Reasoning about Pointer Structures in Higher Order Logic

A. R. Martini

Abstract. Pointers have been a persistent trouble area in program proving. Some of the fundamental issues are related to the problem of aliasing, support for local reasoning and the technical challenge of constructing formal proofs with these structures. The purpose of this work is to provide an accessible exposition of the several ways that one can conduct, explore and write correctness proofs of pointer programs with Hoare Logic and the Isabelle proof assistant.

Keywords. Hoare Logic, Pointer Programs, Interactive Theorem Proving, Higher Order Logic, Teaching Formal Methods.

Contents

1. Introduction

Pointers have been a persistent trouble area in program proving. The first difficulty is the problem of aliasing, i.e., the situation where the same memory location can be accessed using different names. The second difficulty is the local reasoning problem, i.e., we need a program logic where reasoning about a local area of storage should not affect assertions that describe other regions of the memory. The third is the technical difficulty related to writing proofs with these structures: we have to reason formally about a number of mathematical data types, like sets, sequences, trees and graphs [2, 6]. However, if we wish to prove properties of the kind of programs that are actually used (e.g., low level and object oriented programs), it is essential to reason about programs and algorithms with pointers [2].

In 1972, Burstall [5] gave correctness proofs for imperative programs that change data structures, by using a novel kind of assertion that he called a distinct non-repeating list system (DNRL), in the simplest case. Burstall's DNRL was a sequence of assertions, where each one described a distinct region of storage, so that an assignment to a single location could change only one of them .

The purpose of this report is to provide an accessible exposition of the several ways that one can conduct, explore and write correctness proofs of pointer programs with Hoare Logic and the Isabelle proof assistant. It is aimed at programmers and newcomers interested in theorem proving of pointer programs, who are neither experienced with Isabelle nor with interactive theorem proving as well. We also emphasize structured and high-level reasoning with the proof language *Isar*. By means of several examples, we try to convey to the reader how to tackle the problems of aliasing, local reasoning and the complexity of associated logical deductions.

In Isabelle's theory for Hoare Logic we consider in this text[1], a linked list is modeled by a total function that maps each address to a reference object, which can be null or a pointer to another address. There are two fundamental abstractions with which we can reason about pointer structures: an acyclic list of addresses that links a pointer `p` to `Null` and an acyclic, distinct path of addresses, that connects two pointers `p` and `q`. The essential paper on proving pointer programs with Isabelle is [13]. Isabelle's implementation provides a general purpose logic, actually Higher Order Logic augmented with specific types and functions, to reason about pointer structures with Hoare Logic. Moreover, the user is provided with a concrete syntax for the specification of Hoare triples, a verification condition generator, pointer notation in the style of Pascal, and a rich set of proof tactics and tools for Isabelle/HOL. Most importantly, users can express their reasoning in a formal proof language called *Isar*, that supports readable, structured and detailed proofs in natural deduction style.

A modern approach to verification of pointer programs is based on *Separation Logic* [15]. This is a more complex and sophisticated logic that is available in Isabelle on top of Imperative/HOL [10, 4], a extension of Higher Order Logic that supports Haskell's imperative specification style based on monads. Separation Logic and Isabelle's facilities for reasoning with it are outside the scope of this report.

---

[1] `http://isabelle.in.tum.de/dist/library/HOL/HOL-Hoare/index.html.`

The text is organized as follows: in section 2 we describe some basic concepts for describing and conducting proofs of imperative programs with Hoare logic in Isabelle/HOL. Section 3 describes and illustrates the basic model for references and linked lists. In section 4 we describe basic abstraction predicates that model linked lists by means of lists of addresses together with some fundamental properties needed for reasoning about such structures. Section 5 discusses some simple examples of pointer programs together with their corresponding proofs. In sections 6, 7 and 8 we present the main cases studies: deletion at the end of a linked list, in-place list reversal and cyclic list reversal. Finally, in section 9 we summarize the main ideas discussed in this report.

2. Hoare Logic in Isabelle/HOL

The purpose of this section is to introduce the basic concepts of Isabelle/HOL needed to read the paper and to present the essential ideas to use the Isabelle proof assistant to conduct proofs in Hoare Logic. We assume the implementation described in the library `HOL-Hoare`[2]. Our discussion assumes acquaintance with formal proofs in Hoare Logic and the concept of verification conditions associated with annotated Hoare triples [7, 11].

Using Floyd/Hoare logic [16, 8], we can prove that a program is correct by applying a finite set of inference rules to an initial program specification of the form $\{P\}\ c\ \{Q\}$ such that P and Q are logical assertions, and c is a imperative program or program fragment. The intuition behind such a specification, widely known as Hoare triple or as partial correctness assertion (PCA), is that if the program c starts executing in a state where the assertion P is true, then if c terminates, it does so in a state where the assertion Q holds.

Isabelle is a generic meta-logical framework for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, which stands for *Higher Order Logic* [14]. HOL can be understood by the equation $\mathsf{HOL} = \mathsf{Functional\ Programming} + \mathsf{Logic}$. Thus, most of the syntax of HOL will be familiar to anybody with some background in functional programming and logic. We just highlight the essential notation. The space of total functions is denoted by the infix $\Rightarrow$. Other type constructors, e.g., list, set, are written postfix, i.e., follow their argument as in `'a set`, where `'a` is a type variable. Lists in HOL are of type `'a list` and are built up from the empty list `[]` and the infix constructor # for adding an element at the front. In the case of non-empty lists, functions `hd` and `tl` return the first element and the rest of the list, respectively. Two lists are appended with the infix operator @. Function `rev` reverses a list. In HOL, types and terms must be enclosed in double quotes.

The HOL-Hoare theory is an implementation of Hoare logic for a simple imperative language with assignments, null command, conditional, sequence

[2] https://isabelle.in.tum.de/dist/library/HOL/HOL-Hoare/

and while loops. Each while loop must be annotated with an invariant. Hoare triples can be stated like goals of the form `VARS x y`... {`P`} `prog` {`Q`}, where `prog` is a program in the language, `P` is the precondition, `Q` the postcondition. These assertions can be any formula in HOL, which are written in standard logical syntax. The prefix `x y` ... is the list of all program variables in `prog`. The latter list must be nonempty and it must include all variables that occur on the left-hand side of an assignment in `prog`.

The implementation hides reasoning in Hoare logic completely and provides a method (`vcg`) for transforming a goal in Hoare logic into an equivalent list of verification conditions in HOL. The implementation is a logic of partial correctness. You can only prove that your program does the right thing if it terminates, but not that it terminates.

3. References and Linked Lists

We discuss briefly the aliasing problem with arrays, and how the trick used to handle it allow us to deal with pointer aliasing using the traditional assignment axiom of Hoare proof calculus. In the sequel, we describe how linked structures are modeled with the facilities supported by Isabelle/HOL implementation of Hoare Logic.

3.1. The Aliasing Problem in Arrays

The following discussion might be a little off-topic, but it is essential to understand the underlying technique with which we can reason about pointers using traditional Hoare logic. The key idea was formulated to solve the problem of aliasing with arrays. The trick is to consider arrays as functions from indexes (addresses) to values.

One of the most fundamental technical issues that must be dealt with when reasoning about pointers is the problem of aliasing, i.e, when two or more program variables refer to the same location. The Hoare axiom for assignment [8] is surprisingly simple. The beauty of it is that it replaces complicated questions about memory states with a simple formal calculation, an easy substitution of a formula for a name. However, it only works if different variable names denote distinct memory locations [3]. Even without using pointers, we already encounter the problem of aliasing when using program with array variables. For instance, consider this simple Hoare triple, where a is an array variable:

$$\{i = j \wedge a[i] = 3\}\ a[i] := 4\ \{a[j] = 4\}$$

This triple is valid, because both $a[i]$ and $a[j]$ represent the same location. Using the axiom of assignment, we compute the triple $\{a[j] = 4\}\ a[i] = 4\ \{a[j] = 4\}$. Using the rule for precondition strengthening, we would have to prove the invalid consequence $i = j \wedge a[i] = 3 \rightarrow a[j] = 4$, since an array element cannot have two distinct values. Substitution does not work unless you can tell what is an alias and what is not. Array-element aliasing is a consequence of address arithmetic. The expressions $a[i]$ and $a[j]$ are aliases — i.e. they refer to the same array element, the same memory location. The classical solution is to consider an array as a single variable, instead of a collection of separate variables [3, 7]. We can treat an array as a function from natural numbers (or integers) to a domain of values. Then an assignment to an array can be modeled as a function update. Isabelle provides the notation $f(a := v)$ for updating function f at argument a with the new value v. Simplification is easily computed according to the equation $(\texttt{f}(\texttt{x} := \texttt{y}))\ \texttt{z} = (\texttt{if z} = \texttt{x then y else f z})$. Thus, the Hoare triple above can be coded in Isabelle as follows:

```
lemma "VARS (a::nat ⇒ int)
{i=j ∧ a(i) = 3}
a := a(i:=4)
{a(j) = 4}"
apply (vcg)
apply (simp)
done
```

The sequence of `apply` commands is a proof script. After application of the verification condition generator, the single proof goal is solved automatically by the proof tactic `simp`.

```
proof (prove)
goal (1 subgoal):
 1. ⋀a. i = j ∧ a i = 3 ⟹
         (a(i := 4)) j = 4
```

This trick of treating arrays as function allow us to use traditional Hoare logic and deal successfully with aliasing in arrays. Lifting this idea for fields of records (or classes), we can solve the problem of aliasing for pointers. Thus, we can keep using to the standard Hoare proof calculus for reasoning about data structures defined with references, as shown in the next section.

3.2. Linked Lists in Isabelle/HOL

In languages like C and Pascal programmers create and manipulate linked structures like linked lists and trees through the use of *pointers*. They are special variables that can hold memory addresses denoting locations in the *heap*, i.e., the segment of memory that can be accessed and controlled by programmers. The material in this section is based on in the implementation of the library Hoare-Logic [3] and inspired by the material in [13].

In Isabelle, references are distinguished from addresses, and are declared by the following datatype, where references are polymorphic type constructors, indicated by the type variable `'a`. This means that addresses can be values of any type.

[3] http://isabelle.in.tum.de/dist/library/HOL/HOL-Hoare/index.html.

$$\texttt{datatype}\ {}'\texttt{a ref} = \texttt{Null} \mid \texttt{Ref}\ {}'\texttt{a}$$

A reference is either null or a reference to an address. The terms location and address, respectively pointer and reference, are used interchangeably. Instead of declaring `ref` as a type constructor, we could have used a simple unspecified type `address` as in `Ref address`, but in this way we can give concrete examples of the model. The function $\texttt{addr} :: {}'\texttt{a ref} \Rightarrow {}'\texttt{a}$ unpacks the address from container `Ref`, i.e., $\texttt{addr}\ (\texttt{Ref x}) = \texttt{x}$. A local heap model is modeled as a total function from addresses to values for each field name of a record (or class). Using function update notation, an assignment of value v to field f of a record pointed to by reference r is written $f := f((addr\ r) := v)$, and access of f is written $f(addr\ r)$. Using function updates is essential to deal with the problem of `aliasing`, as we will see below. Based on the syntax of Pascal, the formalization of the Heap syntax provides the following syntactic sugar:

$f(r \to e) = f((addr\ r) := e)$ the value of field f at the address pointed to by r is e
$r\hat{}.f := e = f := f(r \to e)$ the field f of location pointed to by r is assigned the value of e
$r\hat{}.f = f(addr\ r)$ the value of field f at the address pointed to by r

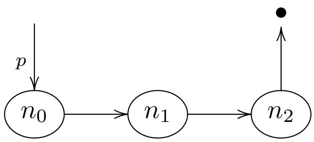


FIGURE 1. Linked List - Students

Linked lists are represented by their `next` field that maps addresses to references, i.e., a heap of type ${}'\texttt{a} \Rightarrow{}' \texttt{a ref}$, where the type variable `'a` is an arbitrary type of addresses. Moreover, an abstraction of a linked list of type `next` is a HOL list of type ${}'\texttt{a list}$. To illustrate how we can model linked lists with this basic model, consider the linked list shown in Figure 1, where we assume that each node has two fields: the `info` field, with information for `name` and `age` and the `next` field, which is a reference to the next node in the chain. For illustration purposes, the following values are given for each node:

node	*name*	*age*	*next*
$n0$	*Anne*	21	$n1$
$n1$	*Paul*	19	$n2$
$n2$	*July*	17	*Null*

Our first model is given in Figure 2, where we model addresses by natural numbers, and each field is defined as a total function from natural numbers to appropriate types. The reference p points to adress 0, the location of the first node. The linked list of addresses is modeled by a function $nat \Rightarrow Ref\ nat$. The lambda expression $\lambda n.\ Null$ initializes all addresses with the null pointer. With function update notation we create the linked list $0 \mapsto 1 \mapsto 2 \mapsto Null$. The field *age* is modeled as a function from addresses to integers, and the field *name* as a function from addresses to strings. The *age* field is initialized with 0 for every address and using function update notation, the appropriate ages are set. Likewise, the field *name* is first defined everywhere with the null string, and then updated with the correct names. Note that he pointer `p` and `tmp` are aliases to location `0`.

```
—‹ example:  0 ↦ 1 ↦ 2  ↦ Null ›
definition next_n::"nat ⇒nat ref" where
   "next_n≡(λn. Null)(0:=Ref 1,1:=Ref(2),2:=Null)"
definition name::"nat ⇒ string"  where
   "name ≡ (λn .'''')(0:=''Anne'',1:=''Paul'',2:=''July'')"
definition age::"nat ⇒ int"  where
   "age ≡ (λn . 0)(0:=19,1:=21,2:=17)"
definition p::"nat ref" where "p≡ Ref 0"
definition tmp::"nat ref" where "tmp≡p"

lemma "p^.name = ''Anne''" by (simp add:p_def name_def)
lemma "p^.next_n^.next_n=Ref 2" by (simp add:p_def next_n_def)
lemma "p^.next_n^.name = ''Paul''"
    by (simp add:p_def name_def next_n_def)
lemma "p^.next_n^.age = 21" by (simp add:p_def next_n_def age_def)
lemma "tmp^.age = 19" unfolding  p_def tmp_def age_def by simp
```

FIGURE 2. Linked List - Students Model

After the basic definitions, we have a series of lemmas that state obvious relations in the model. These propositions are proved with the automatic proof tactic `simp` (from simplifier) which performs higher oder rewriting with equations. Note that in each case we add to the underlying simplifier set the theorems for unfolding of the definitions.We can also code this mode directly as a Hoare triple, as shown bellow. After executing the verification condition generator, the single proof goal obtained is solved automatically by the simplifier. It boils down to compute the value of `age` after a series of function updates.

```
lemma "VARS (next_n::nat ⇒ nat ref)
        (age::nat ⇒ int) (p::nat ref)
        (tmp::nat ref)
{p= Ref 0 ∧ x=Ref 0 ∧ y= Ref 1 ∧ z = Ref 2}
x^.age := 19;x^.next_n := y;
y^.age := 21;y^.next_n := z;
z^.age := 17;z^.next_n := Null;tmp := x
{p^.next_n^.age = 21 ∧ tmp^.age = 19}"
apply (vcg)
apply (simp)
done
```

Using the syntactic sugar introduced earlier and considering the model of Figure 2, the following propositions hold:

lemma next_n 1 = Ref 2
lemma Ref1^.next_n = Ref2
lemma next_n (Ref 2 → Ref 4)) 2 = Ref 4
lemma (next_n (Ref 2 → Ref 4)) 0 = Ref 1
lemma (next_n (2 := Ref 4)) 0 = Ref 1
lemma (next_n 4 = next_n 7) = True
lemma (Ref 4^.next_n = Ref 7^.next_n) = True

4. Relational Abstractions for Heaps

The general approach is to map the implicit chain of addresses represented by the heap field next ::′ a ⇒′ a ref into a list of addresses of type ′a list (see [13] and [9], chapter 8). The predicate List next x as means that as is a list of addresses that connects the reference x to Null by means of the next field. The predicate List is a relational abstraction for acyclic lists and is defined using primitive recursion on the list of addresses.

List :: (′a ⇒′ a ref) ⇒′ a ref ⇒′ a list ⇒ bool
List next r [ ] = (r = Null)
List next r (a#as) = (r = Ref a ∧ List next (next a) as)

Some essential logical consequences of this definition are the following properties:

List next Null as = (as = [ ]) (HNull)
List next (Ref a) as =
(∃ bs. as = a#bs ∧ List next (next a) bs) (HRef)

The equation HNull says that every list of address that starts with the Null pointer is empty. The second equation, HRef, states that if the head of the chain of addresses as is the address a, then the tail of list, bs, is a also a list of addresses that connect the address next to a in the chain to Null. The following rules can be proved by induction on the list of addresses as.

$$\frac{\texttt{List next x as} \quad \texttt{List next x bs}}{\texttt{as} = \texttt{bs}}\ \mathsf{LFun} \qquad \frac{\texttt{List next x (as@bs)}}{\exists \texttt{y}.\ \texttt{List next y bs}}\ \mathsf{LRef}$$

$$\frac{\texttt{List next(next a) x as}}{\texttt{a} \notin \texttt{ set as}}\ \mathsf{LAci} \qquad \frac{\texttt{List next x as}}{\texttt{distinct as}}\ \mathsf{LDist}$$

$$\frac{\texttt{a} \notin \texttt{set as} \quad \texttt{List (next(a := y)) x as}}{\texttt{List next x as}}\ \mathsf{LSep}$$

From rule LFun we know that the relation `List` is functional. The rule LRef states that any suffix of a list is also a list that starts at some address in the original chain and this suffix connects this address to `Null`. Rule LAci states that a list `as` starting with the address that is the successor of `a` does not contain any occurrence of `a`. For, suppose that `a` happens to be in the remaining list `as`. Then `as` can be factored as `bs@(a#cs)` for some lists `bs`, `cs`. Then, by rule LRef we have that `List next (Ref a) (a#cs)` and hence that `List next (next a) cs` by definition of `List`. With the assumption `List next (next a) as` and rule LFun, it follows that `as = cs` and hence, with `as = bs@(a#cs)`, that `as = bs@(a#as)`, which is a contradiction. This discussion shows also why the rule LDist is valid, where `distinct` is a function that returns `True` if and only if all elements in the list are distinct. The last rule, LSep is essential and it denotes an important *separation lemma*. It says that updating the next link of an address that is not part of the linked list denoted by the `next` field does not change the list abstraction. This means that the effect of address updates are local.

5. Simple Pointer Programs

To grasp theses abstractions in practice, we present here a series of simple examples of pointer programs. The simplest one is show in Figure 3 for deletion of the first node of a linked list.

```
lemma "VARS (next::'a ⇒ 'a ref) (p::'a ref) (q::'a ref)
{List next p Ps ∧ p ≠ Null}
  q := p;p := p^.next
{∃ a as. List next p as ∧ q = Ref a ∧ a # as = Ps}"
apply (vcg)
apply (auto)
done
```

FIGURE 3. Deletion of first node

With the precondition, we require that the list of addresses of the heap denoted by `Ps` is nonempty, since The postcondition states existence of a list of addresses `as` which is equal to the tail of the input list and connects the pointer `p` to `Null` by means of the next field. We use an existential quantifier here, since we do not have a way to name the remaining of the heap after the assignment `p := p^.next`. An application of proof method `auto`, which combines simplification with classical reasoning, solves the goal completely. Applying the verification condition generator, we are left with the following proof state:

```
proof (state)
goal (1 subgoal):
 1. ⋀next p q.
       List next p Ps ∧ p ≠ Null ⟹
       ∃a as.
          List next (next (addr p)) as ∧
          p = Ref a ∧ a # as = Ps
```

Since the reference `p` is not `Null`, `Ps` in `List next p Ps` is non-empty and `p = Ref a` for some address `a`. Thus `Ps` can be factored as `Ps = a#as` for some list `as`. Thus, it follows that `List next (next y) bs = List next (next (addr p)) as`. Hence we have a witness for the existential goal. A formalization of this very argument in the Isar proof language is shown in Figure 4.

```
proof (vcg)
  fix "next" p q val
  assume ass:"List next p Ps ∧ p ≠ Null "
  from this obtain a as where "Ps = a # as" and
   "p = Ref a" and "List next (next a) as" by auto
  from this show "∃a as. List next (next (addr p)) as
      ∧ p = Ref a ∧ a # as = Ps" by simp
qed
```

Figure 4. Deletion of first node

The reasoning is contained within the outermost proof delimiters `proof (vcg)...qed`, where the verification condition generator `vcg` is the argument for the `proof` command. This `proof` command is concerned to the proof of the Hoare triple. The next three lines exhibit the `fix...assume...show` structure of Isar proofs. We `fix` the arbitrary variables, we `assume` the assumptions of the proof and after `show` we state the goal of the proof. The predefined name `this` is used to refer to the proposition proved in the previous step. The variables `a,as` after the `obtain` clause are arbitrary variables related to the application of the natural deduction rule for existential elimination. We can avoid the use of the existential quantifier in the postcondition had we used a ghost variable to save the value of the initial list of addresses, as shown in Figure 5.

```
lemma "VARS (next::'a ⇒ 'a ref) (p::'a ref)
       (ps::'a list) (q::'a ref) (val::'a)
{List next p Ps ∧ p ≠ Null}
  ps := Ps;q := p; p := p^.next;
  ps := tl ps; val := addr q
{List next p ps  ∧ val # ps = Ps}"
  apply (vcg)
  apply (auto)
done
```

FIGURE 5. Deletion of first node - Ghost Variables

The use of ghost variable `ps` saves the initial value of the list `Ps` and corresponds to the existential variable in the postcondition of the triple in Figure 5. It plays no special role in the computation but it increases the degree of automation, since we do not have to provide witnesses for existential formulas anymore. Note that after we move the pointer `p` one cell further, we update the value of the ghost variable `ps` to reflect that it now denotes the tail of the initial list of addresses. A ghost variable is not needed for the program to execute. The purpose of ghost variables is to provide a way for the programmer to guide the verifier in checking that a program is correct. Ghost variables usually eliminate the need for existential assertions, and tend to make proofs more amenable to automation. The disadvantage of this approach is that entities from the logic, like lists and sets, enter the program text, thus inserting code that it is irrelevant for solving the problem itself.

```
lemma "VARS (next::'a ⇒ 'a ref) (p::'a ref)
         (ps::'a list) (k::nat) j
{List next p Ps ∧ j = length Ps}
k:=0;
WHILE p ≠ Null
INV {∃ as. List next p as ∧ length as + k = j}
DO p := p^.next; k := k+1 OD
{j = k ∧ List next p []}"
apply (vcg)
  apply (fastforce)
  apply (fastforce)
  apply (fastforce)
done
```

FIGURE 6. Number of nodes in a linked list

Our second preliminary example is a simple program that counts the number of nodes in a linked list, as shown in Figure 6. In the precondition, `Ps` is the list of addresses that connects `p` to `Null` by the `next` field and `j` is a logical variable that holds the initial length of `Ps`. In the postcondition, the list of addresses is empty (and hence that $\texttt{p} = \texttt{Null}$) and the counter `k` equals the initial value `j`. The loop is annotated with an invariant strong enough to imply the postcondition. The existential assertion states that at the beginning and after each loop body execution there is a list of addresses that connects `p` to `Null` and that the total number of address nodes is always equal to sum of the current values of the counter `k` and length of the list `as`. The first line of the proof script is a call to the verification condition generator. After its execution, we are left with the following verification conditions:

```
proof (prove)
goal (3 subgoals):
 1. ⋀next p ps k j.
       List next p Ps ∧ j = length Ps ⟹
       ∃as. List next p as ∧ length as + 0 = j
 2. ⋀next p ps k j.
       (∃as. List next p as ∧ length as + k = j) ∧
       p ≠ Null ⟹
       ∃as. List next (next (addr p)) as ∧
            length as + (k + 1) = j
 3. ⋀next p ps k j.
       (∃as. List next p as ∧ length as + k = j) ∧
       ¬ p ≠ Null ⟹
       j = k ∧ List next p []
```

The three proof goals correspond to the statements that the invariant is true at the beginning of the loop, that it is maintained by the loop body and that it is strong enough to entail the postcondition. The three proof goals are solved by applying the automatic proof tactic `fastforce`. This proof method attempts to prove the the current subgoal using sequent-style reasoning and also employing the simplifier as an additional wrapper. The first verification condition is true because the assumption is provides the witness to justify the existential claim, i,e., the list `Ps` itself together with the fact that `0` is the identity for addition. The third follows from the fact that $\texttt{p} = \texttt{Null}$ and by equation HNull, that `List next Null as` entails $\texttt{as} = \texttt{Null}$. The see why the second is also true, note that from the assumption, we know there is a non-empty ($\texttt{p} \neq \texttt{Null}$) list of addresses `as` that connects `p` to `Null`. From this, it follows that there is some address `a` and some sublist `bs` such that $\texttt{p} = \texttt{Ref a}$ and $\texttt{as} = \texttt{a\#bs}$. Then we have $\texttt{lengh bs} + 1 = \texttt{length as}$ and also that `bs` is a list of addresses that connects `next a` to `Null` by means of the `next` field. Now with $\texttt{addr p} = \texttt{addr (Ref a)} = \texttt{a}$ and $\texttt{length as} + \texttt{k} = \texttt{j}$ we have that $\texttt{length bs} + \texttt{k} + 1 = \texttt{j}$ and `List next (next (addr p)) bs`, which is sufficient to establish the conclusion. The reasoning corresponding to the truth of the second verification condition is formalized with the Isar proof language in Figure 7.

The proof is enclosed within the outermost brackets `proof...qed`. The argument for the proof command is the verification condition generator. The command `(fastforce)+` after the closing `qed` solves automatically the

```
proof (vcg)
  fix "next" and p::"'a ref" and  ps k j
  assume ass: "(∃as. List next p as ∧
                length as + k = j) ∧  p ≠ Null"
  show " ∃as. List next (next (addr p)) as ∧
               length as + (k + 1) = j"
    proof -
       from ass  obtain as  where
       list: "List next p as " and len:"length as + k = j"
        and nNull: "p ≠ Null" by blast
       from nNull and list  obtain a bs
         where   "p = Ref a" and  "as = a # bs"
          and "List next (next a) bs"  by auto
       from this and len  have "length bs + k + 1 = j"
        and "List next (next (addr p)) bs" by auto
      from this show ?thesis by auto
    qed
qed (fastforce)+
```

Figure 7. Number of Nodes - Second VC

first and third verification condition. The first three lines exhibit the `fix`, `assume`,`show` structure of Isar proofs. We `fix` the arbitrary variables, we `assume` the assumptions of the proof and after `show` we state the goal of the proof, which is an existential claim. The reasoning inside the innermost `proof`...`qed` brackets is the formal proof that the invariant is maintained by the loop. The hyphen (-) as argument to the `proof` command means that the proof state remains unchanged, i.e., the goal remains the existential formula stated after the last `show`. The command `obtain` in the first line is used to obtain a local context for the existential elimination of the assumption `ass`. It means the same as declaring an arbitrary variable such that the existential property holds. The next two intermediate assertions inferred with `have` are used as aids in establishing the conclusion stated by `?thesis`. The predefined name `this` refer to the proposition(s) inferred in the previous step and `?thesis` always makes reference to the proposition stated after the last preceding `show`.

6. Case Study : Deletion at the End

In Figure 8 we show a partial correctness assertion for a program that deletes the last node in a non-empty linked list. The list head is the pointer `p` and we use two additional references: the pointer `q` which is used to traverse the list in search of the last but one node, and `tmp` which always points to the predecessor of `q`. We discuss the loop invariant bellow.

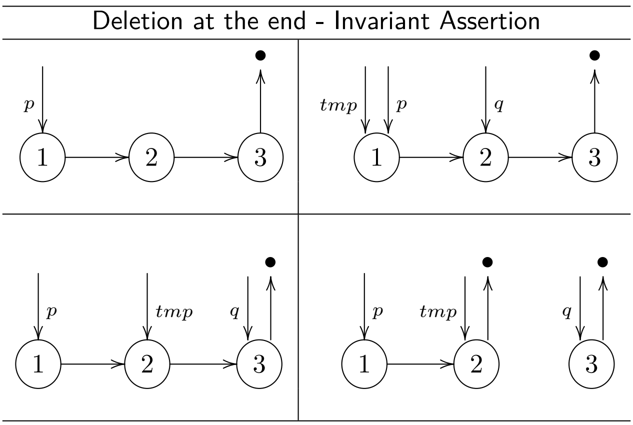
In the precondition, the logical variable `Ps` saves the initial value of the list. The postcondition just states that `p` now points to a list that is equal

```
lemma "VARS p next q tmp  ps qs
{List next p Ps  ∧ p ≠ Null}
IF p^.next = Null
  — ‹Ps has exactly one element›
  THEN p:= Null;ps:=[];qs:=Ps
ELSE
  — ‹Ps has at least two elements ›
  tmp := p; q:= p^.next;
  ps :=[hd Ps]; qs := tl Ps;
  WHILE q^.next ≠ Null
  INV  {inv_del_end}
  DO
     tmp := q; ps := ps @ [hd qs];
     q:= q^.next;qs := tl qs
  OD;
  tmp^.next := Null
FI
{∃ a as. List next p as ∧ as @ [a] = Ps}"
```

FIGURE 8. Hoare Triple - Deletion at the end

to the input list with the last element removed. The ghost variables `ps` and `qs` are used to keep track of two lists segments: `ps` is the portion of the list already traversed and that does not contain the last element. Similarly, `qs` is the remainder of the list that still needs to be searched for the last element. By using these variables, we can avoid using existential assertions.

To see the invariant relations that are maintained by the loop, look at the four states represented in the following table, where the `Null` pointer is denoted by a bullet. The top left diagram denotes an initial state. The top right, the state after initialization in the `ELSE` statement. In bottom left we have the state after the first and only pass through the loop, while the bottom right is the final state, after execution of the assignment right after the loop exit.



In the top left, it is true that `List next p [1,2,3]`. In the bottom right, it does hold that `List next q [2,3]`, but not that `List next p [1]`, since a list of addresses must end with `Null`. However, it is true that `List (next(1:=Null)) p [1]`, i.e, that list where the next address of `1` is `Null`. But this is the same as saying `List (next(addr tmp:=Null)) p [1]` By the same token, in the bottom left state we have that `List (next(addr tmp :=Null)) p [1,2]` and `List nex q [3]`. Note also the address of the reference `tmp` is always the last element of the list segment already searched. Moreover, the two heaps pointed to by `p` and `q` denote distinct portions of the local memory. This discussion motivates the loop invariant shown in next Figure.

```
INV  {p ≠ Null ∧ List next p Ps
      ∧ List next q qs
      ∧ List (next(last ps := Null)) p ps
      ∧ ps @ qs = Ps
      ∧ set ps ∩ set qs = {}
      ∧ next (last ps) = q
      ∧ last ps = addr tmp
      ∧ ps ≠ [] ∧ qs ≠ []
     }
```

After applying the verification condition generator, we are left with three goals: that the invariant is true before the loop, that it is maintained by the loop code and that it is strong enough to entail the postcondition. Due to the size of our invariant assertion, the goals are really long. In Figure 9 we show the third one.

```
3. ⋀p next q tmp ps qs.
      (∃y. p = Ref y) ∧
      List next p Ps ∧
      List next q qs ∧
      List (next(last ps := Null)) p ps ∧
      ps @ qs = Ps ∧
      set ps ∩ set qs = {} ∧
      next (last ps) = q ∧
      last ps = addr tmp ∧ ps ≠ [] ∧ qs ≠ [] ∧ next (addr q) = Null ⟹
      ∃a as. List (next(tmp → Null)) p as ∧ as @ [a] = Ps
```

FIGURE 9. Deletion at the end - third vc

Remind that the (last) long right arrow separates the assumptions (in this case, a single one) from the conclusion. To see that it is true, consider the following argument: from the assumption we know that `List next q qs` and `qs` $\neq [\,]$ and `next (addr q) = Null` and `List (next(last ps := Null)) p ps` and `ps @ qs = Ps` and that `last ps = addr tmp`. From `List next q qs` and `qs` $\neq [\,]$, we know that there is an address `a` and a list `as` such that `q=Ref a` and `qs = a # as` and `List next (next a) as`. Since we know that `next (addr q) = Null`, then with the facts `List next (next a) as` and `q=Ref a`, we have that `as` $= [\,]$. From this, and reminding that `ps @ qs = Ps`, we have that `ps @ [a] = Ps`. With this and reminding that `ps @ qs = Ps`, `List (next(last ps := Null)) p ps` and `last ps = addr tmp`, the postcondition follows. The formalization of this very argument in the Isabelle's proof language Isar is shown in Figure 10.

```
fix p "next" q tmp ps qs
assume ass:"(∃y. p = Ref y) ∧ List next p Ps ∧
  List next q qs ∧ List (next(last ps := Null)) p ps
  ∧ ps @ qs = Ps ∧ set ps ∩ set qs = {}
  ∧ next (last ps) = q ∧ last ps = addr tmp
  ∧ ps ≠ [] ∧ qs ≠ [] ∧ next (addr q) = Null"
show "∃a as.
        List (next(tmp → Null)) p as ∧ as @ [a] = Ps"
 proof -
   from ass have lqs:"List next q qs" and "qs ≠ []"
     and nq:"next (addr q) = Null" and
    lps: "List (next(last ps := Null)) p ps"
   and pq: "ps @ qs = Ps" and
      tmp:"last ps = addr tmp" by auto
   from this(1-2) obtain a as where "q=Ref a" and
     qs:"qs = a # as" and "List next (next a) as"
     by (induction qs) simp_all
   from this(3) and nq and ‹q=Ref a›
      have "as = []" by simp
   from pq and this and qs have "ps @ [a] = Ps" by simp
   from lps and this and tmp  show ?thesis
     by auto
 qed
```

FIGURE 10. Isar Proof - third vc

The proof is enclosed within the outermost brackets `proof...qed`. The first three declarations exhibit the `fix`, `assume`,`show` structure of Isar proofs. We `fix` the arbitrary variables, we `assume` the assumptions of the proof and after `show` we state the goal of the proof, which is an existential claim. The hyphen as an argument to the `proof` command means that the proof state remains unchanged, i.e., the goal remains the existential formula stated after the previous `show`. We use labels to name several facts that will be used later in the chain of reasoning. The command `obtain` is used to declare a local context for the existential elimination implicit in the definition of the predicate `List`. It means the same as declaring an arbitrary variable such that the existential property holds. Intermediate assertions inferred with `have` are used as aids in establishing the conclusion stated by `?thesis`. The predefined name `this` refer to the proposition(s) inferred in the previous step and `?thesis` always makes reference to the proposition stated after the last preceding `show`.

The informal argument given above is a direct rewording of the Isar proof in natural language. The whole proof is long and somewhat involving, especially the proof of the second condition. It can be found in [12].

There are at least two ways we can address the complexity of reasoning in such structures. First we can prove a sufficient set of theorems that can be useful to a specific domain one is interested in. This, together with a powerful set of automatic proof tools can increase considerably the degree of automating proofs in these domains. The Isabelle library `Hoare-Logic` provides a sufficient set of dedicated lemmas that help the user to sketch and explore his/her proofs. Another way, which I think is especially appealing, is to provide the user with a high level language with which he/she can break a complex proof into small chunks, which can be better understood and solved automatically by the system. The Isar proof language has a rich set of constructions that supports breaking a long, challenging proof into more manageable small parts. These simpler chunks can then be chained into a complete proof.

7. Case Study : In-place List Reversal

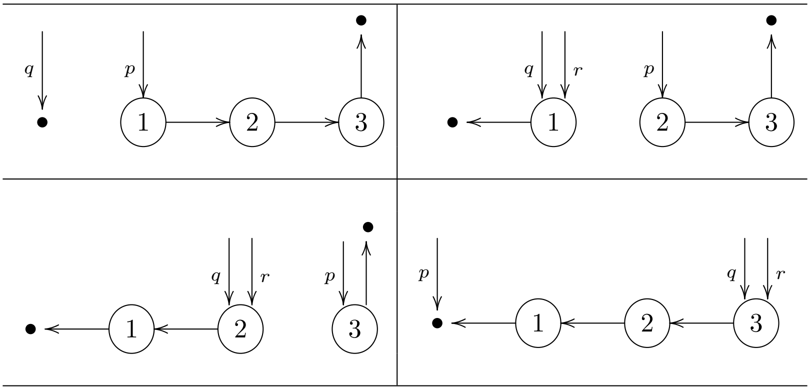
This example is presented and discussed in [13, 5]. I include it here to provide a more in-depth discussion of its proof with Isabelle. In Figure 11 we show a partial correctness assertion for a program that reverses a linked list in one pass, without additional memory. The list head is the pointer `p` and we use two additional references: the pointer `q` always points to the first node of the reversed portion of the list, and `tmp` holds the current value of `p` before it is moved to the next node.

```
lemma in_place_rev: "VARS p q tmp next
{List next p Ps}
q := Null;
WHILE p ≠ Null
  INV {∃ ps qs. List next p ps ∧ List next q qs
       ∧ set ps ∩ set qs = {}
       ∧  rev ps @ qs = rev Ps}
DO
  tmp := p; p:= p^.next;
  tmp^.next := q; q:=tmp
OD
{List next q (rev Ps)}"
```

FIGURE 11. Hoare Triple - In Place List Reversal

In the precondition, the logical variable `Ps` saves the initial value of the list. The postcondition just states that `q` now points to the reverse of the input list. To see the invariant relations that are maintained by the loop, look at the four states represented in the following table, where the `Null` pointer is denoted by a bullet:

List Reversal - Invariant Assertion



The four diagrams represent the states that are true before and after each pass in the loop body, ordered from left to right. The last graph denotes the state after the loop exit. For instance, note that in the first diagram we have that `List next q []` and `List next p [1,2,3]`, while in the third it holds that `List next q [2,1]` and `List next p [3]`. Also, in each state the set of addresses that are linked in each list are mutually disjoint. Moreover, in each of the four states we have the identity `rev Ps = rev ps @ qs`. In the invariant, we need existential quantifiers to refer to the current list of addresses pointed to by `p` and `q`. The function `set` returns the set of elements from a list.

After applying the verification condition generator for this triple, we are left with to the three verification conditions: that the invariant is true after initialization, that it is maintained by the loop and that it is strong enough to entail the postcondition. The first is true because the list pointed to by `q` is empty. The third holds for the list pointed to by `p` is empty. Both can be solved automatically. The first with simplifier and the third by using `auto`. So we concentrate on the second verification condition shown bellow:

```
2. ⋀p q tmp next.
      (∃ps qs.
          List next p ps ∧
          List next q qs ∧
          set ps ∩ set qs = {} ∧ rev ps @ qs = rev Ps) ∧
      p ≠ Null ⟹
      ∃ps qs.
         List (next(p → q)) (next (addr p)) ps ∧
         List (next(p → q)) p qs ∧
         set ps ∩ set qs = {} ∧ rev ps @ qs = rev Ps
```

Trying to solve it by automatic methods and Sledgehammer [1], we get a solution that uses calls to satisfaction modulo theories solvers with a bunch of lemmas from the library:

```
apply (clarsimp)
apply (smt List_hd_not_in_tl
  disjoint_iff_not_equal notin_List_update
  set_ConsD)
```

The partial automatic method `clarsimp` applies a number of safe steps (transformations that do not loose information) and then applies simplification. The second proof command uses a call to SMT solvers (CVC4, Z3) to solve the remaining subgoal. The outcome of the smt command depends on tools external to Isabelle, so it can be hard to predict if they will prove the same things in the future or if they will even still be available in an Isabelle-compatible form in a number of years. On top of it, we do not really get any new insight from this script, i.e., why the invariant is maintained by the loop.

To see that it is true, consider the following argument: the assumption tell us that there exists lists of addresses `ps,qs` such that `ps` and `qs` are list of addresses that connect `p` and `q`, respectively, to `Null`, that the set of address from `ps` and `qs` are mutually disjoint, `p` is a non-null pointer and how the lists of addresses are related to the input list. Since `p` is not null, it follows that there is an address `a` such that `p = Ref a` and `ps = a#as`. Thus `a = addr p` and by definition, `as` is a list of addresses that links `next a` to `Null`. This means that `List next (next (addr p)) as`. But, from the fact that `ps` is a list of distinct addresses, we have that $\mathtt{a} \notin \mathtt{as}$ and hence, from the separation lemma, that $\mathtt{List}\ (\mathtt{next}(\mathtt{p} \rightarrow \mathtt{q}))\ (\mathtt{next}\ (\mathtt{addr}\ \mathtt{p}))\ \mathtt{as}$ (1). Now, from the fact that the set of addresses `ps,qs` are mutually disjoint and `p = a#as`, we have that $\mathtt{a} \notin \mathtt{qs}$. Then the list of address `a#qs` is a list that connects the pointer `p` to `Null` if we point the address of `p` to `q`, the pointer of the list `qs`. This means that $\mathtt{List}\ (\mathtt{next}(\mathtt{p} \rightarrow \mathtt{q}))\ \mathtt{p}\ (\mathtt{a}\#\mathtt{qs})$ (2). Now since the two lists `ps=a#as,qs` have mutually disjoint sets of addresses, and reminding that $\mathtt{a} \notin \mathtt{set\ as}, \mathtt{a} \notin \mathtt{set\ qs}$ it follows that $\mathtt{set\ as} \cap \mathtt{set}(\mathtt{a}\#\mathtt{qs}) = \{\ \}$ (3). Moreover, from the assumption `rev ps @ qs = rev Ps`, definition of reverse, and associativity of concatenation, it follows that `rev as @ (a#qs) = rev Ps` (4). Together, (1), (2), (3) and (4) entail the conclusion of the goal.

This argument is formalized with the proof language Isar in Figure 12. To spare some space, we do now show the initial structure of the declarations `fix,assume,show`, but they are strictly based in the goal corresponding to the picture of the verification condition shown earlier. The (broken) assumptions are labeled from `a1` to `a4`. The first and second deductions in the chain is combination of elimination of the existencial quantifier (via `obtain`) with conjunction elimination, to break the compound conjunction into small pieces. The predefined name `this` refer to the proposition(s) inferred in the previous step. The predefined name `?thesis` refers to the goal of the proof, and it is binded to the proposition stated in the last `show`. The intermediate `have` statements are needed to fill the gap between the assumptions and the

```
proof -
  — ‹ breaking assumptions into pieces ›
  from ass obtain ps qs where
    a1:"List next p ps" and a2 :"List next q qs"
    and a3: "set ps ∩ set qs = {}" and
    a4: "rev ps @ qs = rev Ps" and a5:"p ≠ Null"
        by blast
   from a5 and a1 obtain a as where "p=Ref a"
    and "ps = a # as"  and
    "List next (next a) as" and "a = addr p"  by auto
  from ‹List next (next a) as›
        have "a ∉ set as" by simp
  from this and  ‹List next (next a) as›
    have "List (next(a :=  q)) (next a) as" by simp
  from this and  ‹a = addr p›
    have c1:"List (next(p → q)) (next (addr p)) as" by simp
  from ‹ps = a # as› and a3 have "a ∉ set qs" by simp
  from this and a2 and ‹p=Ref a› have
     c2:"List (next(p → q)) p (a#qs)" by simp
  from ‹ps = a # as› and a3 and ‹a ∉ set qs› and ‹a∉ set as›
      have c3: "set as ∩ set (a#qs) = {}" by simp
  from a4 and ‹ps = a # as›
    have "rev as @ (a # qs) = rev Ps" by simp
  from this and c1 and c2 and c3 show ?thesis by blast
qed
```

FIGURE 12. Hoare Triple - Proof VC2

conclusion, which is established and exported by the `show` in the last line. Propositions inferred early in the chain of reasoning can be quoted whenever needed within angular brackets.

We close this section by presenting bellow a modified version of the in-place list reversal program using ghost variables. The additional program variables `qs,ps` keep track of the of the portion of `ps` that is already reversed and what still remains to be processed. Note that the proof script shows that we are able to solve the three verifications using the basically the simplifier and the proof method `auto`. Besides, we do not need the existial quantifiers in the loop invariant anymore.

```
lemma in_place_rev_ghost: "VARS p ps qs q tmp next
{List next p Ps ∧ ps = Ps}
q := Null; qs := [];
WHILE p ≠ Null
  INV {List next p ps ∧ List next q qs
       ∧ set ps ∩ set qs = {}
       ∧  rev ps @ qs = rev Ps}
DO
  tmp := p; p:= p^.next;
  tmp^.next := q; q:=tmp;
  qs := hd ps # qs;ps := tl ps
OD
{List next q (rev Ps)}"
apply (vcg)
  apply (simp)
  apply (clarsimp)
  apply (auto)
done
```

8. Case Study : Cyclic List Reversal

This example is presented and discussed in [13]. I include it here to provide a more in-depth discussion of its proof with Isabelle. Sometimes it is not enough to speak about complete segments (list of addresses that end in `Null`). We may need to reason about segments of addresses between two references. Thus the more general relation `Path next p as q`, meaning that `as` is a list of addresses that connect `p` to `q` by means of the `next` field.

$$\begin{array}{l}\mathtt{Path} :: ('\mathtt{a} \Rightarrow' \mathtt{a\ ref}) \Rightarrow' \mathtt{a\ ref} \Rightarrow' \mathtt{a\ list} \Rightarrow' \mathtt{a\ ref} \Rightarrow \mathtt{bool}\\ \mathtt{Path\ next\ x}\ [\,]\ \mathtt{y} = (\mathtt{x} = \mathtt{y})\\ \mathtt{Path\ next\ x}\ (\mathtt{a\#as})\ \mathtt{y} = (\mathtt{x} = \mathtt{Ref\ a} \wedge \mathtt{Path\ next}\ (\mathtt{next\ a})\ \mathtt{as\ y})\end{array}$$

Obviously, a list is a special case of a path.

```
List next p as = Path next p as Null
```

Paths in general, need not be unique, as for instance, in cyclic lists. Thus, the relation `distPath next x as y` means that `as` is a non-repeating list of addresses that connects `x` to `y` by means of the `next` field.

$$\mathtt{distPath\ next\ x\ as\ y} = \mathtt{Path\ next\ x\ as\ y} \wedge \mathtt{distinct\ as}$$

Paths enjoy also similar properties satisfied by list abstractions:

$$\frac{\mathtt{p} \neq \mathtt{q} \quad \mathtt{distPath\ next\ p\ as\ q}}{\exists \mathtt{a\ bs.\ p} = \mathtt{Ref\ a} \wedge \mathtt{as} = \mathtt{a\#bs} \wedge \mathtt{a} \notin \mathtt{bs}}\ \mathsf{PAci}$$

$$\frac{\mathtt{Path\ next\ x\ (as@bs)\ z}}{\exists \mathtt{y.\ Path\ next\ x\ as\ y} \wedge \mathtt{Path\ next\ y\ bs\ z}}\ \mathsf{PRef}$$

$$\frac{\mathtt{u} \notin \mathtt{set\ as} \quad \mathtt{Path\ (next(u := v))\ x\ as\ y}}{\mathtt{Path\ next\ x\ as\ y}}\ \mathsf{neq\_dP}$$

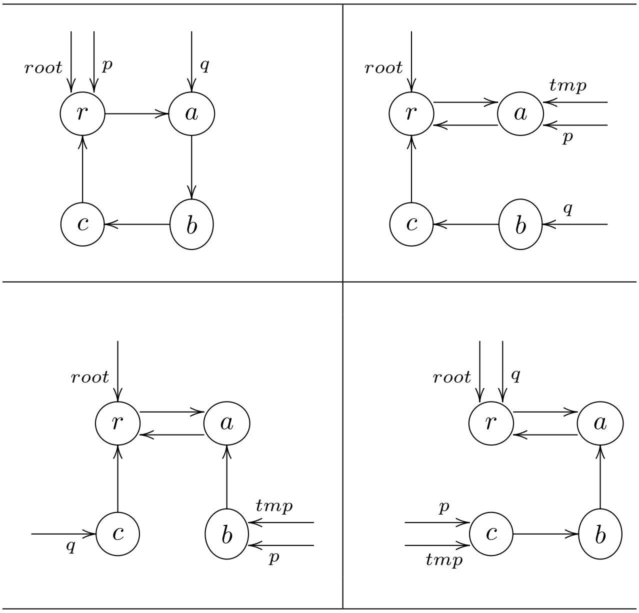
In Figure 13 we show a partial correctness assertion for a program that reverses cyclic list without additional memory. The precondition says that there is a distinct non-empty list of addresses that connects the root address `r` to itself. The postcondition asserts that the distinct list of addresses that follows `r` back to itself is reversed.

```
lemma cyclic_list_rev_I':
  "VARS next root p q tmp
  {root = Ref r ∧ distPath next root (r#Ps) root}
  p := root; q := root^.next;
  WHILE q ≠ root
  INV {∃ ps qs. distPath next p ps root
               ∧ distPath next q qs root ∧
                root = Ref r ∧ r ∉ set Ps  ∧
                set ps ∩ set qs = {} ∧
                 Ps = (rev ps) @ qs  }
  DO tmp := q; q := q^.next; tmp^.next := p; p:=tmp OD;
  root^.next := p
  { root = Ref r ∧ distPath next root (r#rev Ps) root}"
```

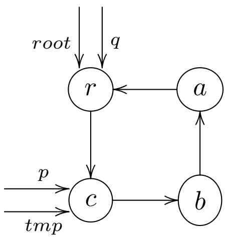
FIGURE 13. Hoare Triple - Cyclic List Reversal

The algorithm uses three auxiliary pointers: `p,q` and `tmp`. After initialization, `p` also points to the root, while `q` points to the root's successor node. During the computation of the while loop, q is always the head of the segment of the list that has yet to be reversed, while `p` points to the part of the list that has already been reversed. As an example, consider the state transitions represented in the table below, where the `Null` pointer is denoted by a bullet:

Cyclic List Reversal - Invariant Assertion



The four diagrams represent the states of the cyclic list that are true before and after each pass in the loop body, ordered from left to right. The first graph denotes the state after the execution of the assignments before the start of the loop. The last diagram (bottom right) represents that state right after the loop exit. Note that in the second state we have `distPath next p [a] root` and `distPath next q [b,c] r`. In the last state, right after the loop exit, we have that `distPath next q [] q` and `distPath next p [c,b,a] root`. Note that in the four states, the non-repeating paths that link `p` and `q` to the root are mutually disjoint. After the end of the loop, there is still one link missing, as the last state show. The subsequent assignment on this state delivers the desired cyclic list reversed, as shown bellow.



After applying the verification condition generator in the triple of Figure 13, we are left with the usual three verification conditions related to the invariant: that it is true after initialization and before the loop starts, that it is maintained by the loop, and that it strong enough to imply the postcondition. We concentrate in the second, most complex one, i.e, that the *invariant is maintained by the loop*, which is shown below:

```
2. ⋀next root p q tmp.
       (∃ps qs.
           distPath next p ps root ∧
           distPath next q qs root ∧
           root = Ref r ∧
           r ∉ set Ps ∧ set ps ∩ set qs = {} ∧ Ps = rev ps @ qs) ∧
       q ≠ root ⟹
       ∃ps qs.
          distPath (next(q → p)) q ps root ∧
          distPath (next(q → p)) (next (addr q)) qs root ∧
          root = Ref r ∧
          r ∉ set Ps ∧ set ps ∩ set qs = {} ∧ Ps = rev ps @ qs
```

Remind that a distinct path (`distPath`) is a path with non-repeating, distinct addresses. We assume the set of assumptions show in the verification condition above. To see that this assertion is valid, note that from the assumptions we know that there is a path `qs1` from `q` to `root` by means of `next` such that `qs1` is a non-repeating path. By the same token, there is a path `ps1` from `p` to `root` by means of `next` such that `ps1` is a non-repeating path. Moreover, the lists `ps1` and `qs1` are mutually disjoint. From the distinct path `ps1` and knowing that `q` $\neq$ `root`, there is some address `x` and some sublist `bs` such that `q=Ref x` and `qs1=x#bs` and `bs` is a distinct path of addresses from `next x` to root by means of next, using property `neq_dP`. Reminding that the two lists `ps1,qs1` are mutually disjoint and that `qs1=x#bs`, it follows that `x` is not in the list `ps1` either. Now we have that `x#ps1` is a distinct path of addresses from `q` to `root` by means of `next`(`q` $\rightarrow$ `p`) (`next (addr p := q)`), reminding that `q = Ref x`. Analogously, since `qs1=x#bs` is a non-repeating path, `x` is not in the list `bs`. This entails that `bs` is distinct path of addresses from `next x` to root by means of `next`. But `x=addr q` is not in the list `bs` either, so `bs` is also a non-repeating path from `next x` to root by means of `next`(`q` $\rightarrow$ `p`). Moreover, from the fact that the non-repeating paths `ps1` and `qs1` are mutually disjoints, it follows that the distinct list `x#ps1` and `bs` are also disjoint. Finally, we have that `rev (x#ps1) @ bs = rev ps1 @ (x#bs1) = rev ps1 @ qs1`. This concludes our informal argument. Figure 14 show a formal version of this reasoning in Isabelle's proof language Isar.

In the proof, each intermediate step is claimed with `have` and proved using one of the several automatic classical reasoners of Isabelle (`simp, auto, blast`). The proof goal itself is proved in the final `show` using intermediate assertions inferred earlier in the reasoning chain. `Metis` is an automatic theorem prover for first order logic with equality. In the proof chain, fact that have already been inferred can be quote directly using angle brackets delimiters, which turns the proof easier and more pleasant to follow.

```
proof -
  from ass have "root = Ref r" and "r ∉ set Ps"
    and "q ≠ root" by auto
  from ass obtain ps1 qs1 where
    pr:"Path next p ps1 root" and dps:"distinct ps1" and
    qr:"Path next q qs1 root" and dqs:"distinct qs1" and
    setI:"set ps1 ∩ set qs1 = {}" and
      psr: "Ps = rev ps1 @ qs1"  by blast
  from qr and ‹q≠root› and dqs  obtain x bs where
   "q=Ref x" and "qs1= x#bs"
   "Path next (next x) bs root"
    by (metis Path.simps(2) neq_dP)
  from setI and ‹qs1 = x#bs› have "x ∉ set ps1" by simp
  from this and pr and ‹q=Ref x› and dps have
    c1:"Path (next(q→p)) q (x#ps1) root ∧ distinct (x#ps1)"
      by simp
  from dqs and ‹qs1 = x#bs› have "x ∉ set bs" by simp
  from this and  qr and ‹qs1 = x#bs› and dqs and ‹q=Ref x›
    have c2:"Path (next(q → p)) (next (addr q)) bs root ∧
           distinct bs" by simp
  from setI and ‹qs1 = x#bs› and dps and dqs
    have c3:"set (x#ps1) ∩ set bs = {}" by simp
  from psr and ‹qs1 = x#bs›
    have c4:"rev (x#ps1) @ bs = rev ps1 @ qs1" by simp
  from this c1 and c2 and c3 and c4 and psr
    and ‹root = Ref r› and ‹r ∉ set Ps›
  show ?thesis  by metis
 qed
```

FIGURE 14. Cyclic List Reversal - Isar Proof

9. Concluding Remarks

In this report I have tried to provide an accessible presentation on how to prove properties of programs using linked lists data structures with Hoare Logic in Isabelle/HOL. The fundamental idea is to represent a local, linked chunk of memory as a function from addresses to references. This local heap is mapped into a corresponding list of addresses. Thus, standard methods for reasoning about lists can be applied to infer properties of the underlying linked structure.

The problem of aliasing is solved by modeling record fields as functions from addresses to values. Pointer assignment is then coded as function updates on addresses. This simple trick allow us to use traditional Hoare calculus in order to reason about pointer structures.

Local reasoning is achieved by relation abstraction. The local heap representing a given linked structure is lifted into lists of non-repeating addresses. This support assertions about small, distinct segments of the storage. Thus, local changes in the program code are reflected by a corresponding local reasoning. A number of logical properties related to these abstractions are automatically fed to the simplifier, conveying a great degree of automation for the reasoning process.

The sophisticated tools for proving available in Isabelle provide a great degree of automation to the whole process of verification. Proof scripts are very important for initial trials and proof exploration. But readable proofs, intended for communication and understanding, have to be given at a more abstract, structured level, similar to informal mathematical proofs found in books and journal articles. In this sense, the Isar proof language is fundamental, because it supports readable and structure proofs, and provide a suitable language with which we can both communicate clearly our ideas. Besides, it provides a rich set of constructions that support proving challenging goals from simpler, easier to understand smaller subgoals.

References

[1] Jasmin Blanchete. User's Guide to Sledgehammer. `http://isabelle.in.tum.de/documentation.html`, June 2018.

[2] Richard Bornat. Proving pointer programs in hoare logic. In *Mathematics of Program Construction, 5th International Conference, MPC 2000, Ponte de Lima, Portugal, July 3-5, 2000, Proceedings*, pages 102–126, 2000.

[3] Richard Bornat. *Proof and Disproof in Formal Logic: An Introduction for Programmers*. New York, Oxford University Press, 2005.

[4] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with isabelle/hol. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 134–149, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In *Proceedings of the Seventh Annual Machine Intelligence Workshop, Edinburgh*, pages 23–50. Edinburgh University Press, 1972.

[6] Cristiano Calcagno, Samin S. Ishtiaq, and Peter W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in hoare logic. In *Proceedings of the 2nd international ACM SIGPLAN conference on on Principles and practice of declarative programming*, pages 190–201, 2000.

[7] Michael J. C. Gordon. *Programming language theory and its implementation - applicative and imperative paradigms*. Prentice Hall International series in Computer Science. Prentice Hall, 1988.

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

[9] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[10] Peter Lammich. Refinement to imperative hol. *J. Autom. Reason.*, 62(4):481–503, April 2019.

[11] Alfio Martini. Reasoning about Imperative Programs in Higher Order Logic. `https://github.com/alfiomartini/hoare-imp-isab`, June 2019.

[12] Alfio Martini. Reasoning about Pointer Structures in Higher Order Logic. `https://github.com/alfiomartini/hoare-pointers-isab`, July 2019.

[13] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.

[14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

[15] Peter O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, January 2019.

[16] Robert W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, 19, 01 1967.

A. R. Martini
Av. Marechal Andrea 11/210
91340-400 - Porto Alegre - Brazil
e-mail: `alfio.martini@gmail.com`