Transforming Programs into Recursive Functions

Magnus O. Myreen, Michael J. C. Gordon[1]

Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge, UK

Abstract

This paper presents a new proof-assistant based approach to program verification: programs are translated, via fully-automatic deduction, into tail-recursive function defined in the logic of a theorem prover. This approach improves on well-established methods based on Hoare logic and verification condition generation (VCG) by removing the need to annotate programs with assertions, making the proof natural to the theorem prover and being easier to implement than a trusted VCG. Our tool has been implemented in the HOL4 theorem prover.

Keywords: program verification, theorem proving.

1 Introduction

Program verification is commonly done by reasoning in programming logics or by generating verification conditions. These approaches serve as interfaces from programs to 'mathematics'. Programming logics provide an axiomatic formal system for reasoning about programs, whilst verification condition generators (VCGs) extract sufficient conditions, typically as first order formulae, for properties of programs to hold.

In this paper we propose a different approach: translating programs, via fully automatic deduction, into recursive functions defined directly in the native logic of a theorem prover. This approach has a number of benefits for program verification:

(i) makes subsequent proofs natural for the theorem prover (e.g. use of induction);

(ii) reduces the verification problem to its essence (the computation);

(iii) allows reuse of already proved algorithm (from library theories);

(iv) facilitates use of existing automation (e.g. existing decision procedures);

(v) easier to implement than a trustworthy (i.e. proved or proof-producing) VCG;

[1] We thank Larry Paulson for pointing out results by Moore and Manolios [5] which made this work possible.

(vi) provides a back-end that can support verification for different languages.

To give the reader a sense for what these translations are, we illustrate one such translation using an example by McCarthy [7]. McCarthy points out the correspondence between imperative programs and recursive functions by showing that the following program

```
y := 1;
while (0 < x) do
  y := x × y;
  x := x - 1;
end
```

implements the function `f` below[2]. Here `g` corresponds to the `while` loop.

```
f(x,y) = let y = 1 in g(x,y)

g(x,y) = if not (0 < x) then (x,y) else
           let y = x × y in
           let x = x - 1 in
             g(x,y)
```

Our contribution is a technique that uses a theorem prover to perform this translation fully automatically by mechanised deduction from a formal semantics of the programming language. The techique automatically creates a proof certifying that the recursive functions represents the semantics of the program. More specifically, this paper:

(i) shows how a translation into recursive functions can be constructed by instantiations of proof rules for both partial and total correctness (Sections 5 and 6)

(ii) states termination conditions in a way which makes them naturally a consequence of any proof by induction used for verification (Sections 6.1 and 7);

(iii) illustrates advantages of proof by induction over established methods based on verification condition generation (Sections 7 and 8).

We have implemented[3] our proof-generating translation method in the HOL4 theorem prover [12]. In order to show that nothing was simplified in the translation to printed form, we keep the notation in this paper close to HOL4's syntax (only special characters `!`, `?` etc. are translated to standard notation ∀, ∃ etc.).

2 Semantics of a simple language

Consider programs constructed from the following. Here `i` stands for integers and `v` for strings.

```
e ::= Const i | Var v | Plus e e | Sub e e | ...
```

[2] McCarthy used a different notation for both the program and the function.

[3] Our implementation is available at http://hol.sf.net/ in SVN under /HOL/examples/opsemTools/extract.

```
      b ::= Not b | Equal e e | Less e e | ...
program ::= Skip | Assign v e | Seq program program
          | Cond b program program | While b program
```

In HOL, the program above is encoded as:

```
Seq (Assign "y" (Const 1))
    (While (Less (Const 0) (Var "x"))
      (Seq (Assign "y" (Times (Var "x") (Var "y")))
           (Assign "x" (Sub (Var "x") (Const 1)))))
```

We build on work by Camilleri and Melham [1], who define in HOL a big-step operational semantics for such programs. In this presentation we consider states that are (finite) partial functions from variable names (of type `:string`) to integers (type `:int`). Let `neval` and `beval` define evaluations over expressions.

```
neval (Const k) state = k
neval (Var v) state = state(v)
neval (Plus e1 e2) state = neval e1 state + neval e2 state
...

beval (Not b) state = ¬(beval b state)
beval (Equal e1 e2) state = (neval e1 state = neval e2 state)
...
```

The semantics of programs is defined by `EVAL program s1 s2`, which relates `s1` to `s2` if and only if state `s1` can be transformed into state `s2` by an execution of `program`. Here `s[v ↦ x]` updates state `s` so that it maps `v` to `x`. `EVAL` is the least relation satisfying the following:

```
EVAL Skip s s

EVAL (Assign v e) s (s[v ↦ neval e s])

EVAL c1 s1 s2 ∧ EVAL c2 s2 s3 ⇒ EVAL (Seq c1 c2) s1 s3

EVAL c1 s1 s2 ∧ beval b s1 ⇒ EVAL (Cond b c1 c2) s1 s2

EVAL c2 s1 s2 ∧ ¬beval b s1 ⇒ EVAL (Cond b c1 c2) s1 s2

¬beval b s ⇒ EVAL (While b c) s s

EVAL c s1 s2 ∧ EVAL (While b c) s2 s3 ∧ beval b s1
⇒ EVAL (While b c) s1 s3
```

3 Hoare logic

We will express specifications using Hoare triples [3]. A standard *partial-correctness* Hoare triple, $\{p\}\,c\,\{q\}$, is written here as `SPEC p c q` and given the semantics defined by:

```
SPEC p c q = ∀s1 s2. p s1 ∧ EVAL c s1 s2 ⇒ q s2
```

Informally, this reads as: if precondition `p` is satisfied by some current state `s1` then postcondition `q` holds for any state `s2` reachable by an execution of program `c`.

Notice that `SPEC` does not guarantee that there exists any reachable final state `s2`, i.e. `SPEC` does not ensure termination. The following *total-correctness* Hoare triple requires termination for any state `s1` which satisfies precondition `p`:

```
TOTAL_SPEC p c q = SPEC p c q ∧
                   ∀s1. p s1 ⇒ ∃s2. EVAL c s1 s2
```

4 A thin layer of separation logic

We will use a separating conjunction (borrowed from separation logic [11]) to aid automation and add expressiveness to our Hoare triples. The separating conjunction `p * q` is true for state `s`, if `s` can be split into two disjoint states `s1`, `s2` such that `p` holds for `s1` and `q` holds for `s2`:

```
SPLIT s (s1,s2) = (domain s1 ∪ domain s2 = domain s) ∧
                  (domain s1 ∩ domain s2 = {}) ∧
                  (∀v. v ∈ domain s ⇒
                       s(v) = if v ∈ domain s1
                              then s1(v) else s2(v))

(p * q) s = ∃s1 s2. SPLIT s (s1,s2) ∧ p s1 ∧ q s2
```

The `*`-operator is both associative and commutative.

We define new versions of Hoare triples using the separating conjunction:

```
    SEP_SPEC p c q = ∀r. SPEC (p * r) c (q * r)

SEP_TOTAL_SPEC p c q = ∀r. TOTAL_SPEC (p * r) c (q * r)
```

These definitions incorporate the semantic idea underlying the frame rule of separation logic, but differ because, in separation logic, the `*`-operator is used to separate heap assertions but not stack assertions. In contrast, here the separating conjunction is used to separate all resources; this bears some similarity to the idea of "Variable as Resource" [10].

The remaining sections discuss proof rules that can be derived from the definitions of `SEP_SPEC` and `SEP_TOTAL_SPEC`. One such rule, the frame rule, allows any assertion `f` to be added (unconditionally) to specifications:

```
∀f. SEP_SPEC p c q ⇒ SEP_SPEC (p * f) c (q * f)

∀f. SEP_TOTAL_SPEC p c q ⇒ SEP_TOTAL_SPEC (p * f) c (q * f)
```

An example will illustrate this. Our specifications use `VAR` (defined below) to assert the value of a variable, i.e. `VAR v x` states that variable `v` has value `x`:

```
(VAR v x) s = (domain s = {v}) ∧ (s(v) = x)
```

Now the assignment "`b := a + b`" has the following specification. Here the precondition assumes that variable `"a"` has initial value `a` and that `"b"` has value `b`. The postcondition states that the assignment updates variable `"b"` to `a + b`.

```
SEP_SPEC (VAR "a" a * VAR "b" b)
         (Assign "b" (Plus (Var "a") (Var "b")))
         (VAR "a" a * VAR "b" (a + b))
```

The frame rule allows us to infer that resources not mentioned in the specification are left untouched, e.g. we can instantiate `f` in the frame rule with `VAR "c" c` and, thus have:

```
SEP_SPEC (VAR "a" a * VAR "b" b * VAR "c" c)
         (Assign "b" (Plus (Var "a") (Var "b")))
         (VAR "a" a * VAR "b" (a + b) * VAR "c" c)
```

Instantiating the frame rule with `VAR "a" x` would give a false precondition, and thus a vacuously true statement. An example of what a `SEP_SPEC` specification means in terms of `EVAL` is given in Section 7.1.

5 Constructing specifications for partial-correctness

This section presents proof rules for `SEP_SPEC`, that can be derived from its definition, and shows how the rule can be used to automatically derive functions describing the effect of programs.

5.1 Assignments

Assignments update the value of a variable and can always be implemented by a let-expression, e.g. the effect of `Assign "b" (Plus (Var "a") (Var "b"))` is captured by `let b = a + b in (a,b)`. We express this fact by the following theorem:

```
SEP_SPEC (VAR "a" a * VAR "b" b)
         (Assign "b" (Plus (Var "a") (Var "b")))
         (let b = a + b in (VAR "a" a * VAR "b" b))
```

5.2 Sequential composition

Sequential composition is introduced using the following proof rule:

```
SEP_SPEC p c1 m ∧ SEP_SPEC m c2 q  ⇒
SEP_SPEC p (Seq c1 c2) q
```

Given specifications for two programs, say, `program1` and `program2`,

```
SEP_SPEC (VAR "a" a * VAR "b" b * VAR "c" c)
         program1
         (let (b,c) = f1(a,b,c) in
           (VAR "a" a * VAR "b" b * VAR "c" c))

SEP_SPEC (VAR "a" a * VAR "b" b)
         program2
         (let b = f2(a,b) in
           (VAR "a" a * VAR "b" b))
```

we can compose the two by first using the frame rule to insert `VAR "c" c` into the second specification and then instantiating the second specification with the let-update from the first specification, followed by an application of composition:

```
SEP_SPEC (VAR "a" a * VAR "b" b * VAR "c" c)
         (Seq program1 program2)
         (let (b,c) = f1(a,b,c) in
          let b = f2(a,b) in
            (VAR "a" a * VAR "b" b * VAR "c" c))
```

5.3 Conditional statements

Conditional execution is based on a guard, e.g. `(Less (Var "a") (Const 5))`. Let `SEP_GUARD p g b` require that function `g` is equivalent to the guard `b`, if the resource assertion `p` holds:

```
SEP_GUARD p g b = ∀r s x. (p x * r) s ⇒ (g x = beval b s)
```

As an example, `(λa. a < 5)` is related to `(Less (Var "a") (Const 5))`:

```
SEP_GUARD (λa. VAR "a" a) (λa. a < 5) (Less (Var "a") (Const 5))
```

Functions describing conditionals are constructed using the following rule:

```
SEP_GUARD p g b ⇒
SEP_SPEC (p x) c1 (p y) ⇒
SEP_SPEC (p x) c2 (p z) ⇒
SEP_SPEC (p x) (Cond b c1 c2) (p (if g x then y else z))
```

As an example, we have:

```
SEP_SPEC (VAR "a" a)
         (Cond (Less (Var "a") (Const 5))
               (Assign "a" (Const 1))
               (Assign "a" (Plus (Var "a") (Const 1))))
         (let a = if a < 5
                  then (let a = 1 in a)
                  else (let a = a + 1 in a) in (VAR "a" a))
```

5.4 While loops

A recursive function is needed for describing the behaviour of `While b c`. For this purpose, we define a tail-recursive function `WHILE` as follows:

```
WHILE g f x = if g x then WHILE g f (f x) else x
```

Moore and Manolios showed that such a function can be defined in logic without a termination proof [5]. `WHILE` corresponds to the effect of a while loop, as can be seen from the following loop rule:

```
SEP_GUARD p g b ⇒
(∀x. g x ⇒ SEP_SPEC (p x) c (p (f x))) ⇒
(∀x. SEP_SPEC (p x) (While b c) (p (WHILE g f x)))
```

Before presenting a sketch of the proof of this rule, we illustrate its use by an example. Suppose `c = Assign "a" (Plus (Var "a") (Const 1))` and `b` is the guard `(Less (Var "a") (Const 5))`, then

```
SEP_SPEC ((λa. VAR "a") a)
         (Assign "a" (Plus (Var "a") (Const 1))
         ((λa. VAR "a") ((λa. let a = a + 1 in a) a))
```

fits the loop rule, and if `add_loop` is defined by

```
add_loop = WHILE (λa. a < 5) (λa. let a = a + 1 in a)
```

then the loop rule produces:

```
SEP_SPEC (VAR "a" a)
         (While (Less (Var "a") (Const 5))
                (Assign "a" (Plus (Var "a") (Const 1))))
         (let a = add_loop a in VAR "a" a)
```

An equation for `add_loop` can be proved by an unfolding of `WHILE`:

```
   add_loop a
 = WHILE (λa. a < 5) (λa. let a = a + 1 in a) a
 = if (λa. a < 5) a then
     WHILE (λa. a < 5) (λa. let a = a + 1 in a)
```

```
        ((λa. let a = a + 1 in a) a) else a
    = if a < 5 then add_loop (let a = a + 1 in a) else a
    = if a < 5 then let a = a + 1 in add_loop a else a
```

5.5 Proof of loop rule

The proof of the loop rule is based on the case split on whether one can reach a state where the guard for `WHILE` is made false by repeatedly executing the body of the loop. Let `FUNPOW f n x` be the value of `n` applications of `f` to `x`:

```
FUNPOW f 0 x = x
FUNPOW f (n + 1) x = FUNPOW f n (f x)
```

The proof is based on a case split on:

```
∃n. ¬g (FUNPOW f n x)
```

If there exists such an `n` then we unfold the loop `n`-times to get the desired result. If there is no such `n` then the loop does not terminate, making the partial-correctness Hoare triple `SEP_SPEC` vacuously true.

5.6 McCarthy's Example

The theorem stating the correspondence between the function and the program in McCarthy's example (from Section 1, where `f` is defined) is the following:

```
SEP_SPEC (VAR "x" x * VAR "y" y)
         (Seq (Assign "y" (Const 1))
              (While (Less (Const 0) (Var "x"))
                 (Seq (Assign "y" (Times (Var "x") (Var "y")))
                      (Assign "x" (Sub (Var "x") (Const 1))))))
         (let (x,y) = f(x,y) in (VAR "x" x * VAR "y" y))
```

6 Constructing specifications for total-correctness

Most of the proof rules and hence most of the derivation of the executed function is exactly the same for total-correctness. The required change is that we need to assume termination for loops, i.e. for a loop described by `WHILE guard f x` we need to assume that some number of applications of `f` to `x` will eventually turn `guard` to false:

```
∃n. ¬guard (FUNPOW f1 n x)
```

This necessary assumption in the loop rule introduces a precondition which needs to be threaded through the entire development.

6.1 *While loops*

For the loop rule we define the following format for its precondition. Here `PRE` states that the loop described by `WHILE guard f x` terminates and that each execution of the loop body can assume the side-condition `q`.

```
PRE f guard q x =
  (∃n. ¬guard (FUNPOW f n x)) ∧
  (∀k. (∀m. m ≤ k ⇒ guard (FUNPOW f m x)) ⇒ q (FUNPOW f k x))
```

The following theorem states that `PRE` can be unfolded like a recursive function:

```
PRE f guard q x = (guard x ⇒ q x ∧ PRE f guard q (f x))
```

This unfolding is particularly useful in induction proofs where the base case makes `guard` false and the step case unfolds `PRE` once, i.e. this unfolding makes the termination condition follow from any induction used for verification of `WHILE`.

The following induction principle is proved from the definition of `PRE`.

```
∀P. (∀x. guard x ∧ side x ∧ P (f x) ⇒ P x) ∧
    (∀x. ¬guard x ⇒ P x) ⇒
    (∀x. PRE f guard side x ⇒ P x)
```

The induction is used in proving a rule for `While`:

```
SEP_GUARD p guard b ⇒
(∀x. guard x ∧ side x ⇒ SEP_TOTAL_SPEC (p x) c (p (f x))) ⇒
(∀x. PRE f guard side x ⇒
     SEP_TOTAL_SPEC (p x) (While b c) (p (WHILE guard f x)))
```

This rule is proved from the induction arising from `PRE` with `P` instantiated to:

```
λx. SEP_SPEC (p x) (While b c) (p (WHILE guard f x))
```

The total correctness specification for the example used for illustrating the partial-correctness will have a precondition:

```
add_loop_pre = PRE (λa. let a = a + 1 in a) (λa. a < 5) (λa. T)
```

The `While`-rule gives the following:

```
add_loop_pre a ⇒
SEP_TOTAL_SPEC (VAR "a" a)
               (While (Less (Var "a") (Const 5))
                      (Assign "a" (Plus (Var "a") (Const 1)))
               (let a = add_loop a in VAR "a" a)
```

The precondition, i.e. the termination condition, can be stated as the following rewrite, which can be unfolded until the guard `a < 5` becomes false.

```
add_loop_pre a = (a < 5 ⇒ add_loop_pre (let a = a + 1 in a))
```

For this case it is easy to prove ∀a. add_loop_pre a.

6.2 Propagating termination conditions

For total-correctness, special termination conditions need to be passed through the rules for composing specifications. The loop rule allows accumulation of side-conditions in PRE. For sequential composition the side conditions are composed, e.g.

```
side1 x ⇒ SEP_TOTAL_SPEC (p x) c1 (let x = f1 x in p x)

side2 x ⇒ SEP_TOTAL_SPEC (p x) c2 (let x = f2 x in p x)
```

produces the following, using the rule for sequential composition.

```
side1 x ∧ (let x = f1 x in side2 x) ⇒
SEP_TOTAL_SPEC (p x)
               (Seq c1 c2)
               (let x = f1 x in (let x = f2 x in p x))
```

The rule for conditional statements introduces an if into the side condition:

```
(if h x then side1 x else side2 x) ⇒
SEP_TOTAL_SPEC (p x)
               (Cond g c1 c2)
               (let x = (if h x then f1 x else f2 x) in p x)
```

Assignments stay the same:

```
∀p v e q. SEP_TOTAL_SPEC p (Assign v e) q =
                SEP_SPEC p (Assign v e) q
```

6.3 McCarthy's Example

The following is the accumulated precondition for the initial example program, when a total-correctness specification is derived.

```
f_pre(x,y) = let y = 1 in g_pre(x,y)

g_pre(x,y) = (0 < x ⇒ let y = x * y in
                      let x = x - 1 in
                        g_pre(x,y))
```

7 Proving conditional termination

Consider the following program, which stores `n DIV 2` in `a` if `n` is initially non-negative and even. The program fails to terminate if `n` is odd.

```
a := 0;
while (n ≠ 0) do
  a := a + 1;
  n := n - 2
end
```

This is an example used by Moore to illustrate his approach to verification condition generation [8]. We illustrate our approach on the same example to ease a comparison of the two approaches (next section).

When the above code is translated into recursive functions, function `d` is generated:

```
d(a,n) = let a = 0 in d1(a,n)

d1(a,n) = if n = 0 then (a,n) else
            let a = a + 1 in
            let n = n - 2 in
              d1(a,n)
```

and precondition `d_pre` specifies a sufficient condition for termination:

```
d_pre(a,n) = let a = 0 in d1_pre(a,n)

d1_pre(a,n) = ¬(n = 0) ⇒ (let a = a + 1 in
                           let n = n - 2 in
                             d1_pre(a,n))
```

We prove, by a straight-forward induction on `n` (a 4-line HOL4 proof), the following property of the loop `d1` and its precondition `d1_pre`:

```
∀n a. 0 ≤ n ⇒ d1_pre(a,2×n) ∧ (d1(a,2×n) = (n + a, 0))
```

from which it is easy (3 lines of HOL4) to prove termination and functional correctness of the translated function `d`, i.e.

```
∀n a. 0 ≤ n ∧ EVEN n ⇒ d_pre(a,n) ∧ (d(a,n) = (n DIV 2, 0))
```

Let this theorem be called `d_spec`.

7.1 Relating the verification proof to the automatically derived theorem

The following theorem was derived when the original program (which we will refer to as `d_program`) was translated into recursive functions `d` and `d_pre`.

```
d_pre(a,n) ⇒
SEP_TOTAL_SPEC (VAR "a" a * VAR "n" n)
               d_program
               (let (a,n) = d(a,n) in (VAR "a" a * VAR "n" n))
```

In order to justify that `d_spec` is a verification proof of `d_program`, note that `d_spec` implies:

```
0 ≤ n ∧ EVEN n ⇒
SEP_TOTAL_SPEC (VAR "a" a * VAR "n" n)
               d_program
               (VAR "a" (n DIV 2) * VAR "n" 0)
```

which by expansion of various definitions is equivalent to the following statement:

```
0 ≤ s1("n") ∧ EVEN s1("n") ∧ {"a","n"} ⊆ domain s1 ⇒
(∃s2. EVAL s1 s2 d_program) ∧
(∀s2. EVAL s1 s2 d_program ⇒
        s2 = s1["a" ↦ (s1("n") DIV 2)]["n" ↦ 0])
```

Thus, the user need only prove `d_spec` in order to imply termination and functional correctness of `d_program`.

8 Comparing different approaches

The previous section presented an example verification using our method of "translation into recursive functions". This section compares the above proof with well established techniques based on verification condition generators (VCGs) and reasoning directly using a Hoare logic.

8.1 Using a verification condition generator

To verify a program using a VCG, one starts by annotating the code with assertions. For the program above (`d_program`) one needs to invent a precondition, a postcondition and, for the loop, an invariant and a variant (the result is to be a total-correctness specification). Here `n` is logical variable (the initial values of variable `"n"`).

```
pre n = λs. (s("n") = n) ∧ EVEN n ∧ 0 ≤ n

post n = λs. (s("a") = n DIV 2) ∧ (s("n") = 0)

inv n = λs. (n = 2×s("a")+s("n")) ∧ EVEN s("n") ∧ 0 ≤ s("n")

variant = λs. s("n")
```

The program with the annotations:

```
{ pre n }
a := 0;
while (n ≠ 0) do  { inv n } [ variant ]
  a := a + 1;
  n := n - 2
end
{ post n }
```

A VCG produces the following verification conditions:

```
∀n s. pre n (s["a" ↦ 0]) ⇒ inv n s ∧ 0 ≤ variant s

∀n s. let s' = s["n" ↦ s("n")-2]["a" ↦ s("a")+1] in
        (inv n s ∧ (s("n") ≠ 0)
         ⇒ inv n s' ∧ variant s' < variant s)

∀n s. inv n s ∧ (s("n") = 0) ⇒ post n s

∀n s. inv n s ⇒ 0 ≤ variant s
```

If these verification conditions are proved, by the user in some way, then this total-correctness theorem follows:

```
∀n. TOTAL_SPEC (pre n) d_program (post n)
```

The definitions `pre`, `post`, `inv` and `variant` were defined to reflect intuition. However, the resulting specification is much weaker than the specification proved using our technique (previous section), since the above specification does not say anything about variables other than `"a"` and `"n"`. It is obvious that other variables, say `"b"` or `"m"`, are left untouched by the program, but the VCG proof did not prove it. Other differences that are worth noting are:

(i) the VCG proof requires more user input: stating the assertions for the VCG proof requires more writing than the proof goals stated in the previous section;

(ii) the VCG proof requires the user to invent an invariant expression

```
n = 2 × s("a") + s("n")
```

describing the relationship between the intermediate values and the initial value `n`, while the induction proof, from the previous section, only required stating the desired result of executing the remaining part of the loop:

```
d1(a, 2 × n) = (n + a, 0)
```

(iii) the VCG proof uses a variant where the previous section uses an induction;

(iv) the VCG proof deals directly with the state `s`, while the verification proof from the previous section only concerns logical variables (and is, as a result, reusable for similar code based on a different definitions of ‘state’).

Our example program d_program intentionally coincides with the example Moore uses to illustrate his approach to verification using a VCG [8]. Moore suggests that the user defines a function halfa (which is basically equivalent to our definition of d, Section 7) and that the user writes annotations which state that halfa is executed by the program he aims to verify. He proves using a VCG that his program executes halfa. In our approach 'halfa' is derived completely automatically and then we went further to prove a non-recursive specification about the original program.

8.2 Using a Hoare logic directly

The VCG method automates Hoare logic. In this section we demonstrate how d_program, from Section 7, can be proved manually using Hoare logic, given definitions pre, post, inv and variant from above; and assuming that the verification conditions from above have been proved.

The manual Hoare logic proof will make use of the following five proof rules: assignment, precondition strengthening, postcondition weakening, sequence and while.

```
TOTAL_SPEC (λs. p (s[v ↦ (neval e s)]) (Assign v e) p

(∀s. p' s ==> p s) ∧ TOTAL_SPEC p c q ⇒ TOTAL_SPEC p' c q

(∀s. q s ==> q' s) ∧ TOTAL_SPEC p c q ⇒ TOTAL_SPEC p c q'

TOTAL_SPEC p c1 q ∧ TOTAL_SPEC q c2 r
⇒ TOTAL_SPEC p (Seq c1 c2) r

(∀s. i s ⇒ 0 ≤ f(s)) ∧
(∀v. TOTAL_SPEC (λs. i s ∧ beval b s ∧ (f(s) = v)) c
                (λs. i s ∧ (f(s) < v)))
⇒ TOTAL_SPEC i (While b c) (λs. i s ∧ ¬(beval b s))
```

Appropriate instantiations of the assignment rule and sequence rule give:

```
TOTAL_SPEC (λs. inv n (s["a" ↦ 0])) (Assign "a" (Const 0)) (inv n)

TOTAL_SPEC (λs. let s' = s["n" ↦ s("n")-2]["a" ↦ s("a")+1] in
                  (inv n s' ∧ (variant s' = v)))
               (Seq (Assign "a" (Plus (Var "a") (Const 1)))
                    (Assign "n" (Sub (Var "n") (Const 2))))
             (λs. inv n s ∧ (variant s = v))
```

Assuming that the second verification condition from above has been proved, the while rule can be applied to the above theorem for the following:

```
TOTAL_SPEC (inv n)
      (While (Not (Equal (Var "n") (Const 0)))
         (Seq (Assign "a" (Plus (Var "a") (Const 1)))
              (Assign "n" (Sub (Var "n") (Const 2)))))
      (λs. inv n s ∧ (s("n") = 0))
```

which by application of the sequence rule followed by precondition strengthening and postcondition weakening proves:

```
  TOTAL_SPEC (pre n) d_program (post n)
```

9 Discussion of related work

The previous section compared our approach with approaches based on direct reasoning using a Hoare logic [3] and an approach based on a VCG. Hoare logics are reasonably easily implemented in a theorem prover, but tend to be labour intensive to use manually inside a theorem prover. Trustworthy verification condition generators are on the other hand easier to use but harder to implement [4,8,6]. The method presented here requires less user input, provides stronger specifications and is readily implementable. The automatic translator has been implemented as a 400-line ML program.

Representing imperative programs as recursive functions, due to McCarthy [7], was key in this work. Ideas from separation logic [11] were used to aid automation (by keeping specifications free of side conditions); and ideas from decompilation into HOL were used for dealing with loops [9]. Our programming environment is the HOL4 theorem prover [12].

The net effect of the presented translation into functional programs bears some resemblance to automation developed by Filliâtre for verification of imperative programs using the Coq proof assistant [2]. However, Filliâtre's approach requires the user to annotate the program with invariants before the translation can be performed. In contrast, the approach presented here is fully automatic and requires no annotations of the original program.

References

[1] Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, August 1992.

[2] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program.*, 13(4):709–745, 2003.

[3] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[4] Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *Comput. J.*, 38(2):131–141, 1995.

[5] Panagiotis Manolios and J. Strother Moore. Partial functions in ACL2. *J. Autom. Reasoning*, 31(2):107–127, 2003.

[6] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In *LPAR*, volume 4246 of *LNCS*, pages 362–376. Springer, 2006.

[7] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[8] J. Strother Moore. Inductive assertions and operational semantics. In *CHARME*, volume 2860 of *LNCS*, pages 289–303. Springer, 2003.

[9] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures – An application of decompilation into logic. In *FMCAD*, 2008.

[10] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In *LICS*, pages 137–146. IEEE Computer Society, 2006.

[11] John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE Computer Society, 2002.

[12] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *TPHOLs*, LNCS. Springer, 2008.