Interactive Theorem Proving and Pointer Structures*

Alfio Martini

[1]Av. Marechal Andrea 11/210
91340-400 – Porto Alegre – RS – Brazil

alfio.martini@gmail.com

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.*

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 [Bornat 2000].

In Isabelle 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 [Mehta and Nipkow 2005]. 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.

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. It extends and complements [Mehta and Nipkow 2005] in the following way: we discuss in considerable detail the mathematical structures used to model linked lists in Isabelle. Besides that,

*Work completed: see at https://github.com/alfiomartini/hoare-pointers-isab.

we highlight a proof methodology base on proof scripts and high level structured proofs in Isar. The first is very helpful in proof exploration, while the second is fundamental to control proof complexity and to convey clear reasoning. As an example of this approach, we develop in detail a correctness proof of a non-trivial case study: deletion of a node at the end of a linked list.

The text is organized as follows: section 2 describes and illustrates the basic model for references and linked lists. In section 3 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. In section 4 we describe some basic concepts for describing and conducting proofs of imperative programs with Hoare logic in Isabelle/HOL. In section 5, we present our case study: deletion at the end of a linked list. Finally, in section 6 we summarize the main ideas discussed in this report. The complete report related to this work together with the corresponding Isabelle theories can be found at https://github.com/alfiomartini/hoare-pointers-isab.

2. References and Linked Lists

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.

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

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 `addr` :: `'a ref` $\Rightarrow$ `'a` unpacks the address from container `Ref`, i.e., `addr (Ref x)` = `x`. A local heap model is represented 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` [Martini 2019]. Based on the syntax of Pascal, the formalization of the Heap syntax provides the following syntactic sugar:

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

Linked lists are represented by their `next` field that maps addresses to references, i.e., a heap of type `'a` $\Rightarrow'$ `a ref`, where the type variable `'a` is an arbitrary

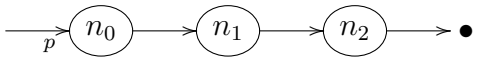

Figure 1. Linked List - Students

type of addresses. Moreover, an abstraction of a linked list of type `next` is a HOL list of type `'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. A simple 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.

```
—‹ 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

The reference p points to address 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 the pointer `p` and `tmp` are aliases to location `0`.

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 model directly as a Hoare triple, as shown in Figure 3. 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
```

Figure 3. Students Model - Hoare Triple

3. 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 [Mehta and Nipkow 2005]). 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.

$$\begin{array}{l}\texttt{List} :: (\texttt{'a} \Rightarrow' \texttt{a ref}) \Rightarrow' \texttt{a ref} \Rightarrow' \texttt{a list} \Rightarrow \texttt{bool}\\ \texttt{List next r [ ]} = (\texttt{r} = \texttt{Null})\\ \texttt{List next r (a\#as)} = (\texttt{r} = \texttt{Ref a} \wedge \texttt{List next (next a) as})\end{array}$$

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

$$\begin{array}{ll}\texttt{List next Null as} = (\texttt{as} = [\,]) & \textsf{(HNull)}\\ \texttt{List next (Ref a) as} = & \\ \quad (\exists\, \texttt{bs}.\ \texttt{as} = \texttt{a\#bs} \wedge \texttt{List next (next a) bs}) & \textsf{(HRef)}\end{array}$$

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`.

$$\begin{array}{ll}\texttt{List next x as} \wedge \texttt{List next x bs} \rightarrow \texttt{as} = \texttt{bs} & \textsf{LFun}\\ \texttt{List next x (as@bs)} \rightarrow \exists \texttt{y}.\ \texttt{List next y bs} & \textsf{LRef}\\ \texttt{List next(next a) x as} \rightarrow \texttt{a} \notin \texttt{set as} & \textsf{LAci}\\ \texttt{List next x as} \rightarrow \texttt{distinct as} & \textsf{LDist}\\ \texttt{a} \notin \texttt{set as} \quad \texttt{List (next(a := y)) x as} \rightarrow \texttt{List next x as} & \textsf{LSep}\end{array}$$

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`. The rule LDist states that all elements in a local list of addresses pointed to by `x` are non-repeating , 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.

4. 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`[1]. Our discussion assumes acquaintance with formal proofs in Hoare Logic and the concept of verification conditions associated with annotated Hoare triples [Gordon 1988].

Using Hoare logic [Hoare 1969], 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* [Nipkow and Klein 2014]. HOL can be understood by the equation HOL = Functional Programming + 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 ⇒. 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 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

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

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.

5. Case Study : Deletion at the End

In Figure 4 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 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.

```
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 4. Hoare Triple - Deletion at the end

To see the invariant relations that are maintained by the loop, look at the four states represented in Table 1, 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

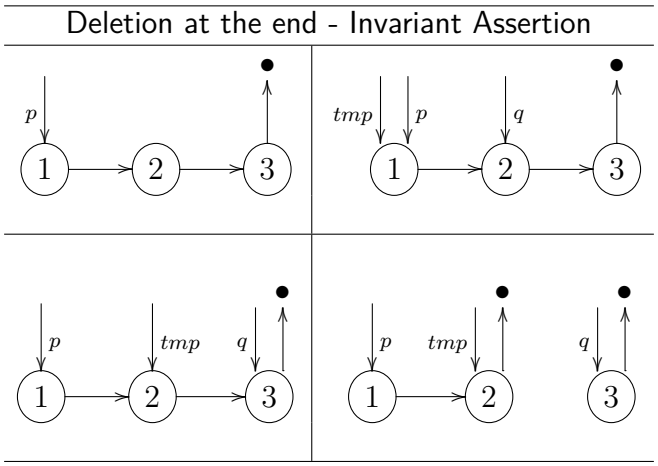

Table 1. Invariant - Deletion at the end

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 Figure 5.

```
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 ≠ []
     }
```

Figure 5. Invariant - Deletion at the end

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 6 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 6. 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 7.

```
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 7. 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 [Martini 2019]. 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.

6. Concluding Remarks

In this work 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.

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

Bornat, R. (2000). 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.

Gordon, M. J. C. (1988). *Programming language theory and its implementation - applicative and imperative paradigms*. Prentice Hall International series in Computer Science. Prentice Hall.

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

Martini, A. (2019). Reasoning about Pointer Structures in Higher Order Logic. https://github.com/alfiomartini/hoare-pointers-isab.

Mehta, F. and Nipkow, T. (2005). Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227.

Nipkow, T. and Klein, G. (2014). *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated.