mod (+ (c s) (a s) (n s)) 256)
                (floor (+ (c s) (a s) (n s)) 256)
                (nsave s))))
(sigma2 (lambda (s)
          (list (dec (n s))
                (mod (+ (c s) (a s) (n s)) 256)
                (floor (+ (c s) (a s) (n s)) 256)
                (nsave s))))
(p (lambda (s)
     (and (not (zp (n s)))
          (not (zp (nsave s)))
          (< (+ (a s) (floor (* (n s) (+ 1 (n s))) 2)) 256)
          (natp (a s))
          (<= (nsave s) (a s))
          (equal (c s) 0))))
(id-alt (lambda (s)
          (list (n s)
                (- (a s) (nsave s))
                (c s)
                (+ -1 (nsave s)))))
(measure1 (lambda (s) (if (zp (n s)) 256 (n s))))))))
```

The instantiation is highly repetitive since fn1 is the same as fn2 when deriving an alternative induction. All instantiations follow directly from the definition of wp-loop except for p and id-alt. The form of id-alt is motivated by a requirement to leave q1 invariant and to commute with sigma1. Subtracting 1 from **nsave** was driven by the identity (**nsave***(**nsave**+1))/2 = **nsave**+(**nsave***(**nsave**-1))/2 for nonzero **nsave**. It should be noted that given a recursive definition of multiplication for natural numbers, this identity can be automatically derived using function expansion and simplification. Since A is equated to (**nsave***(**nsave**+1)/2 in the postcondition, we see that A must be decremented by **nsave** to preserve the equality.

The flat state space version of wp-loop-fn1-as-fn2 is

```
(defthm wp-loop-fn1-as-fn2-rewrite
  (implies (and (not (zp n))
                (not (zp nsave))
                (equal c 0)
                (< (+ a (floor (* n (+ 1 n)) 2)) 256)
                (natp a)
```

```
                (<= nsave a))
           (equal (wp-loop n a c nsave)
                  (wp-loop n (- a nsave) c (+ -1 nsave))))
:hints
(("Goal"
  :use (:instance wp-loop-fn1-as-fn2
                  (s (list n a c nsave))))))
```

The proof of wp-loop-is-correct follows automatically using equal-transpose-constant.

5.0 Concluding Remarks

5.1 The Argument for Generic Theories

Since ACL2 and NQTHM both use bottom up rewriting, the evolution of a proof is controlled primarily by the local context. Decisions about whether a recursive function will be expanded are heavily influenced by the presence of specific subterms in the formula being proven. Lengthy rewriting of subterms may prevent progress toward a key rewrite at an outer level of the formula. The choice of induction, the number of case splits, and the use of elim rules are all influenced by local context. All of this is difficult to predict and control, especially if the set of rewrite rules does not adequately normalize subterms. Generic theories offer a means to hide the local context until the generic theorem is instantiated, thus providing more predictable control over the proof. They may be used in much the same manner as *proof plans* [3] to implement broad proof strategies.

5.2 Automated Support for Applying Generic Theories

The generic theories presented in this report essentially recast the search for a suitable generalization into a search for a suitable loop invariant, equivalent primitive recursive function, or a commuting substitution. This is helpful, especially when the search for a generalization proves to be difficult. The added user overhead in using the generic theories may be diminished by extracting information from the ACL2 and NQTHM databases and automatically formatting the ***:functional-instance*** hint within ACL2 or the ***functionally-instantiate*** event within NQTHM. Stereotypical events, such as the flat state space version of the functional instantiation of the generic theory, can also be mechanically generated. Beyond this, one could extend the ACL2 theorem prover and/or its proof libraries to provide automated support for choosing appropriate instantiations for generic theories. Generic theorems could be applied in much the same way as rewrite rules,[9] with an added feature that allows the user to specify a "choice" function to identify non soundness bearing instantiations, e.g. the contents of the ***:functional-instance*** hint.

9. For performance reasons, one would restrict such rules to apply only to new functions, i.e. those not built into ACL2 or part of its proof libraries.

5.3 A Comparison of the Generic Theories

The following tables display a raw count of the number of theorems needed to prove the example programs using each of the four methods with ACL2 and NQTHM.

Theorem Count for the Sum Program

	Generalization	Loop Invariant	Tail Recursion	Alternative Induction
ACL2	3	4	5	4
NQTHM	2	4	4	3

Theorem Count for the Multiply Program

	Generalization	Loop Invariant	Tail Recursion	Alternative Induction
ACL2	6	12	18	11
NQTHM	8	11	19	11

In all cases an attempt was made to minimize the theorem count. One should not place too much emphasis on small differences between the ACL2 and NQTHM statistics, especially since the author is more familiar with the NQTHM technology. The differences between the proof methods appear more significant, suggesting that proof by generalization leads to shorter proofs. This does not directly translate into ease of proof, which appears equally difficult across the methods and between theorem provers.

5.4 General Impressions

Based upon the above examples, it appears that ACL2 and NQTHM are roughly comparable in the degree to which they automate computer arithmetic proofs. This is not surprising, since both share the same basic design, which dates back to 1979.[10] By contrast, improvements in stand alone and integrated decision procedures have improved dramatically over the same period [4,5,8]. Comparable improvements in the design of general purpose theorem provers may be necessary before clear progress in proof library construction is discernible. More extensive benchmarks and metrics are needed to both measure progress and point research in the proper direction.

10. ACL2 does support a much richer array of numeric types, but this does not seem to have a large impact on the examples tried.

References

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. Bundy, Alan, “The Use of Explicit Plans to Guide Inductive Proofs,” Proceedings of CADE-9, eds Lusk,E. and Overbeek, R., Springer-Verlag Lecture Notes in Computer Science No. 310, 1988, pp 111-120.

4. Filliantre, Jean-Christophe and Owre, Sam and Ruess, Harald and Shankar, Natarajan, “ICS: Integrated Canonizer and Solver,” http://www.csl.sri.com/papers/cav01.

5. Flanagan, Joshi, Ou, and Saxe, “Theorem Proving Using Lazy Proof Explication,” Computer Aided Verification, 15th International Conference, CAV 2003, Warren A. Hunt, Jr. and Fabio Somenzi (Eds.), Springer-Verlag, ISBN 3-540-40524-0

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

7. Legato, Wilfred J., “A Weakest Precondition Model for Assembly Language Programs,” April 10, 2002, available at http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/contrib/legato/reference-7.pdf.

8. Ruess, Harald and Shankar, Natarajan, “Deconstructing Shostak,” Proc. of IEEE LICS 2001.