NQTHM proving sequential programs

Wim H. Hesselink[1] (whh148), September 26, 1996

Dept. of Mathematics and Computing Science
Rijksuniversiteit Groningen
Postbox 800, 9700 AV Groningen
The Netherlands

Abstract. This is a presentation of the application of the theorem prover NQTHM of Boyer and Moore to correctness proofs of imperative programs in the style of programming methodology. Predicates and programs are represented syntactically. The interpretation is based on NQTHM's interpreter `eval$`. A library is constructed for the interpretation and proofs of while–programs, possibly with array modification. Linear search and a regrouping algorithm for arrays are provided as examples.

1 Introduction

The purpose of this paper is to show how the theorem prover NQTHM of Boyer and Moore, cf. [2], can be used to certify correctness proofs of sequential programs, as developed according to the guidelines of programming methodology, cf. [4, 9, 12].

The sequential programming language used is as simple as possible. It only contains assignment, conditional choice, while–repetition, and sequential composition. It has no nondeterministic constructs. Its expression level, however, contains all NQTHM expressions and is therefore very powerful. We have omitted variable declarations and type restrictions, since the verifications associated can be delegated to compilers. We use the expression level to deal with arrays. So we do not provide separate array primitives at the command level.

One of the key concepts in programming methodology is partial correctness of a command s with respect to precondition p and postcondition q. This means that, if the initial state satisfies p and command s terminates, the final state satisfies q. Such a specification is called a Hoare triple and is usually written $\{p\}\ s\ \{q\}$. Since the theorem prover NQTHM uses the syntax of Lisp, we shall use the notation `(hoare p s q)` instead. Here, the predicates `p` and `q` are arguments of the function `hoare`. Since NQTHM does not allow higher order functions, the predicates and commands are represented syntactically, i.e., by explicit value terms in the NQTHM logic. These terms are interpreted by means of NQTHM's interpreter `eval$`. The programming language is defined by means of an operational semantics. The rules of the so-called axiomatic semantics are proved mechanically. They include both partial and total correctness.

In this way an NQTHM library is constructed that can be used for correctness proofs of many while–programs. The library and its use heavily depend on the way NQTHM treats and interprets explicit value terms. Part of this paper therefore is a kind of tutorial for explicit value terms and their interpretation. Other aspects of NQTHM will be treated more casually. We may occasionally simplify the truth about NQTHM slightly. The real sources for NQTHM are the handbook [2] and the excerpts in [3].

The library file can be obtained from `ftp.cs.rug.nl`, see Section 10.

1.1 The choice for NQTHM

Program derivation often requires the development of small special-purpose mathematical theories. It follows that a mechanical program prover must be powerful enough to deal with such

[1]email: wim@cs.rug.nl

theories and hence must contain a general-purpose theorem prover. General purpose theorem provers have been constructed in various flavors, so there is no need to construct a new one. The choice of a theorem prover is difficult. There are various criteria and it is hard to get unbiased assessments of the provers. See [14]. Let us mention the following aspects:

1. The language of the prover, its flexibility and its expressive power.
2. Theorem-proving capabilities: available decision procedures and heuristics, execution speed in case no user interaction is needed.
3. Certification: what kind of correctness certificate is created? Is the certificate easy to interpret? Is it possible to create false certificates?
4. Interactivity: what means are available to guide the prover to its goal? How easy is it to interpret the output of the prover when it does not reach the goal?
5. Availability, stability, trustworthiness, support, etc ...

In our opinion, the theorem prover NQTHM scores high on the points 2, 3, and 5. Its score on point 1 is disputed. It is argued that NQTHM's assertion language has a rigid syntax, is weakly typed, and is not very expressive. NQTHM is capable, however, of proving theorems about the "interpretation of quotations of terms". This is a powerful mechanism that can be used to construct bounded or unbounded quantifications.

With respect to point 4, NQTHM must be guided by somewhat implicit means, e.g., the order of the hypotheses of a lemma influences the applicability of the lemma. This is the price for NQTHM's heuristic power: it uses the hypotheses to find adequate instantiations. NQTHM can be guided by means of a hint to use or not to use specified facts known to it. NQTHM misses tactics and tacticals to specify a complete proof of a lemma, such as are available in, e.g., HOL, cf. [7].

We regard NQTHM's output in case of failure as quite informative, but it is important to stress that the user must understand NQTHM's language to interpret this output and to understand how this output has been influenced by the input. We regard it therefore as useless to provide a more flexible input language by constructing a preprocessor for NQTHM (even in combination with a postprocessor).

In conclusion, we have chosen for NQTHM because of its strong theorem proving capabilities and in spite of a certain lack of elegance.

1.2 Ways to use a prover to reason about programs

As indicated above, we have chosen to use an existing reliable and powerful theorem prover as a starting point. There are many ways of adapting an existing tool to construct a proof checker for a given language, see [17]. One approach is to use the existing tool as a subroutine to a newly constructed proof checker. This approach is used in the TLA [13] proof checker where the Larch Prover (LP) [6] is used as a back-end theorem prover [5]. It is also possible to encapsulate the prover by providing a special purpose interface, i.e. a parser/unparser which translates the programming logic into the logic of the prover and vice versa. This is the way the duration calculus proof assistent of [17] is built on top of the Prototype Verification System (PVS) of [16]. It is also possible to express the semantics of the programming language directly in the language of the prover, as is done for instance by Hooman [11], who also uses the theorem prover PVS. Other projects have used HOL in the same way, eg. [8, 1].

We have chosen not to hide or encapsulate the prover. There are two reasons. Firstly, the user must be able to guide the prover. In our view, this requires the user to have knowledge and experience with the prover and its language. We expect that an encapsulation could work nicely for simple problems, but become a hindrance when the proof is harder.

A second argument against hiding is that we expect the proof system to yield a certificate of validity. If the prover is accepted as sound, an input file that leads to a complete proof may be regarded as a certificate,– at least if the proof obligations are adequate and complete, and the axioms introduced are valid. When the prover is encapsulated in a new tool, however, the soundness of the combination is questionable again.

We have therefore chosen to work directly with the prover. The question then comes up how to represent programs and specifications of programs. As mentioned above, Hooman [11] uses a semantic encoding. For example, on p. 30, he encodes

```
r:= r - y ; z:= z + 1
```

as the function

```
seq(assign(r, LAMBDA s: val(s)(r) - val(s)(y)),
    assign(z, LAMBDA s: val(s)(z) + 1) ) .
```

In [1], almost the same representation is used, but function `val` is not needed.

Of course, such an encoding can be generated mechanically by some preprocessor, but then the soundness of the preprocessor is a new proof obligation. We therefore prefer to use a syntactic encoding of the program. Below we propose an imperative programming language in which the above fragment is written as

```
'( (put r (difference r y))
   (put z (plus z 1)) ) .
```

This is much closer to an actual programming language. In particular, the state variable `s`, the functions `val` and `seq`, and the lambda abstraction have been removed. In a simple program, the semantic overhead may be acceptable, but in larger programs it can soon be unmanageable. In [10], we had a program of 32 elementary commands. We therefore chose for a syntactic representation of the program. Yet we used a semantic representation of the specification. In fact, the syntactic layer was added in the last stages of the project since we regarded an earlier semantic description of the program as unsatisfactory. The mixture of syntax and semantics in the proof obligations of [10] is not quite elegant.

In the present paper it is our purpose to develop proof methods that follow as close as possible the well established ways of programming methodology. These ways consist of syntactic manipulations of predicates, but at some level a translation from syntax to semantics is necessary. In order to treat interesting programs and specifications we need syntactic representations of all functions and operators that are being used. It is cumbersome and error–prone to use ad–hoc syntactic representations. Therefore, it is important that the theorem prover itself has the possibility for syntactic treatment of terms with an associated semantic interpretation. The theorem prover NQTHM offers this facility in its function `eval$`. This function interprets quotations of terms with respect to a given list that associates values to variables that occur in the terms. The mixture of terms and quotations of terms has some similarity with aspects of higher order functions. In fact, we also use the syntactic level for the introduction of universal quantification.

1.3 Working with NQTHM

In this subsection we briefly sketch how to work with NQTHM. We refer to the Handbook [2] for more information.

The interaction with NQTHM is a dialogue in which the user submits definitions and lemmas to the prover and the prover ideally answers by accepting the definitions and proving the lemmas. In this way a data base of known facts (rewrite rules) is built. In practice the user often submits a lemma the prover cannot prove (often because it is not valid). In that case, the user must try and diagnose the failure by inspecting NQTHM's output. It may be helpful to submit an auxiliary lemma first, or to instruct NQTHM that some definition must (or must not) be unfolded.

NQTHM is deterministic and it only backtracks when it starts a proof by induction. Therefore the search path of NQTHM is very important. This path can be influenced, implicitly or explicitly, by many decisions of the user. For example, as mentioned above, the order of the hypotheses of a lemma may influence the way the lemma can be used as a rewrite rule later. We give an example of this in the discussion of Lemma (8) in Section 5.

The library we have created consists of a number of definitions and lemmas that NQTHM can prove without user intervention. This leads to a data base, which can be used as a starting point to prove the correctness of while–programs. A second file consists of correctness proofs of three simple while–programs. Here again no user interaction is needed. If the reader wants to experiment, they can slightly modify the programs or the proofs and see where and how the prover may fail.

1.4 Overview of the paper

Section 2 describes some aspects of NQTHM in relation to the concept of state for the imperative programs. In Section 3, we describe the list concepts of NQTHM and the interpretation of quotations of terms. In Section 4, we introduce the programming language and the weakest preconditions for straight-line programs. Hoare triples for the specification of commands are introduced in Section 5. Termination and total correctness of commands is treated in Section 6. In Section 7, we give an NQTHM correctness proof of a version of linear search. The primitives for arrays and array modification are described in Section 8. Section 9 contains a correctness proof for a nontrivial array-modifying algorithm. Finally, Section 10 contains a short description of the NQTHM library files which have been constructed and which are available for inspection. The appendix contains the complete file for the proof of the array-modifying program.

1.5 NQTHM notation

Many proofs published by Boyer, Moore, and their associates (eg. [15]) rephrase definitions and lemmas into more traditional notation to make it more broadly accessible. Since we want to explain how NQTHM can be used to verify proofs, we cannot do so. Actually, it is our experience, eg. while reading [15] with a group of non–users of NQTHM, that rendering into more traditional notation frequently raises questions that can only be answered by discussion of the NQTHM notation. Finally, this paper relies on the interpretation of quotations of terms for which a more traditional notation could be utterly confusing.

2 Representation and modification of the state

First, very briefly, some aspects of NQTHM's language and logic. NQTHM's language is a dialect of pure LISP. In particular, all operators are prefix functions. The application of a function `fn` on two arguments `x` and `y` is denoted `(fn x y)`. Function `cons` is the operator for pair formation. The functions `car` and `cdr` yield the first and second component of a pair, respectively. In order to enter list structures deeper, there are abbreviation functions like `caar`, `cadr`, `cdaar`, etc., defined as the composition of functions `car` or `cdr` for every `a` or `d`, respectively. For example,

```
(cdaar x) = (cdr (car (car x))) .
```

NQTHM's truth values are `(true)` and `(false)`, which can be abbreviated to `t` and `f`. The most fundamental logical operator is "`if`", characterized by the axioms

(0) $\quad \mathtt{x} = \mathtt{f} \;\Rightarrow\; \mathtt{(if\ x\ y\ z)} = \mathtt{z}$,
$\quad\quad \mathtt{x} \neq \mathtt{f} \;\Rightarrow\; \mathtt{(if\ x\ y\ z)} = \mathtt{y}$.

In fact, NQTHM is weakly typed and its functions are almost always total: when `x` is not a truth value, the term `(if x y z)` is welldefined and equal to `y`.

The operational semantics of an imperative programming language requires the concept of state. The state of a computer program determines the values of all program variables. So it can be represented by a list of pairs where each pair consists of a variable and the associated value. In NQTHM, such a list is called an association list. The function that retrieves the first value with given key is defined by

```
  (lookup key x)
= (if (nlistp x) 0
      (if (equal key (caar x)) (cdar x)
          (lookup key (cdr x)) ) ) .
```

The first line says that `(lookup key x) = 0` if `x` is not a list, or rather, if it is empty when regarded as a list. If `x` is a nonempty list of key-value pairs, the first pair is `(car x)` and the first key is `(car (car x))`, abbreviated `(caar x)`. The corresponding value is `(cdr (car x))`, abbreviated `(cdar x)`. If `key` differs from `(caar x)`, function `lookup` is called recursively on the tail of list `x`, denoted by `(cdr x)`.

Our library begins with the definition of a function to modify the state. This function is called `putassoc`, and is given by

```
  (putassoc var w x)
= (if (nlistp x) (cons (cons var w) nil)
      (if (equal var (caar x))
          (cons (cons var w) (cdr x))
          (cons (car x) (putassoc var w (cdr x))) ) ) .
```

If `x` represents the empty list, the first line specifies that the result is an association list with `(cons var w)` as its only element. Otherwise, the first key-value pair with key equal to `var` gets the new value `w`. After these definitions NQTHM easily proves the crucial identity

```
(1)     (lookup key (putassoc var w x))
      = (if (equal key var) w
            (lookup key x) ) .
```

3 Lists, interpretation and explicit value terms

As shown above, the operator for pair formation is `cons`. This operator is used repeatedly in the creation of lists. Long expressions with repeated cons's are abbreviated in NQTHM by means of the symbols `list*` and `list` in the following way:

```
(list* x) = x ,
(list* x ... z) = (cons x (list* ... z)) ,
(list x ... z) = (list* x ... z nil) .
```

For example, we have

```
(list* a b c) = (cons a (cons b c)) ,
(list) = nil ,
(list m n) = (cons m (cons n nil)) .
```

In the terms of NQTHM, words are used as function symbols when preceeded by an open parenthesis and as free variables otherwise. We also need word constants, so-called literal atoms or litatoms. Such a constant is represented by a quote followed by the word and is regarded as a quotation of the word. For example, the term `(equal 'm 'n)` equals `(false)`, whereas the term `(equal m 'n)` contains the free variable `m` that may (or may not) be equal to the litatom `'n`.

NQTHM's interpretation mechanism is built upon the function `apply$` which interprets a litatom as the corresponding function symbol (if the latter is known to the system). Function `apply$` takes two arguments. The first argument should be a quotation of a known function symbol. The second argument is regarded as the list of arguments of the function. For example, `apply$` satisfies

```
(apply$ 'plus (list m n)) = (plus m n) .
```

The function `eval$` interprets quotations of terms. It uses `apply$` for the interpretation of litatoms in a functional position and it uses `lookup` and an association list for the interpretation of litatoms in a variable position. For example,

```
  (eval$ t '(plus '5 v) x)
= (apply$ 'plus (list 5 (lookup 'v x)))
= (plus 5 (lookup 'v x)) .
```

Before going into the details of `eval$` (and its first argument `t`), we have to discuss abbreviations, quotations and quotes.

Above we introduced a number of *abbreviations* which are available and which are useful for clarity and conciseness. For example, the symbols `t`, `f`, `caar`, `cdar`, `list*`, `list`, etc., all serve as (or in) abbreviations. In order to form quotations of terms, one has to be able to "expand" abbreviations. Somewhat surprizingly, the unabbreviated term for a natural number like `5` is `'5`. Similarly, `nil` is only an abbreviation of the litatom `'nil`.

A *quotation* of a term is obtained by putting a quote before the *unabbreviated* term. It follows that `'(plus '5 v)` is a quotation of `(plus 5 v)` since `'5` is the unabbreviated form of `5`. Similarly, `'(cons v 'nil)` is a quotation of `(cons v nil)` since `'nil` is the unabbreviated form of `nil`.

Quotes are not only used for the quotation of terms. More generally, they serve in the denotation of explicit value terms. We sketch this denotation mechanism briefly. There are the following rules. For any object `phi`, the notation `'phi` is a shorthand for `(quote phi)`. Under a quote, no rewriting is allowed, apart from replacing `'phi` by `(quote phi)`. Quotes can be moved inward and outward by the distribution laws

```
(2)     '(phi ... psi) = (list 'phi ... 'psi) ,
        '(phi ... psi . chi) = (list* 'phi ... 'psi 'chi) .
```

For example, we have

```
'(a '7 . c) = (list* 'a ''7 'c) = (cons 'a (cons ''7 'c)) ,
'((v . 5) (w . 7)) = (list (cons 'v '5) (cons 'w '7)) ,
(car '(plus '5 v)) = (car (list 'plus ''5 'v)) = 'plus ,
''5 = '(quote 5) = (list 'quote '5) ,
(cadr ''5) = '5 .
```

Notice that `'5` $=$ `5`, as an abbreviation, but `''5` $\neq$ `'5`. This does not lead to a contradiction since rewriting under a quote is not allowed.

We now turn to the function `eval$`. The idea is to make a function `ev` such that, if `u` is the quotation of a term, `(ev u x)` yields the value of `u` where all variables in `u` get values from the association list `x`. For a litatom, `ev` yields the value associated (by `x`). Every other nonlist is interpreted as itself. A term of the form `(list 'quote y)` is interpreted as `y`. A term like `(list 'fn y z)` is interpreted by

```
  (ev (list 'fn y z) x)
= (apply$ 'fn (ev y x) (ev z x)) ,
```

that is, as the application of function `fn` on the interpretation of the arguments. This is realized by the definition

```
  (ev u x)
= (if (litatom u) (lookup u x)
      (if (nlistp u) u
          (if (equal (car u) 'quote) (cadr u)
              (apply$ (car u) (evlist (cdr u) x)) ) ) ) ,
```

where function `evlist` interprets its first argument as a list of quotations of terms and applies `ev` to each of them, with respect to association list `x`. Function `evlist` is thus given by

```
  (evlist u x)
= (if (nlistp u) nil
      (cons (ev (car u) x)
            (evlist (cdr u) x) ) ) .
```

NQTHM does not allow mutual recursion, however. The reason for this will be given in the discussion of Formula (3) in the next section. Therefore, the above definitions are combined in one function `eval$` and the choice between `ev` and `evlist` is encoded by means of a flag:

```
  (eval$ flag u x)
= (if (equal flag 'list)
      (if (nlistp u) nil
          (cons (eval$ t (car u) x)
                (eval$ 'list (cdr u) x) ) )
      (if (litatom u) (lookup u x)
          (if (nlistp u) u
              (if (equal (car u) 'quote) (cadr u)
                  (apply$ (car u)
                          (eval$ 'list (cdr u) x) ) ) ) ) ) )
```

Here `t` (= `(true)`) serves as an arbitrary term that differs from the litatom `'list`. After this definition, the functions `ev` and `evlist` can be defined by

```
  (ev u x) = (eval$ t u x) ,
  (evlist u x) = (eval$ 'list u x) .
```

Terms intended to be interpreted by `eval$` will be called *syntactic terms*.

4 Assignment, substitution, and commands

We come back to the construction of the imperative programming language. We now provide assignments to modify the state by means of the function `putassoc` of Section 2. The assignment $v := E$ for a program variable v and an expression E requires the interpretation of the expression E in the current state and the assignment of the value delivered to v. We therefore define

```
  (modify var exp x)
= (putassoc var (eval$ t exp x) x) .
```

In the axiomatic semantics, the weakest precondition of postcondition Q under assignment $v := E$ is obtained from Q by substituting E for v. More generally, the value of an expression Q in the modified state is the value of Q' in the original state, where Q' is obtained from Q by substituting E for v. In order to formulate and prove this fact, we define substitution. Substitution requires the same kind of mutual recursion as `eval$`. So we define

```
 (substitute flag var exp u)
= (if (equal flag 'list)
      (if (nlistp u) nil
          (cons (substitute t var exp (car u))
                (substitute 'list var exp (cdr u)) ) )
      (if (nlistp u)
          (if (and (equal var u)
                   (litatom var) )
              exp u )
          (if (equal (car u) 'quote) u
              (cons (car u)
                    (substitute 'list var exp (cdr u)) ) ) ) )
```

Notice that function `substitute` acts on syntactic terms: its parameter `u` and its result are both intended to be interpreted by `eval$`.

Now, NQTHM is able to prove the following generalization of the assertion suggested above:

```
(3)     (eval$ flag u (modify var exp x))
      = (eval$ flag (substitute flag var exp u) x) .
```

This theorem illustrates why NQTHM has no mutual recursion. In fact, if `eval$` were replaced by `ev` and `evlist` and `substitute` were replaced by analogous functions `subst` and `sublist`, the theorem would consist of two versions, one for `ev` and `subst` and the other for `evlist` and `sublist`. In the proof of either version, however, the other version would be needed in the induction hypothesis.

We now define a command interpreter `exe` for a small imperative language. The language consists of structured commands. It has assignment, conditional choice, repetition, and sequential composition. Function `exe` takes three arguments and yields the new state. Its main arguments are a command `cmd`, which is analysed as a list, and an argument `x`, regarded as the old state. We use an integer argument `n` as a bound for the number of repetitions, roughly speaking. If this bound is violated or the initial state is `(false)`, function `exe` delivers `(false)`.

Function `exe` does not modify the state if the command is not a list. If `(car cmd)` is not a list, it is expected to be one of the key words `'put`, `'if`, `'while`. Then `cmd` is an assignment, a conditional choice, or a repetition, respectively. In other cases, function `exe` yields `(false)`. If `(car cmd)` is a list, `cmd` is regarded as the sequential composition of commands `(car cmd)` and `(cdr cmd)`. Accordingly, the definition is

```
 (exe n cmd x)
= (if (or (zerop n) (not x)) f
      (if (nlistp cmd) x
          (if (nlistp (car cmd))
              (if (equal (car cmd) 'put)
                  (modify (cadr cmd) (caddr cmd) x)
                  (if (equal (car cmd) 'if)
                      (if (eval$ t (cadr cmd) x)
                          (exe n (caddr cmd) x)
                          (exe n (cadddr cmd) x) )
                      (if (equal (car cmd) 'while)
                          (if (eval$ t (cadr cmd) x)
                              (exe (sub1 n) cmd
                                   (exe n (caddr cmd) x) )
                              x )
                          f ) ) )
              (exe n (cdr cmd) (exe n (car cmd) x)) ) ) ) .
```

See Section 2 for the symbols `caddr` and `cadddr`.

NQTHM only allows recursive definitions that can be guaranteed to terminate. The recursive definitions of `lookup`, `putassoc`, `eval$`, and `substitute` are justified by the observation that the recursive calls use `(car x)` or `(cdr x)` instead of parameter `x` and that `(listp x)` implies that `(car x)` and `(cdr x)` are both smaller than `x`. NQTHM provides this argument itself. In the case of function `exe`, however, NQTHM does not find an obviously decreasing measure. NQTHM accepts the definition of `exe`, when the user provides the measure "sum of `n` and the size of `cmd`".

Example. To show the applicability of this command interpreter, we give an encoding of the following linear search program for finding a value `y` in a bounded array `a`, that begins at index 0 and has #`a` elements.

```
     k := 0 ; m := # a
  ;  while k < m do if a[k] = y then m:=k else k:=k+1 fi od .
```

This program is claimed to establish the postcondition

(4) $\quad (\mathtt{m} = \ \mathtt{\#a} \quad \vee \quad \mathtt{a[m]} = \mathtt{y}) \quad \wedge \quad (0 \leq \mathtt{i} < \mathtt{m} \quad \Rightarrow \quad \mathtt{a[i]} \neq \mathtt{y})\ .$

We now assume that `length` is the encoding of # and that `(sub k a)` yields the `k`'th element of array `a` (see Section 8 for implementations of `length` and `sub`). Then this program can be encoded in the explicit value term

```
(5)      (prog)
       = '( ( (put k 0) (put m (length a)) )
            (while (lessp k m)
                   (if (equal (sub k a) y)
                       (put m k)
                       (put k (add1 k)) ) ) ) .
```

Notice that the two initial assignments have been grouped together. This is not necessary, but merely convenient for the correctness proof in Section 7 below. (End of example)

We now construct the weakest precondition `wp0` for commands without iteration. We do not exclude iterative commands but, in that case, we let the function yield the stronger precondition `(false)`. So the function must satisfy

```
     (wp0 (list 'put var exp) q) = (substitute t var exp q) ,
     (wp0 (list 'if b c d) q) = (list 'if b (wp0 c q) (wp0 d q)) ,
     (wp0 (list 'while b s) q) = '(false) .
```

In this way, we get

```
     (wp0 cmd q)
   = (if (nlistp cmd) q
         (if (nlistp (car cmd))
             (if (equal (car cmd) 'put)
                 (substitute t (cadr cmd) (caddr cmd) q)
                 (if (equal (car cmd) 'if)
                     (list 'if
                           (cadr cmd)
                           (wp0 (caddr cmd) q)
                           (wp0 (cadddr cmd) q) )
                     '(false) ) )
             (wp0 (car cmd)
                  (wp0 (cdr cmd) q) ) ) ) .
```

The state `x = f` serves for nontermination. We therefore define two special functions `evf` and `evt` for the interpretation of preconditions and postconditions under total and partial correctness. The definition is based on the special nature of NQTHM's untyped "`if`", see Formula (0).

```
(evf q x) = (if x (ev q x) f) ,
(evt q x) = (if x (ev q x) t) .
```

The following lemma shows that, indeed, function `wp0` yields a sufficient precondition for total correctness:

```
(6)     (implies (and (not (zerop n))
                      (evf (wp0 cmd q) x) )
                 (evf q (exe n cmd x)) ) .
```

The result for partial correctness follows:

```
(implies (evt (wp0 cmd q) x)
         (evt q (exe n cmd x)) ) .
```

The function `wp0` is not useful for while–commands like `(prog)`. In order to remove this deficiency, we shall develop an apparatus to deal with structured programs in a structured way.

5 Specification of commands

Specification of commands requires universal quantification. In fact, we have to express that, for every state that satisfies the precondition, the command does not terminate or terminates in a state that satisfies the postcondition. This can be expressed in NQTHM by taking the state as a free variable in a theorem. This way of expressing is not sufficient, however, when we need the specification as an assumption of some theorem. We therefore use a method to express universal quantification by means of witness functions characterized by axioms. This method goes back to Hilbert.

In order to express that a predicate `p` holds for all states, we postulate the existence of a function `witness` such that, if `p` does not hold in all states, it does not hold in the state `(witness p)`, — or equivalently, if `p` holds in `(witness p)`, it holds in all states. This introduction of an unspecified function with a characterizing property is done by announcing

```
(dcl witness (p)) ,

(add-axiom hilbert :
    (implies (eval$ t p (witness p))
             (eval$ t p x) ) ) .
```

Notice that the `dcl`–command only tells the prover that `witness` is a function of one argument. This implies, e.g.,

```
(implies (equal p q)
         (equal (witness p) (witness q)) ) .
```

This form of universal quantification is mainly used to express that some predicate implies another predicate in all states. We therefore construct a function `stronger`, such that `(stronger p q)` says that `p` implies `q` in all states:

```
(7)     (evalimplies p q x)
      = (eval$ t (list 'implies p q) x) ) ,

        (stronger (p q)
      = (evalimplies p q (witness (list 'implies p q))) ) .
```

Now NQTHM can be guided to prove:

```
(implies (and (stronger p q)
              (eval$ t p x) )
         (eval$ t q x) ) .
```

Remark. Function `evalimplies` can easily be eliminated. Its introduction here is useful, since it will enable us to design more helpful rewrite lemmas for the prover. See the end of Section 7. (End of remark)

In order to get a feeling for the applicability of the axiom the reader is advised to prove the lemmas:

```
(implies (and (stronger q r)
              (stronger p q) )
         (stronger p r) ) ,

(stronger (list 'and q r) q) ,

(implies (and (stronger p q)
              (stronger p r) )
         (stronger p (list 'and q r)) ) .
```

In these examples, predicate `(list 'and q r)` is the conjunction of the predicates `q` and `r`. This predicate must be distinguished from `'(and q r)`, which equals `(list 'and 'q 'r)`.

It is useful at this point to introduce NQTHM's backquote convention, which allows the alternative notation

```
`(and ,q ,r) = (list 'and q r) .
```

In other words, the backquote serves as the quote, but has the additional rule that `` `,phi `` equals `phi` for every term `phi`.

We now introduce a function `hoare` such that `(hoare p s q)` expresses the partial correctness $\{$p$\}$ s $\{$q$\}$ of command `s` for precondition `p` and postcondition `q`, as explained in Section 1. So we want to have

$$\texttt{(hoare p s q)} \;\equiv\; (\forall\, x, n :: \texttt{(evt p x)} \;\Rightarrow\; \texttt{(evt q (exe n s x))}) .$$

Again using Hilbert's idea, we express this equivalence by postulating the existence of two witness functions `xwit` and `nwit` that, depending on the parameters `p`, `s`, and `q`, yield a counterexample to the quantified expression if such a counterexample exists. The unspecified functions `xwit` and `nwit` are introduced to NQTHM by the declarations

```
(dcl xwit  (p s q)) ,
(dcl nwit  (p s q)) .
```

We then define and postulate

```
  (hoare p s q)
= (implies (evt p (xwit p s q))
           (evt q (exe (nwit p s q) s (xwit p s q))) ) ,

(add-axiom hoare-implies :
    (implies (and (hoare p s q)
                  (evt p x) )
             (evt q (exe n s x)) ) ) .
```

One of the first things one may want to prove about Hoare triples, is strengthening of the precondition: if predicate `p0` is stronger than precondition `p`, then `(hoare p s q)` implies `(hoare p0 s q)`. So the next step consists of proving the rules for strengthening of precondition and weakening of postcondition.

```
(implies (and (hoare p s q)
              (stronger p0 p) )
         (hoare p0 s q) ) ,

(implies (and (hoare p s q)
              (stronger q q0) )
         (hoare p s q0) ) .
```

The link between Hoare triples and weakest preconditions consists of

```
(implies (stronger p (wp0 s q))
         (hoare p s q) ) .
```

The composition rule for Hoare triples has the form

```
(8)     (implies (and (listp s0)
                      (hoare p s0 r)
                      (hoare r s1 q) )
                 (hoare p (cons s0 s1) q) ) .
```

The condition `(listp s0)` is necessary here, because function `exe` only yields sequential composition when `s0` is a list (like, e.g., `'(put k (add1 k))`).

A subtle point is the order of the premisses in (8). These premisses contain a free variable `r`. In every application of the lemma the prover will try to find an instantiation of `r` that matches the first premiss in which `r` occurs. In our experience, the present order often yields a better heuristic than when the premisses have been reversed.

Hoare triples are especially useful for repetitions, since they allow a discussion of partial correctness separated from the issue of termination. Hoare's repetition rule is

$$\{b \wedge j\}\ s\ \{j\} \quad \text{implies} \quad \{j\}\ \textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}\ \{\neg b \wedge j\} \ ;$$

here predicate j is called the invariant of the repetition. Indeed, we have guided NQTHM to prove:

```
(9)    (implies (hoare `(and ,b ,j) s j)
                (hoare j `(while ,b ,s)
                       `(and (not ,b) ,j) ) ) .
```

In this term, NQTHM's backquote convention becomes handy.

6 Termination and total correctness

Hoare triples only specify partial correctness. We also want to be able to specify and prove termination of programs. For this purpose, we introduce a function `termination`, such that `(termination p s)` expresses that command `s` terminates for all states `x` that satisfy `p`. Since NQTHM regards all values $\neq$ `f` as true, this can be formalized by

$$(10) \qquad \texttt{(termination p s)} \;\equiv\; (\forall\, \texttt{x} :: \texttt{(evf p x)} \;\Rightarrow\; (\exists\, \texttt{n} :: \texttt{(exe n s x)})) \ .$$

This equivalence ($\equiv$) is split into a forward implication ($\Rightarrow$) and a backward implication ($\Leftarrow$). Since the domain of `n` is nonempty, the forward implication is equivalent to

$(\forall$ x :: $(\exists$ n :: (termination p s) $\wedge$ (evf p x) $\Rightarrow$ (exe n s x))) .

This can be expressed by means of a so–called Skolem function for n that depends on p, s, and x. Since n only occurs in the term (exe n s x), which is independent of p, we omit the argument p. We choose the name time for the Skolem function and thus declare and postulate

```
(dcl termination (p s)) ,
(dcl time (s x)) ,

(add-axiom termination-implies :
    (implies (and (termination p s)
                  (evf p x) )
             (exe (time s x) s x) ) ) .
```

The backward implication of (10) can be rewritten in the following way:

$$
\begin{array}{ll}
 & (\forall\, \mathtt{x} :: ((\mathtt{evf\ p\ x}) \Rightarrow (\exists\, \mathtt{n} :: (\mathtt{exe\ n\ s\ x})))) \quad \Rightarrow \quad (\mathtt{termination\ p\ s}) \\
\equiv & \{\text{implication and De Morgan}\} \\
 & (\exists\, \mathtt{x} :: ((\mathtt{evf\ p\ x}) \wedge (\forall\, \mathtt{n} :: \neg(\mathtt{exe\ n\ s\ x})))) \quad \vee \quad (\mathtt{termination\ p\ s}) \\
\equiv & \{\text{the domains of } \mathtt{x} \text{ and } \mathtt{n} \text{ are nonempty}\} \\
 & (\exists\, \mathtt{x} :: (\forall\, \mathtt{n} :: ((\mathtt{evf\ p\ x}) \wedge \neg(\mathtt{exe\ n\ s\ x})) \quad \vee \quad (\mathtt{termination\ p\ s}))) \\
\equiv & \{\text{implication}\} \\
 & (\exists\, \mathtt{x} :: (\forall\, \mathtt{n} :: ((\mathtt{exe\ n\ s\ x}) \vee \neg(\mathtt{evf\ p\ x})) \quad \Rightarrow \quad (\mathtt{termination\ p\ s}))) \ .
\end{array}
$$

We therefore introduce a Skolem function nterm for x and postulate

```
(dcl nterm (p s)) ,

(add-axiom termination-implied :
    (implies (or (exe n s (nterm p s))
                 (not (evf p (nterm p s))) )
             (termination p s) ) ) .
```

Function termination has been introduced in this way in order to be able to define total correctness as the conjunction of partial correctness and termination. So we define

```
(defn dijkstra (p s q)
    (and (hoare p s q)
         (termination p s) ) ) .
```

Now it is easy to prove the wp0-rule:

```
(11)    (implies (stronger p (wp0 s q))
                 (dijkstra p s q) ) .
```

The repetition rule for total correctness (cf. [4, 9]) requires an invariant j as in Hoare's repetition rule, but it also requires termination of the loop body s under precondition $\mathtt{b} \wedge \mathtt{j}$, and it requires a variant function vf that decreases in each step of the loop body s.

Indeed, with some work, we were able to guide NQTHM to prove the variation of (9) for total correctness:

```
(implies (and (notoccurs 'list v (list j b vf))
              (notoccursp v s)
              (dijkstra `(and ,b ,j) s j)
              (hoare `(and (equal ,vf ,v) (and ,b ,j))
                     s `(lessp ,vf ,v) ) )
         (dijkstra j `(while ,b ,s)
                     `(and (not ,b) ,j) ) ) .
```

The first premiss says that `v` is a litatom that does not occur in the list of the terms `j`, `b`, and `vf`. The second premiss says that `v` also does not occur in the body `s` of the repetition. The third premiss says that `s` terminates and keeps `j` invariant. In view of the first two premisses, the last premiss implies that state function `vf` decreases when `s` is executed under precondition $b \wedge j$.

We next formulated and proved more powerful theorems in which an initialization is used to establish the invariant and in which a finalization may be used to establish a postcondition. The version without finalization reads:

```
(12)    (implies (and (notoccurs 'list v (list j b vf))        ; 1
                      (notoccursp v s)                          ; 2
                      (dijkstra `(and ,b ,j) s j)               ; 3
                      (hoare `(and (equal ,vf ,v) (and ,b ,j))
                             s `(lessp ,vf ,v) )                ; 4
                      (dijkstra p init j)                       ; 5
                      (stronger `(and (not ,b) ,j) q)           ; 6
                      (listp init) )                            ; 7
                 (dijkstra p
                        `(,init (while ,b ,s))
                        q ) ) .
```

The premisses of this theorem have been numbered for future reference. The order of the premisses is chosen in such a way that NQTHM, when it considers to apply the theorem, first searches for instantiations for `v`, `j`, and `vf` for which the first premiss has been established. In this way, NQTHM can be told which variable symbol, invariant, and variant function the user has in mind for the proof of the repetition.

7 The linear search program

In this section, we sketch the NQTHM proof of the linear search program `(prog)` given in (5). Postcondition (4) is translated into:

```
  (q1)
= '(and (or (equal m (length a))
            (equal (sub m a) y) )
        (implies (and (numberp i)
                      (lessp i m) )
                 (not (equal (sub i a) y)) ) ) .
```

The aim is to prove

```
(dijkstra '(true) (prog) (q1)) .
```

Unfortunately, this specification is also satisfied by the program

```
'( (put m 0)
   (put y (sub 0 a)) ) .
```

In order to disallow such erratic solutions we have to specify and prove that the program variables `n`, `a`, and `y` are not written and that the logical variable `i` is neither written nor read. Since the program is represented syntactically, this is quite easy. We let NQTHM prove:

```
(and (notoccursp 'i (prog))
     (notwritten 'a (prog))
     (notwritten 'y (prog)) ) ,
```

where `notwritten` verifies that its first argument does not occur as the first argument of a `'put` command in the second argument.

In order to discuss parts of the program, we define

```
(init1) =  '( (put k 0) (put m (length a)) ) ,
(b1)    =  '(lessp k m) ,
(s1)    =  '(if (equal (sub k a) y)
                (put m k)
                (put k (add1 k)) ) ,
```

and let NQTHM prove

```
(prog)  =  `(,(init1) (while ,(b1) ,(s1))) .
```

We then introduce an invariant:

```
   (j1)
 = '(and (or (equal m (length a))
             (equal (sub m a) y) )
         (and (implies (and (numberp i)
                            (lessp i k) )
                       (not (equal (sub i a) y)) )
              (numberp k) ) ) .
```

It may be mentioned here that NQTHM's function `and` may have more than two arguments, but that `eval$` is more rigid and interprets `and` as a function symbol with two arguments. Hence the repeated occurrence of `and` in `(j1)`.

In order to apply Formula (12), we then let NQTHM prove, for an arbitrary state `x`, that the invariant is initialized, that the invariant together with the negation of the guard implies the postcondition, and that `(j1)` is indeed an invariant:

```
(13)    (evalimplies '(true) (wp0 (init1) (j1)) x) ,
(14)    (evalimplies `(and (not ,(b1)) ,(j1)) (q1) x) ,
(15)    (evalimplies `(and ,(b1) ,(j1))
                     (wp0 (s1) (j1))
                     x ) .
```

We then define a variant function `vf1` and use a litatom `'v` to show that `vf1` decreases and that `'v` does not occur in `vf1`.

```
        (vf1)  =  '(difference m k) ,

(16)    (evalimplies `(and (equal ,(vf1) v) (and ,(b1) ,(j1)))
                     (wp0 (s1) (list 'lessp (vf1) 'v))
                     x ) .

(17)    (and (notoccurs 'list 'v (list (j1) (b1) (vf1)))
             (notoccursp 'v (s1)) ) .
```

Now all pieces are collected. Since NQTHM must not unfold the definitions of the constants involved, we disable all of them. After that, NQTHM can immediately prove:

```
(18)    (dijkstra '(true) (prog) (q1)) .
```

Formula (17) is used by NQTHM to determine adequate instantiations for `j`, `b`, `vf`, and `v`, such that the premisses 1 and 2 of (12) hold. The third premiss of (12) is obtained from (15), together with the definition of `stronger` and Formula (11). The premisses 4, 5, and 6 of (12) are proved in the same way from (16), (13), and (14). The last premiss only requires a syntactic check.

Remark. We introduced function `evalimplies` in (7) in order to be able to use lemmas like (13), (14), (15), (16). If function `evalimplies` is eliminated, and function `stronger` and these lemmas are formulated with the unfolding of `evalimplies`, the prover does not find these lemmas for the proof of (18). (End of remark)

8 Arrays and array modification

Arrays are modelled as lists with a constant length. We define a subscription function `sub` for the inspection of an array element and an update function `upd` for the modification of an array element. Inspection outside of the array yields `f`. Modification outside of the array has no effect. We use NQTHM's arithmetic functions `add1` and `sub1`, which increment and decrement with 1. The four relevant definitions are:

```
   (sub i a)
 = (if (or (nlistp a)
           (not (numberp i)) )
           f
       (if (equal i 0) (car a)
           (sub (sub1 i) (cdr a)) ) ) ) ,

   (upd i w a)
 = (if (or (nlistp a)
           (not (numberp i)) )
       a
       (if (equal i 0) (cons w (cdr a))
           (cons (car a) (upd (sub1 i) w (cdr a))) ) ) ) ,

   (length a)
 = (if (nlistp a) 0 (add1 (length (cdr a)))) ) ,

   (card v x)
 = (if (nlistp x) 0
       (if (equal v (car x))
           (add1 (card v (cdr x)))
           (card v (cdr x)) ) ) ) .
```

The first crucial relation between these definitions is:

```
(19)   (sub i (upd j w a))
     = (if (and (equal i j)
                (numberp i)
                (lessp i (length a)) )
           w
           (sub i a) ) .
```

This lemma should be compared with formula (1).

We provide a function `swap` for the interchange of array elements:

```
   (swap i j x)
 = (upd i (sub j x) (upd j (sub i x) x)) .
```

It is easy to prove that the functions `upd` and `swap` do not change the length of the array.

In order to formulate and prove that the modified array has the same "bag" of elements as the original one, we use function `card`, which counts the occurrences of an element in an array.

We then let NQTHM prove that, under reasonable circumstances, function `swap` does not change the number of occurrences:

```
(20)    (implies (and (numberp i)
                      (lessp i (length x))
                      (numberp j)
                      (lessp j (length x)) )
                 (equal (card v (swap i j x))
                        (card v x) ) ) .
```

9 Regrouping

One of the simplest array modifying problems is that of regrouping the elements of an array in such a way that all elements of the first part of the array satisfy some given criterion whereas all elements of the second part do not. We let the criterion be given by some function `cri`.

The task is to determine a value `p` and to permute the array, in such a way that the first `p` elements of the array satisfy `cri` and that the remaining elements do not. We postulate the invariance of `(card w a) = g` for arbitrary specification constants `w` and `g` to express that the bag of elements of array `a` does not change. In this way, we arrive at the following formal specification. The problem is to determine a command T that satisfies

$$
\begin{array}{l}
\textbf{var}\ \ \mathtt{p} : \textit{integer}\ ; \\
\{\mathtt{(card\ w\ a)\ =\ g}\} \\
T \\
\{\mathtt{(card\ w\ a)\ =\ g} \ \ \wedge\ \ 0 \le \mathtt{p} \le \#\mathtt{a} \\
\wedge\ \ (\forall\, \mathtt{i} : 0 \le \mathtt{i} < \mathtt{p} : \mathtt{cri(a[i])}) \\
\wedge\ \ (\forall\, \mathtt{i} : \mathtt{p} \le \mathtt{i} < \#\mathtt{a} : \neg\, \mathtt{cri(a[i])})\}\ \ ,
\end{array}
$$

with the additional requirement that `w` and `g` must not occur in command T.

The invariant is found by splitting the program variable `p` into two variables `p` and `q` with the invariant

$$
\begin{array}{ll}
J: & \mathtt{(card\ w\ a)\ =\ g} \ \ \wedge\ \ 0 \le \mathtt{p} \le \mathtt{q} \le \#\mathtt{a} \\
 & \wedge\ \ (\forall\, \mathtt{i} : 0 \le \mathtt{i} < \mathtt{p} : \mathtt{cri(a[i])}) \\
 & \wedge\ \ (\forall\, \mathtt{i} : \mathtt{q} \le \mathtt{i} < \#\mathtt{a} : \neg\, \mathtt{cri(a[i])})\ .
\end{array}
$$

We introduce the guard B : $\mathtt{p} < \mathtt{q}$. It is easy to see that the postcondition follows from $\neg B \wedge J$. In view of the precondition, it is also easy to initialize the invariant: it suffices to take

$$\mathtt{p} := 0\ ;\quad \mathtt{q} := \#\mathtt{a}\ .$$

In the repetition, we decrease the difference $\mathtt{q} - \mathtt{p}$ while preserving the invariant. This is done by

$$
\begin{array}{l}
\textbf{if}\ \ \mathtt{cri(a[p])}\ \ \textbf{then}\ \ \mathtt{p} := \mathtt{p} + 1 \\
\textbf{else}\ \ \mathtt{q} := \mathtt{q} - 1 \\
\quad ;\ \ \textbf{if}\ \ \mathtt{cri(a[q])}\ \ \textbf{then} \\
\qquad\quad \mathtt{a} := \mathtt{(swap\ p\ q\ a)} \\
\qquad ;\ \ \mathtt{p} := \mathtt{p} + 1 \\
\qquad\ \ \textbf{fi} \\
\textbf{fi}\ .
\end{array}
$$

We now turn to the NQTHM encoding of the program and its proof. We assume that a function `cri` of one argument is given. The encodings of the postcondition and the invariant require some representation of the universal quantifications involved. We have chosen to do this by means of the functions defined in

```
 (acri p a)
= (if (zerop p) t
      (and (acri (sub1 p) a)
           (cri (sub (sub1 p) a)) ) ) ,

 (anocri (q a)
= (if (lessp q (length a))
      (and (anocri (add1 q) a)
           (not (cri (sub q a))) )
      t ) .
```

For NQTHM, the definition of `anocri` is not obviously well-founded, but NQTHM can accept this definition when we suggest to take the difference $\#\mathtt{a} - \mathtt{q}$ as measure.

Now the precondition `(p2)`, the postcondition `(q2)`, the guard `(b2)`, and the invariant `(j2)` are defined by

```
(p2) =  '(equal g (card w a)) ,
(q2) =  `(and ,(p2)
            (and (numberp p)
              (and (leq p (length a))
                (and (acri p a) (anocri p a)) ) ) ) ,
(b2) =  '(lessp p q)) ,
(j2) =  `(and ,(p2)
            (and (numberp p)
              (and (leq p q)
                (and (numberp q)
                  (and (leq q (length a))
                    (and (acri p a)
                         (anocri q a) ) ) ) ) ) ) .
```

The finalization is proved in

```
(evalimplies `(and (not ,(b2)) ,(j2)) (q2) x) .
```

The initialization is defined and proved in

```
(defn init2 () '((put p 0) (put q (length a))))

(prove-lemma initialize2 (rewrite)
    (evalimplies (p2) (wp0 (init2) (j2)) x)
    ((expand (anocri (length (cdr (assoc 'a x)))
                     (cdr (assoc 'a x)) ))) )
```

Here, we have given the definition and the lemma in the NQTHM syntax, to show the hint we had to give to the prover. In this hint, the term `(cdr (assoc 'a x))` equals `(lookup 'a x)`, i.e., the value of variable `'a` in state `x`. The hint is needed, since it is not obvious to the prover that `(anocri (length a) a)` holds.

The body of the repetition is defined in

```
(s2) =  '(if (cri (sub p a))
             (put p (add1 p))
             ( (put q (sub1 q))
               (if (cri (sub q a))
                   ( (put a (swap p q a))
                     (put p (add1 p)) ) ) ) ) .
```

With respect to the second occurrence of `if` here, we use that a command of the form `'(if ,b ,s)` is interpreted by `exe` as skip when guard `b` is false (just like Pascal).

The analogue of (15) requires three lemmas. The first and the second one state that the quantifications remain valid when the array is swapped outside of the range of the quantification. In the second case, we had to tell NQTHM the right induction hypothesis.

Many human provers might have overlooked the third lemma: if `(s2)` is executed with precondition $\mathtt{q} = \mathtt{p} + 1$, the guards in `(s2)` ensure that $\mathtt{p} \leq \mathtt{q}$ remains valid. This is only obvious to NQTHM, when we have given it the lemma `subtlety`, which it proves without problem.

```
(prove-lemma swap-out-safe-pos (rewrite)
    (implies (and (acri p a)
                  (leq p i) (leq p j)
                  (numberp i)
                  (numberp j)
                  (leq i (length a))
                  (leq j (length a)) )
             (acri p (swap i j a)) ) ) ;

(prove-lemma swap-out-safe-neg (rewrite)
    (implies (and (anocri q a)
                  (lessp i q) (lessp j q)
                  (numberp i)
                  (numberp j)
                  (leq q (length a)) )
             (anocri q (swap i j a)) )
    ((induct (anocri q a))) ) ;

(prove-lemma subtlety (rewrite)
     (implies (and (lessp p q)
                   (numberp p)
                   (lessp (sub1 (sub1 q)) p) )
              (equal (sub1 q) p) ) ) .
```

After these preparations the remainder of the input to the prover is just like the linear search case. For the interested reader, the full input to the prover is given in the appendix.

10 Description of the library files

The main library file constructed is called "`hoare.events`". It supports a slightly bigger programming language than indicated in Section 4. In fact, the language also contains a `case`–statement of the form

```
(case exp (a0 s0) ... (an sn)) .
```

The meaning of this command is `si` if `ai` is the first constant equal to the value of expression `exp`. If `exp` differs from all constants `ai`, the case statement is equivalent to *skip*.

An interesting lemma in the library is one that allows the elimination of a specification constant `v` from the precondition of a Hoare triple, according to

```
(implies (and (notoccurs t v q)
              (notoccursp v s)
              (hoare p s q) )
         (hoare (substitute t v exp p) s q) ) .
```

Notice that the expression `exp` is interpreted in the initial state of command `s` and that there are no conditions on the variables that occur in `exp`. This lemma is often used in programming methodology to justify the introduction of specification constants in an annotated proof.

A second library file, called "`hoare-ex.events`", contains three simple example applications: the correctness of an exponentiation program, and the two example programs treated in this paper. The files can be obtained by anonymous ftp from `ftp.cs.rug.nl`, in the directory `/pub/boyer-moore`.

References

[1] R.J.R. Back, J. von Wright: Refinement concepts formalized in higher order logic. Report Åbo Akademi Ser. A, 85, 1989.

[2] R.S. Boyer, J S. Moore: A Computational Logic Handbook. Academic Press, Boston etc., 1988.

[3] R.S. Boyer, J S. Moore: A Computational Logic Handbook, Authorized Excerpts from a Proposed Second Edition, to be obtained by ftp from Computational Logic Inc. Information available at `nqthm-request@cli.com`.

[4] E.W. Dijkstra: A discipline of programming. Prentice–Hall 1976.

[5] U. Engberg, P. Grønning, L. Lamport: Mechanical verification of concurrent systems with TLA. In *Computer Aided Verification*. Springer Verlag 1992, LNCS 663.

[6] S.J. Garland, J.V. Guttag: LP: the Larch Prover. In E. Lusk, R. Overbeek (eds.): 9th Conferenence on automated deduction (CADE). Springer Verlag 1988, LNCS 310, pp. 748–749.

[7] M.J.C. Gordon, T.F. Melham (eds.): Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press, Cambridge, UK, 1993.

[8] M.J.C. Gordon: Mechanizing programming logics in higher-order logic. In G. Birtwistle, P.A. Subrahmanyam (eds.): *Current trends in hardware verification and theorem proving.* Springer Verlag 1989, pp. 387–439.

[9] D. Gries: The science of programming. Springer V. 1981.

[10] W.H. Hesselink: Wait-free linearization with a mechanical proof. Submitted.

[11] J. Hooman: Correctness of real time systems by construction. In: H. Langmaack, W.-P. de Roever, J. Vytopil (eds.): *Formal Techniques in real-time and fault-tolerant Systems.* Springer Verlag 1994. LNCS 863. pp. 19–40.

[12] A. Kaldewaij: Programming: the Derivation of Algrithms. Prentice Hall International, 1990.

[13] L. Lamport: The temporal logic of actions. Research Report 79, DEC, SRC, December 1991.

[14] P.A. Lindsey: A survey of mechanical support for formal reasoning. Software Engineering Journal, January 1988, 3–27.

[15] J Moore: A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. Formals Aspects of Computing **6** (1994) 60–91.

[16] S. Owre, N. Shankar, J.M. Rushby: User Guide for the PVS Specification and Verification System, Language, and Proof Checker (Beta Release). CSL, SRI International, Menlo Park, CA, February 1993 (three volumes).

[17] J.U. Skakkebæk, N. Shankar: A duration calculus proof checker: using PVS as a semantic framework. SRI, CSL Tech. Report, December 1993.

Appendix

Below follows the complete input to the prover for the correctness proof described in Section 9. The hints of the form `do-not-induct` are superfluous. During the design phase these hints serve to terminate a failing proof attempt. After each lemma, we give the time triple reported by NQTHM. The first number is the number of seconds spent for input of the lemma, the second number the number of seconds spent for the proof, the third number is the number of seconds spent printing information to the user. The numbers given here were obtained on a HP 9000-735.

```
(note-lib "hoare")  ;  load the library.
; file to be obtained from ftp.cs.rug.nl, in pub/boyer-moore.

(defn cri (x) (lessp 100 x)) ; just some boolean function
(disable cri)

(defn p2 ()    '(equal g (card w a)))

(defn acri (p a)
    (if (zerop p) t
        (and (acri (sub1 p) a)
             (cri (sub (sub1 p) a)) ) ) )

(defn anocri (q a)
    (if (lessp q (length a))
        (and (anocri (add1 q) a)
             (not (cri (sub q a))) )
        t )
    ((lessp (difference (length a) q))) ) ; a decreasing measure


(defn q2 ()
    `(and ,(p2)
       (and (numberp p)
         (and (leq p (length a))
              (and (acri p a) (anocri p a)) ) ) ) )

(defn b2 ()    '(lessp p q))

(defn j2 ()
    `(and ,(p2)
       (and (numberp p)
         (and (leq p q)
           (and (numberp q)
             (and (leq q (length a))
               (and (acri p a)
                    (anocri q a) ) ) ) ) ) )
```

```
(prove-lemma finalize2 (rewrite)
    (evalimplies `(and (not ,(b2)) ,(j2)) (q2) x) ) ; [ 0.0 0.1 0.1 ]

(defn init2 () '((put p 0) (put q (length a))))

(prove-lemma initialize2 (rewrite)
    (evalimplies (p2) (wp0 (init2) (j2)) x)
    ((expand (anocri (length (cdr (assoc 'a x)))
                     (cdr (assoc 'a x)) ))) ) ; [ 0.0 0.1 0.0 ]

(defn s2 ()
    '(if (cri (sub p a))
         (put p (add1 p))
         ( (put q (sub1 q))
           (if (cri (sub q a))
               ( (put a (swap p q a))
                 (put p (add1 p)) ) ) ) ) )

(prove-lemma swap-out-safe-pos (rewrite)
    (implies (and (acri p a)
                  (leq p i) (leq p j)
                  (numberp i)
                  (numberp j)
                  (leq i (length a))
                  (leq j (length a)) )
             (acri p (swap i j a)) ) ) ; [ 0.0 0.2 0.2 ]

(prove-lemma swap-out-safe-neg (rewrite)
    (implies (and (anocri q a)
                  (lessp i q) (lessp j q)
                  (numberp i)
                  (numberp j)
                  (leq q (length a)) )
             (anocri q (swap i j a)) )
    ((induct (anocri q a))) ) ; [ 0.0 1.6 0.5 ]

(prove-lemma subtlety (rewrite)
    (implies (and (lessp p q)
                  (numberp p)
                  (lessp (sub1 (sub1 q)) p) )
             (equal (sub1 q) p) ) ) ; [ 0.0 0.0 0.0 ]


(prove-lemma invariant-j2-s2 (rewrite)
    (evalimplies `(and ,(b2) ,(j2))
                 (wp0 (s2) (j2))
                 x )
    ((do-not-induct t)) ) ; [ 0.0 1.5 0.4 ]

(defn vf2 () '(difference q p))
```

```
(prove-lemma vf2-decreases (rewrite)
    (evalimplies `(and (equal ,(vf2) v) (and ,(b2) ,(j2)))
                 (wp0 (s2) `(lessp ,(vf2) v))
                 x ) ) ; [ 0.0 0.7 0.4 ]

(prove-lemma notoccurs2-v (rewrite)
    (and (notoccurs 'list 'v (list (j2) (b2) (vf2)))
         (notoccursp 'v (s2)) ) ) ; [ 0.0 0.0 0.0 ]

(defn prog2 () `(,(init2) (while ,(b2) ,(s2))))

(prove-lemma honesty ()
    (and (notoccursp 'w (prog2))
         (notoccursp 'g (prog2)) ) ) ; [ 0.0 0.0 0.0 ]

(disable-theory (init2 *1*init2 j2 *1*j2 s2 *1*s2
         b2 *1*b2 p2 *1*p2 vf2 *1*vf2 q2 *1*q2 ))

(prove-lemma dijkstra-prog2 (rewrite)
    (dijkstra (p2) (prog2) (q2)) ) ; [ 0.0 0.1 0.0 ]
```

After the definition of `prog2`, its value can be given by NQTHM's execution environment `r-loop`. Below we give a dialogue with the `r-loop` in which the program is asked, followed by an execution of the program for a given array. The star $*$ is the prompt of the `r-loop`.

```
* (prog2)
 '(((PUT P 0) (PUT Q (LENGTH A)))
   (WHILE (LESSP P Q)
          (IF (CRI (SUB P A))
              (PUT P (ADD1 P))
              ((PUT Q (SUB1 Q))
               (IF (CRI (SUB Q A))
                   ((PUT A (SWAP P Q A))
                    (PUT P (ADD1 P))))))))
* (exe 9 (prog2) '((a . (1 2 3 103 146 3 109 5 118 4))))
 '((A 118 109 146 103 3 3 2 5 1 4)
   (P . 4)
   (Q . 4))
```