Proving Pointer Programs in Higher-Order Logic

Farhad Mehta and Tobias Nipkow

Institut für Informatik, Technische Universität München
http://www.in.tum.de/~{mehta,nipkow}/

Abstract. This paper develops sound modelling and reasoning methods for imperative programs with pointers: heaps are modelled as mappings from addresses to values, and pointer structures are mapped to higher-level data types for verification. The programming language is embedded in higher-order logic, its Hoare logic is derived. The whole development is purely definitional and thus sound. The viability of this approach is demonstrated with a non-trivial case study. We show the correctness of the Schorr-Waite graph marking algorithm and present part of the readable proof in Isabelle/HOL.

1 Introduction

It is a truth universally acknowledged, that the verification of pointer programs must be in want of machine support. The basic idea in all approaches to pointer program proofs is the same and goes back to Burstall [4]: model the heap as a collection of variables of type $address \to value$ and reason about the programs in Hoare logic. A number of refinements of this idea have been proposed; see [11] for a partial bibliography. The most radical idea is that of *separation logic* [12]. Although very promising, it is difficult to combine with existing theorem proving infrastructure because of its special logical connectives. Instead we take Bornat's [2] presentation of Burstall's ideas as our point of departure.

Systematic approaches to automatic or interactive verification of pointer programs come in two flavours. There is a large body of work on program analysis techniques for pointer programs. These are mainly designed for use in compilers and can only deal with special properties like aliasing. In the long run these approaches will play an important role in the verification of pointer programs. But we ignore them for now because our goal is a general purpose logic. For the same reason we do not discuss other special purpose logics, e.g. [6].

General theorem proving approaches to pointer programs are few. A landmark is the thesis by Suzuki [13] who developed an automatic verifier for pointer programs that could handle the Schorr-Waite algorithm. However, that verification is based on 5 recursively defined predicates (which are not shown to be consistent — mind the recursive "definition" $P = \neg P$!) and 50 unproved lemmas about those predicates. Bornat [2] has verified a number of pointer programs with the help of Jape [3]. However, his logical foundations are a bit shaky because he works with potentially infinite or undefined lists but explicitly ignores definedness issues. Furthermore, since Jape is only a proof editor with little automation, the Schorr-Waite proof takes 152 pages [1].

The contributions of our paper are as follows:

- An embedding of a Hoare logic for pointer programs in a general purpose theorem prover (Isabelle/HOL).
- A logically fully sound method for the verification of inductively defined data types like lists and trees on the pointer level.
- A readable and machine checked proof of the Schorr-Waite algorithm.

The last point deserves special discussion as it is likely to be controversial. Our aim was to produce a proof that is close to a journal-style informal proof, but written in a stylised proof language that can be machine-checked. Isabelle/Isar [14, 9], like Mizar, provides such a language. Publishing this proof should be viewed as creating a reference point for further work in this area: although an informal proof is currently shorter and more readable, our aim should be to bridge this gap further. It also serves as a reference point for future mechanisations of other formal proofs like the separation logic one by Yang [15].

So what about a fully automatic proof of the Schorr-Waite algorithm? This seems feasible: once the relevant inductive lemmas are provided, the preservation of the invariant in the algorithm should be reducible to a first-order problem (with some work, as we currently employ higher-order functions). If the proof is within reach of current automatic first-order provers is another matter that we intend to investigate in the future. But irrespective of that, a readable formal proof is of independent interest because the algorithm is sufficiently complicated that a mere “yes, it works” is not very satisfactory.

The rest of the paper is structured as follows. After a short overview of Isabelle/HOL notation (§2) and an embedding of a simple imperative programming language in Isabelle/HOL (§3), we describe how we have extended this programming language with references (§4). We show in some detail how to prove programs involving linked lists (§5) and discuss how this extends to other inductive data types (§6). Finally we present our main case study, the structured proof of the Schorr-Waite algorithm (§7).

2 Isabelle/HOL notation

Isabelle/HOL [10] is an interactive theorem prover for HOL, higher-order logic. The whole paper is generated directly from the Isabelle input files, which include the text as comments. That is, if you see a lemma or theorem, you can be sure its proof has been checked by Isabelle. Most of the syntax of HOL will be familiar to anybody with some background in functional programming and logic. We just highlight some of the nonstandard notation.

The space of total functions is denoted by the infix $\Rightarrow$. Other type constructors, e.g. *set*, are written postfix, i.e. follow their argument as in $'a\ set$ where $'a$ is a type variable.

The syntax $[\![P;\ Q]\!] \Longrightarrow R$ should be read as an inference rule with the two premises P and Q and the conclusion R. Logically it is just a shorthand for $P \Longrightarrow Q \Longrightarrow R$. Note that semicolon will also denote sequential composition of programs, which should cause no major confusion. There are actually two implications $\longrightarrow$ and $\Longrightarrow$. The two mean the same thing, except that $\longrightarrow$ is HOL's "real" implication, whereas $\Longrightarrow$ comes from Isabelle's meta-logic and expresses inference rules. Thus $\Longrightarrow$ cannot appear inside a HOL formula. Beware that $\longrightarrow$ binds more tightly than $\Longrightarrow$: in $\forall x.\ P \longrightarrow Q$ the $\forall x$ covers $P \longrightarrow Q$, whereas in $\forall x.\ P \Longrightarrow Q$ it covers only P.

A HOL speciality is its ε-operator: $SOME\ x.\ P\ x$ is an arbitrary but fixed x that satisfies P. If there is no such x, an arbitrary value is returned — note that all HOL types are non-empty! HOL provides the notation $f(a := v)$ for updating function f at argument a with the new value v. Set comprehension is written $\{x.\ P\}$ rather than $\{x \mid P\}$ and is also available for tuples, e.g. $\{(x,\ y,\ z).\ P\}$. Lists in HOL are of type *'a list* and are built up from the empty list [] via the infix constructor # for adding an element at the front. Two lists are appended with the infix function @. Function *set* turns a list into a set, function *rev* reverses a list.

3 A simple programming language

In the style of Gordon [5] we defined a little programming language and its operational semantics. The basic constructs of the language are assignment, sequential composition, conditional and while-loop. The rules of Hoare logic (for partial correctness) are derived as theorems about the semantics and are phrased in a weakest precondition style. To automate their application, a proof method *vcg* has been defined in ML. It turns a Hoare triple into an equivalent set of HOL formulae (i.e. its verification conditions). This requires that all loops in the program are annotated with invariants. More semantic details can be found elsewhere [8]. Here is an example:

lemma *multiply-by-add*: $VARS\ m\ s\ a\ b{::}nat$
$\{a{=}A \wedge b{=}B\}$
$m := 0;\ s := 0;$
$WHILE\ m \neq a\ \ INV\ \{s{=}m{*}b \wedge a{=}A \wedge b{=}B\}\ \ DO\ s := s{+}b;\ m := m{+}1\ OD$
$\{s = A{*}B\}$

The program performs multiplication by successive addition. The first line declares the program variables $m\ s\ a\ b$ to distinguish them from the auxiliary variables A and B. In the precondition A and B are equated with a and b — this enables us to refer to the initial value of a and b in the postcondition.

The application of *vcg* leaves three subgoals: the validity of the invariant after initialisation of m and s, preservation of the invariant, and validity of the postcondition upon termination. All three are proved automatically using linear arithmetic.

1. $\bigwedge m\ s\ a\ b.\ a = A \wedge b = B \Longrightarrow 0 = 0 * b \wedge a = A \wedge b = B$
2. $\bigwedge m\ s\ a\ b.$
   $(s = m * b \wedge a = A \wedge b = B) \wedge m \neq a \Longrightarrow$
   $s + b = (m + 1) * b \wedge a = A \wedge b = B$
3. $\bigwedge m\ s\ a\ b.\ (s = m * b \wedge a = A \wedge b = B) \wedge \neg\ m \neq a \Longrightarrow s = A * B$

4 References and the heap

This section describes how we model references and the heap. We distinguish *addresses* from *references*: a reference is either null or an address. Formally:
datatype $'a\ ref = Null \mid Ref\ 'a$

We do not fix the type of addresses but leave it as a type variable $'a$ throughout the paper. Function $addr :: 'a\ ref \Rightarrow 'a$ unpacks *Ref*, i.e. $addr\ (Ref\ a) = a$.

A simpler model is to declare a type of references with a constant *Null*, thus avoiding *Ref* and *addr*. We found that this leads to slightly shorter formulae but slightly less automatic proofs, i.e. it makes very little difference.

Our model of the heap follows Bornat [2]: we have one heap f of type $address \to value$ for each field name f. 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 $f(addr\ r)$. Based on the syntax of Pascal, we introduce some more convenient notation:

$$\begin{array}{rcl} f(r \to e) & = & f((addr\ r) := e) \\ r\hat{\ }.f := e & = & f := f(r \to e) \\ r\hat{\ }.f & = & f(addr\ r) \end{array}$$

Note that the rules are ordered: the last one only applies if the previous one does not apply, i.e. if it is a field access and not an assignment.

5 Lists on the heap

The general approach to verifying low level structures is *abstraction*, i.e. mapping them to higher level concepts. Linked lists are represented by their 'next' field, i.e. a heap of type
types $'a\ next\ =\ 'a \Rightarrow 'a\ ref$

An abstraction of a linked list of type $'a\ next$ is a HOL list of type $'a\ list$.

5.1 Naive functional abstraction

The obvious abstraction function *list* has type $'a\ next \Rightarrow 'a\ ref \Rightarrow 'a\ list$, where the second parameter is the start reference, and is defined as follows:

$$\begin{array}{l} list\ next\ Null = [] \\ list\ next\ (Ref\ a) = a\ \#\ list\ next\ (next\ a) \end{array}$$

However, this is not a legal definition in HOL because HOL is a logic of total functions but function *list* is not total: *next* could contain a loop or an infinite chain. We will now examine two alternative definitions.

5.2 Relational abstraction

Instead of functions we work with relations. Although we could define the desired relation directly, it turns out to be useful to define a more general relation first: $Path\ next\ x\ as\ y$ means that as is a path of addresses that connects x to y by means of the *next* field.

$Path :: 'a\ next \Rightarrow 'a\ ref \Rightarrow 'a\ list \Rightarrow 'a\ ref \Rightarrow bool$
$Path\ next\ x\ []\ y = (x = y)$
$Path\ next\ x\ (a\#as)\ y = (x \neq y \wedge x = Ref\ a \wedge Path\ next\ (next\ a)\ as\ y)$

This is a valid definition by primitive recursion on the list of addresses. Note that due to the condition $x \neq y$, this list corresponds to the unique minimal length path, which is useful in proofs about circulat lists.

We now define lists as those paths that end in *Null*:

$List :: 'a\ next \Rightarrow 'a\ ref \Rightarrow 'a\ list \Rightarrow bool$
$List\ next\ x\ as \equiv Path\ next\ x\ as\ Null$

It is trivial to derive the following recursive characterisation, which we could have taken as the definition of *List* had we not started with *Path*:

lemma $List\ next\ r\ [] = (r = Null)$
lemma $List\ next\ r\ (a\#as) = (r = Ref\ a \wedge List\ next\ (next\ a)\ as)$

By induction on *as* we can show

$$a \notin set\ as \Longrightarrow List\ (next(a := y))\ x\ as = List\ next\ x\ as \qquad (List\text{-}update\text{-}conv)$$

which, in the spirit of [2], is an important *separation lemma*: it says that updating an address that is not part of some linked list does not change the list abstraction. This allows to localise the effect of assignments.

An induction on *as* shows that *List* is in fact a function

$[\![List\ next\ x\ as;\ List\ next\ x\ bs]\!] \Longrightarrow as = bs$

and that a list is a path followed by a rest list:
$List\ next\ x\ (as\ @\ bs) = (\exists\, y.\ Path\ next\ x\ as\ y \wedge List\ next\ y\ bs)$

Thus a linked list starting at *next a* cannot contain *a*:
lemma $List\ next\ (next\ a)\ as \Longrightarrow a \notin set\ as$

Otherwise *as* could be decomposed into $bs\ @\ a\ \#\ cs$ and then the previous two lemmas lead to a contradiction.

It follows by induction on *as* that all elements of a linked list are distinct:

$$List\ next\ x\ as \Longrightarrow distinct\ as \qquad (List\text{-}distinct)$$

5.3 Examples: Linear and circular list reversal

After this collection of essential lemmas we turn to a real program proof: in place list reversal. We first treat linear acyclic lists.

lemma $VARS\ next\ p\ q\ r$
$\{List\ next\ p\ Ps \wedge List\ next\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\}$
$WHILE\ p \neq Null$
$INV\ \{\exists\, ps\ qs.\ List\ next\ p\ ps \wedge List\ next\ q\ qs \wedge set\ ps \cap set\ qs = \{\} \wedge$
$\quad rev\ ps\ @\ qs = rev\ Ps\ @\ Qs\}$

$DO\ r := p;\ p := p\hat{}.next;\ r\hat{}.next := q;\ q := r\ OD$
$\{List\ next\ q\ (rev\ Ps\ @\ Qs)\}$

The precondition states that Ps and Qs are two disjoint lists starting at p and q. Afterwards, the list starting at q is $rev\ Ps\ @\ Qs$: Ps has been reversed onto Qs. The invariant is existentially quantified because we have no way of naming the intermediate lists.
The argument for circular list reversal is similar:

lemma $VARS\ next\ root\ p\ q\ tmp$
$\{root = Ref\ r \wedge Path\ next\ (root\hat{}.next)\ Ps\ root\}$
$p := root;\ q := root\hat{}.next;$
$WHILE\ q \neq root$
$INV\ \{\exists\ ps\ qs.\ Path\ next\ p\ ps\ root \wedge Path\ next\ q\ qs\ root \wedge$
$\quad root = Ref\ r \wedge set\ ps \cap set\ qs = \{\} \wedge Ps = (rev\ ps)\ @\ qs\ \}$
$DO\ tmp := q;\ q := q\hat{}.next;\ tmp\hat{}.next := p;\ p{:=}tmp\ OD;$
$root\hat{}.next := p$
$\{\ root = Ref\ r \wedge Path\ next\ (root\hat{}.next)\ (rev\ Ps)\ root\}$

5.4 Functional abstraction

The proof of linear list reversal is still automatic. Circular list reversal, and other more complicated algorithms like the merging of two lists require manual instantiation of the existential quantifiers. Although more powerful automatic provers for predicate calculus would help, providing a few witnesses interactively can be more economical than spending large amounts of time coaxing the system into finding a proof automatically.

Trying to avoid existential quantifiers altogether, we turned to a third alternative for abstracting linked lists:
$islist :: 'a\ next \Rightarrow 'a\ ref \Rightarrow bool$
$islist\ next\ p\ \ \equiv\ \ \exists as.\ List\ next\ p\ as$
$list :: 'a\ next \Rightarrow 'a\ ref \Rightarrow 'a\ list$
$list\ next\ p\ \ \equiv\ \ SOME\ as.\ List\ next\ p\ as$

As a direct consequence we obtain:

lemma $List\ next\ p\ as = (islist\ next\ p \wedge as = list\ next\ p)$

The following lemmas are easily derived from their counterparts for $List$ and the relationship just proved:

lemma $islist\ next\ Null$
lemma $islist\ next\ (Ref\ a) = islist\ next\ (next\ a)$
lemma $list\ next\ Null = []$
lemma $islist\ next\ (next\ a) \Longrightarrow list\ next\ (Ref\ a) = a\ \#\ list\ next\ (next\ a)$
lemma $islist\ next\ (next\ a) \Longrightarrow a \notin set(list\ next\ (next\ a))$
lemma $[\![islist\ next\ p;\ y \notin set(list\ next\ p)]\!] \Longrightarrow islist\ (next(y := q))\ p$
lemma $[\![islist\ next\ p;\ y \notin set(list\ next\ p)]\!] \Longrightarrow list\ (next(y := q))\ p = list\ next\ p$

This suffices for an automatic proof of list reversal:

lemma $VARS\ next\ p\ q\ r$
$\{islist\ next\ p \wedge islist\ next\ q \wedge$
$Ps = list\ next\ p \wedge Qs = list\ next\ q \wedge set\ Ps \cap set\ Qs = \{\}\}$
$WHILE\ p \neq Null$
$INV\ \{islist\ next\ p \wedge islist\ next\ q \wedge set(list\ next\ p) \cap set(list\ next\ q) = \{\} \wedge$
$rev(list\ next\ p)\ @\ (list\ next\ q) = rev\ Ps\ @\ Qs\}$
$DO\ r := p;\ p := p\hat{}.next;\ r\hat{}.next := q;\ q := r\ OD$
$\{islist\ next\ q \wedge list\ next\ q = rev\ Ps\ @\ Qs\}$

We have verified a few more algorithms, like searching a list and merging two ordered lists, in the same manner. We found that proofs could eventually be automated by proving further specialized rewrite rules for both *islist* and *list*. But this was less direct and more time consuming than providing existential witnesses for *List*. Thus we believe that relational abstraction, along with its associated existential quantification, is often easier to use than functional abstraction.

5.5 Storage allocation

We conclude the section on lists by showing how we treat the allocation of new storage. Allocated addresses are distinguished from unallocated ones by introducing a separate variable that records the set of allocated addresses. Selecting a new address is easy:

$new :: {'a}\ set \Rightarrow {'a}$
$new\ A \ \equiv\ SOME\ a.\ a \notin A$

As long as the type of addresses is infinite and the set of currently allocated addresses finite, a new address always exists.

The following example program creates a linked list on the heap whose *elem* fields contain the elements of the input list *xs* (of type *'b list*) in reverse order:

lemma $\neg\ finite(UNIV::{'a}\ set) \Longrightarrow$
$VARS\ xs\ elem\ next\ alloc\ p\ (q::{'a}\ ref)$
$\{Xs = xs \wedge p = Null\}$
$WHILE\ xs \neq []$
$INV\ \{islist\ next\ p \wedge set(list\ next\ p) \subseteq set\ alloc \wedge$
$map\ elem\ (rev(list\ next\ p))\ @\ xs = Xs\}$
$DO\ q := Ref(new(set\ alloc));\ alloc := (addr\ q)\#alloc;$
$q\hat{}.next := p;\ q\hat{}.elem := hd\ xs;\ xs := tl\ xs;\ p := q \quad OD$
$\{islist\ next\ p \wedge map\ elem\ (rev(list\ next\ p)) = Xs\}$

We assume that the type of addresses is infinite — *UNIV* is the set of all elements of a given type. Variable *alloc* contains the list (rather than the set) of allocated addresses — lists have the advantage of always being finite. Allocating an address simply means adding it to *alloc*. The input list *xs* is taken apart with *hd* (head) and *tl* (tail). The proof is automatic.

6 Inductive data types on the heap

Every inductively defined data type has a canonical representation on the heap and therefore a canonical relational abstraction. The basic idea is simple: define the abstraction relation inductively, following the inductive definition of the data type. Instead of showing the general case with lots of indices we go through an example, trees. Given the following data type of binary trees:

datatype $'a\ tree = Tip \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

the corresponding abstraction relation is defined as:

$Tree :: 'a\ next \Rightarrow 'a\ next \Rightarrow 'a\ ref \Rightarrow 'a\ tree \Rightarrow bool$

$$Tree\ l\ r\ Null\ Tip$$
$$[\![Tree\ l\ r\ (l\ a)\ t1;\ Tree\ l\ r\ (r\ a)\ t2]\!] \Longrightarrow Tree\ l\ r\ (Ref\ a)\ (Node\ t1\ a\ t2)$$

Of course one could also define *Tree* recursively:

$$Tree\ l\ r\ p\ Tip = (p = Null)$$
$$Tree\ l\ r\ p\ (Node\ t1\ a\ t2) = (p = Ref\ a \land Tree\ l\ r\ (r\ a)\ t1 \land Tree\ l\ r\ (l\ a)\ t2)$$

As in §5.4 we could derive two functions *istree* and *tree* from *Tree*.

Note that *Tree* actually characterizes dags rather than trees. To avoid sharing we need an additional condition in the *Node*-case: $set\text{-}of\ t1 \cap set\text{-}of\ t2 = \{\}$ where *set-of* returns the nodes in a tree. Loops cannot arise because the definition of *Tree* is wellfounded.

7 The Schorr-Waite algorithm

The Schorr-Waite algorithm is a non-recursive graph marking algorithm. Most graph marking algorithms (e.g. depth-first or breadth-first search) are recursive, making their proof of correctness relatively simple. In general one can eliminate recursion in favour of an explicit stack. In certain cases, the need for an explicit stack can be relaxed by using the data structure at hand to store state information. The Schorr-Waite algorithm does just that. The incentive for this is not merely academic. Graph marking algorithms are normally used during the first stage of garbage collection, when scarcity of memory prohibits the luxury of a stack.

The problem with graph marking without recursion is backtracking: we have to remember where we came from. The Schorr-Waite algorithm uses the fact that if we always keep track of the current predecessor node once we have descended into the next node in the graph, the pointer reference from the predecessor to the next node is redundant, and can be put to better use by having it point to the predecessor of this predecessor node, and so on till the root of the graph. If done carefully, this reverse pointer chain preserves connectivity, facilitates backtracking through the graph, and is analogous to a stack. Figure 1 illustrates a complete marking cycle for a small subgraph. We have a pointer to the current node or *tip* (t) and to its previously visited predecessor (p). The tip is marked and the algorithm descends into its left child, updating the predecessor pointer, and using the forward link of the tip to point to its predecessor. The tip has been "pushed" onto the predecessor stack. After exploring the left child, a "swing" is performed to do the same with the right. When all children of our original tip

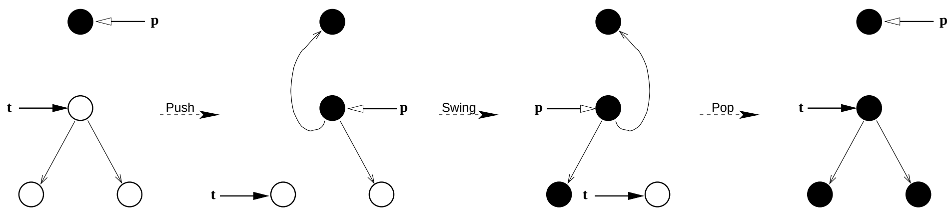

Fig. 1. A marking cycle

have been explored, no more swings are possible, and the tip is "popped" out of the predecessor stack, leaving us with the original subgraph with all reachable nodes marked.

Every pointer that is traversed in the graph is reversed, making it non trivial to see that we are indeed left with the graph we had started with, when the algorithm has terminated. This difficulty is amplified when one tries to formally prove its correctness. The Schorr-Waite algorithm is therefore considered a benchmark for any pointer formalisation. Below is the version of the algorithm we will prove correct in this paper, along with Hoare logic assertions which we will discuss in the next section.

$$
\begin{array}{l}
\mathit{VARS}\ c\ m\ l\ r\ t\ p\ q\ \mathit{root} \\
\{R = \mathit{reachable}\ (\mathit{relS}\ \{l,\ r\})\ \{\mathit{root}\} \wedge (\forall x.\ \neg\ m\ x) \wedge \mathit{iR} = r \wedge \mathit{iL} = l\} \\
t := \mathit{root};\ p := \mathit{Null}; \\
\mathit{WHILE}\ p \neq \mathit{Null} \vee t \neq \mathit{Null} \wedge \neg\ t\hat{\ }.m \\
\mathit{INV}\ \{\exists\, \mathit{stack}. \\
\qquad \mathit{List}\ (S\ c\ l\ r)\ p\ \mathit{stack} \wedge \hfill (*i1*) \\
\qquad (\forall x \in \mathit{set\ stack}.\ m\ x) \wedge \hfill (*i2*) \\
\qquad R = \mathit{reachable}\ (\mathit{relS}\{l,\ r\})\ \{t,p\} \wedge \hfill (*i3*) \\
\qquad (\forall x.\ x \in R \wedge \neg m\ x \longrightarrow \hfill (*i4*) \\
\qquad\qquad x \in \mathit{reachable}\ (\mathit{relS}\{l,r\}|m)\ (\{t\} \cup \mathit{set}(\mathit{map}\ r\ \mathit{stack}))) \wedge \\
\qquad (\forall x.\ m\ x \longrightarrow x \in R) \wedge \hfill (*i5*) \\
\qquad (\forall x.\ x \notin \mathit{set\ stack} \longrightarrow r\ x = \mathit{iR}\ x \wedge l\ x = \mathit{iL}\ x) \wedge \hfill (*i6*) \\
\qquad (\mathit{stkOk}\ c\ l\ r\ \mathit{iL}\ \mathit{iR}\ t\ \mathit{stack}) \hfill (*i7*)\ \} \\
\mathit{DO}\ \mathit{IF}\ t = \mathit{Null} \vee t\hat{\ }.m \\
\quad \mathit{THEN}\ \mathit{IF}\ p\hat{\ }.c \\
\qquad\quad \mathit{THEN}\ q := t;\ t := p;\ p := p\hat{\ }.r;\ t\hat{\ }.r := q \hfill (*pop*) \\
\qquad\quad \mathit{ELSE}\ q := t;\ t := p\hat{\ }.r;\ p\hat{\ }.r := p\hat{\ }.l; \hfill (*swing*) \\
\qquad\qquad\quad p\hat{\ }.l := q;\ p\hat{\ }.c := \mathit{True} \qquad \mathit{FI} \\
\quad \mathit{ELSE}\ q := p;\ p := t;\ t := t\hat{\ }.l;\ p\hat{\ }.l := q; \hfill (*push*) \\
\qquad\quad p\hat{\ }.m := \mathit{True};\ p\hat{\ }.c := \mathit{False} \qquad \mathit{FI} \qquad \mathit{OD} \\
\{(\forall x.\ (x \in R) = m\ x) \wedge (r = \mathit{iR} \wedge l = \mathit{iL})\ \}
\end{array}
$$

We consider graphs where every node has at most two successors. The proof with arbitrary out degree uses the same principles and is just a bit more tedious. For every node in the graph, l and r are pointer fields that point to the successor nodes, m is a boolean field that is true for all marked nodes, and will be the result of running the algorithm. The boolean helper field c keeps track of which of the two child pointers has been reversed. Pointer t points to the tip. It is initially set to the *root*. Within the while loop, the algorithm divides into three arms, corresponding to the operation being performed on the predecessor stack. Pointer p points to the predecessor of t and is also the top of the predecessor stack.

7.1 Specification

The specification uses the following auxiliary definitions:

$$\textit{reachable}\ r\ P \equiv r^{*}\ ``\ \textit{addrS}\ P \qquad (\textit{reachable-def})$$

$$\textit{addrS}\ P \equiv \{a.\ \textit{Ref}\ a \in P\} \qquad (\textit{addrS-def})$$

$$\textit{relS}\ M \equiv \bigcup m{\in}M.\ \{(a,\ b).\ m\ a = \textit{Ref}\ b\} \qquad (\textit{relS-def})$$

$$r \mid m \equiv \{(x,\ y).\ (x,\ y) \in r \wedge \neg\ m\ x\} \qquad (\textit{restr-def})$$

Reachability is defined as the image of a set of addresses under a relation (r `` S is the image of set S under relation r). This relation is given by *relS* which casts a set of mappings (i.e. field names) to a relation. $r|m$ is the restriction of the relation r w.r.t. the boolean mapping m.

We will now explain the Hoare logic assertions shown in §7. The precondition requires all nodes to be unmarked. It "remembers" the initial value of l, r and the set of nodes reachable from *root* in *iL*, *iR* and R respectively. As the postcondition we want to prove that a node is marked iff it is in R, i.e. is reachable, and that the graph structure is unchanged. To prove termination, we would need to show that there exists a loop measure that decreases with each iteration. Bornat [2] points out a possible loop measure. Since our Hoare logic implementation does not deal with termination, we prove only partial correctness.

The loop invariant is a bit more involved. Every time we enter the loop, *stack* is made up of the list of predecessor nodes starting at p, using the mapping $S\ c\ l\ r \equiv \lambda x.\ \textit{if}\ c\ x\ \textit{then}\ r\ x\ \textit{else}\ l\ x$, that returns l or r depending on the value of c (*i1*). Everything on the stack is marked (*i2*). Everything initially reachable from *root* is now reachable from t and p (*i3*). If something is reachable and unmarked, it is reachable using only unmarked nodes from t or from the r fields of nodes in the stack (we traverse l before r) (*i4*). If a node is marked, it was initially reachable (*i5*). All nodes not on the stack have their l and r fields unchanged (*i6*). *stkOk* says that for the nodes on the stack we can reconstruct their original l and r fields (*i7*). It is defined using primitive recursion:

$$\begin{array}{l}\textit{stkOk}\ c\ l\ r\ \textit{iL}\ \textit{iR}\ t\ [] = \textit{True}\\ \textit{stkOk}\ c\ l\ r\ \textit{iL}\ \textit{iR}\ t\ (p\ \#\ \textit{stk}) =\\ (\textit{stkOk}\ c\ l\ r\ \textit{iL}\ \textit{iR}\ (\textit{Ref}\ p)\ \textit{stk} \wedge\\ \ \textit{iL}\ p = (\textit{if}\ c\ p\ \textit{then}\ l\ p\ \textit{else}\ t) \wedge \textit{iR}\ p = (\textit{if}\ c\ p\ \textit{then}\ t\ \textit{else}\ r\ p))\end{array}$$

7.2 Proof of correctness

In this section we will go through part of the Isabelle/Isar proof of correctness, emphasising its readability. Although we provide additional comments, we rely on the self-explanatory nature of the Isar proof language, details of which can be found elsewhere [14, 9]. The entire proof is available at [7]. At many places in the proof a compromise was made between using automatic proof tactics when the proof looked intuitive, and manually going into the proof state when it was felt that more explanation was necessary. The entire proof is about four hundred lines of text. As far as we know, it is the shortest and most human readable, machine checkable proof of this algorithm. References to traditional proofs can be found in [2].

For automatic proofs, Isabelle is equipped with a number of proof tactics (e.g. *blast* for predicate calculus reasoning, *simp* for simplification, and *auto* for combinations of the two). In the case of lengthy invocations of these tactics, we will not show the tactic itself, but only important pre-proved lemmas used to invoke it. For every construct defined, we prove its corresponding *separation lemmas*, such as *List-update-conv* in §5.2. They are used as simplification rules wherever applicable. Proofs of these *separation lemmas* normally follow from short and simple inductive arguments. The complete proof document [7] contains all such proven simplification rules.

We first state the correctness theorem as the Hoare triple in §7 and use the Isabelle verification condition generator *vcg* to reduce it to pure HOL subgoals. We perform pattern matching on this Hoare triple to bind the invariant to *?inv c l m r t p*. The *?* before *inv* denotes that it is a schematic variable. Schematic variables are abbreviations for other terms.

Note that assertions are modelled as functions that depend on program variables. Thus substitution in an assertion is simply function application with changed parameters.

We first show that the precondition leads to the invariant. Starting from the precondition, we need to prove *?inv c l m r root Null* (i.e. *?inv c l m r t p* pulled back over the initial assignments $t := root$; $p := Null$). In our goal, since $p = Null$, the variable *stack* under the existential is the empty list. This simplifies things sufficiently, making the proof trivial enough to be omitted.

We then prove the postcondition to be true, assuming the invariant and loop termination condition hold. Variable *stack* is the empty list here as well, and the postcondition is easily shown using parts of the invariant *i4*, *i5*, and *i6*.

The bulk of the proof lies in trying to prove that the invariant is preserved. Assuming the invariant and loop condition hold, we need to show the invariant after variable substitution arising from all three arms of the algorithm. After a case distinction on the if-then-else conditions we are left with three large but similar subproofs. In this paper we will only walk through the proof of the pop arm in order to save whatever is left of the reader's interest. The pop arm serves as a good illustration as it involves the "seeing is believing" graph reconstruction step, a decrease in the length of the stack, as well as a change of the graph mapping r.

fix $c\ m\ l\ r\ t\ p\ q\ root$
let $\exists\, stack.\ ?Inv\ stack\ =\ ?inv\ c\ m\ l\ r\ t\ p$
let $\exists\, stack.\ ?popInv\ stack\ =\ ?inv\ c\ m\ l\ (r(p \to t))\ p\ (p\hat{}.r)$
assume $(\exists\, stack.?Inv\ stack) \wedge (p \neq Null \vee t \neq Null \wedge \neg\ t\hat{}.m)$ (**is** - $\wedge$ *?whileB*)

then obtain *stack* **where** *inv*: *?Inv stack* **and** *whileB*: *?whileB* **by** *blast*
let $?I1 \wedge ?I2 \wedge ?I3 \wedge ?I4 \wedge ?I5 \wedge ?I6 \wedge ?I7 = ?Inv\ stack$
from *inv* **have** *i1*: *?I1* **and** *i2*: *?I2* **and** *i3*: *?I3* **and** *i4*: *?I4*
and *i5*: *?I5* **and** *i6*: *?I6* **and** *i7*: *?I7* **by** *simp+*

Command **fix** introduces new free variables into a proof — the statement is proved for "arbitrary but fixed values". We start by dismantling the invariant and instantiating its seven conjuncts to *?*-variables by pattern matching. Commands **is** and **let** perform pattern matching and instantiate *?*-variables. Note that *?I1*, etc are merely formulae, i.e. syntax, and that the corresponding facts *i1*, etc need to be proven explicitly (from *inv* using $\wedge$-elimination). *?Inv* is the original invariant after existential elimination using the witness *stack*. *?popInv* corresponds to *?Inv* pulled back over the pop arm assignments.

We begin the pop arm proof by assuming the two if-then-else conditions and proving facts that we use later. We introduce a new variable *stack-tl* to serve as the witness for $\exists\, stack.\ ?popInv\ stack$, our goal.

assume *ifB1*: $t = Null \vee t\hat{}.m$ **and** *ifB2*: $p\hat{}.c$
from *ifB1 whileB* **have** *pNotNull*: $p \neq Null$ **by** *auto*
then obtain *addr-p* **where** *addr-p-eq*: $p = Ref\ addr\text{-}p$ **by** *auto*
with *i1* **obtain** *stack-tl* **where** *stack-eq*: $stack = (addr\ p)\ \#\ stack\text{-}tl$
by *auto*
with *i2* **have** *m-addr-p*: $p\hat{}.m$ **by** *auto*
have *stackDist*: *distinct* (*stack*) **using** *i1* **by** (*rule List-distinct*)
from *stack-eq stackDist* **have** *p-notin-stack-tl*: $addr\ p \notin set\ stack\text{-}tl$ **by** *simp*

We now prove the seven individual conjuncts of $\exists\, stack.\ ?popInv\ stack$ separately as facts *poI1* to *poI7*, which we state explicitly. Note that we could also pattern match *?popInv stack-tl* to assign these individual conjuncts to seven *?*-variables, eliminating the need to mention them explicitly. In general, it is a good idea to instantiate *?*-variables to use later in proofs. Like user defined constants in programs, it makes proofs a lot more tolerant to change and allows one to see their structure. The disadvantage is that too much pattern matching and back referencing makes the proof difficult to read.

Our first goal follows directly from our definitions, and spatial separation:

— List property is maintained:
from *i1 p-notin-stack-tl ifB2*
have *poI1*: $List\ (S\ c\ l\ (r(p \rightarrow t)))\ (p\hat{}.r)\ stack\text{-}tl$
by *addr-p-eq stack-eq S-def*
moreover

Next we have to show that all nodes in *stack-tl* are marked. This follows directly from our original invariant, where we know that all nodes in *stack* are marked.

— Everything on the stack is marked:
from *i2* **have** *poI2*: $\forall\ x \in set\ stack\text{-}tl.\ m\ x$ **by** (*simp add*:*stack-eq*)
moreover

Next we prove that all nodes are still reachable after executing the pop arm. We need the help of lemma *still-reachable* that we have proven separately:

$[\![B \subseteq Ra^* \text{ `` } A;\ \forall (x,\ y) \in Rb - Ra.\ y \in Ra^* \text{ `` } A]\!] \Longrightarrow Rb^* \text{ `` } B \subseteq Ra^* \text{ `` } A$

A little pattern matching will give us something of the form to which we can apply this lemma.

— Everything is still reachable:
let $(R = \mathit{reachable}\ ?Ra\ ?A) = ?I3$
let $?Rb = (\mathit{relS}\ \{l,\ r(p \to t)\})$
let $?B = \{p,\ p\hat{}.r\}$
— Our goal is $R = \mathit{reachable}\ ?Rb\ ?B$.
have $?Ra^* \text{ `` } \mathit{addrS}\ ?A = ?Rb^* \text{ `` } \mathit{addrS}\ ?B$ (**is** $?L = ?R$)
proof
  show $?L \subseteq ?R$
  proof (*rule still-reachable*)
    show $\mathit{addrS}\ ?A \subseteq ?Rb^* \text{ `` } \mathit{addrS}\ ?B$ **by** *relS-def oneStep-reachable*

After filling in the pattern matched variables, this last subgoal is:

$\mathit{addrS}\ \{t,\ p\} \subseteq (\mathit{relS}\ \{l,\ r(p \to t)\})^* \text{ `` } \mathit{addrS}\ \{p,\ p\hat{}.r\}$

and is true as p can be reached by reflexivity, and t by a one step hop from p. The second subgoal generated by *still-reachable* is:

$\forall (x,\ y) \in \mathit{relS}\ \{l,\ r\} - \mathit{relS}\ \{l,\ r(p \to t)\}.\ y \in (\mathit{relS}\ \{l,\ r(p \to t)\})^* \text{ `` } \mathit{addrS}\ \{p,\ p\hat{}.r\}$

and can be seen to be true as if any such pair (x,y) exists, it has to be $(p,\ p\hat{}.r)$:

    show $\forall (x,y) \in ?Ra - ?Rb.\ y \in (?Rb^* \text{ `` } \mathit{addrS}\ ?B)$ **by** *relS-def addrS-def*
  qed

The other direction of $?L = ?R$ can be shown to be correct by similar arguments and is proven by appropriately instantiated automatic proof tactics.

  show $?R \subseteq ?L$ — Proof hidden
qed
with *i3* **have** *poI3*: $R = \mathit{reachable}\ ?Rb\ ?B$ **by** (*simp add:reachable-def*)
moreover

The proof for the next part of the invariant is a bit more indirect.

— If it is reachable and not marked, it is still reachable using...
let $\forall x.\ x \in R \wedge \neg\ m\ x \longrightarrow x \in \mathit{reachable}\ ?Ra\ ?A = ?I4$
let $?Rb = \mathit{relS}\ \{l,\ r(p \to t)\} \mid m$
let $?B = \{p\} \cup \mathit{set}\ (\mathit{map}\ (r(p \to t))\ \mathit{stack\text{-}tl})$
— Our goal is $\forall x.\ x \in R \wedge \neg\ m\ x \longrightarrow x \in \mathit{reachable}\ ?Rb\ ?B$.
let $?T = \{t,\ p\hat{}.r\}$

Assuming we have an x that satisfies $x \in R \wedge \neg\ m\ x$, we have $x \in \mathit{reachable}\ ?Ra\ ?A$ (from *i4*). What we need is $x \in \mathit{reachable}\ ?Ra\ ?B$. Examining these two sets, we see that their difference is $\mathit{reachable}\ ?Rb\ ?T$, which is the set of elements removed from $\mathit{reachable}\ ?Ra\ ?A$ as a result of the pop arm. We therefore do the proof in two stages. First we prove the subset with difference property, and then show that this fits with what happens in the pop arm.

have $?Ra^* \text{ `` } \mathit{addrS}\ ?A \subseteq ?Rb^* \text{ `` } (\mathit{addrS}\ ?B \cup \mathit{addrS}\ ?T)$
  — Proof hidden; similar to previous use of *still-reachable*
— We now bring a term from the right to the left of the subset relation.
hence *subset*: $?Ra^* \text{ `` } \mathit{addrS}\ ?A - ?Rb^* \text{ `` } \mathit{addrS}\ ?T \subseteq ?Rb^* \text{ `` } \mathit{addrS}\ ?B$
  by *blast*

have $poI4$: $\forall x.\ x \in R \wedge \neg\ m\ x \longrightarrow x \in reachable\ ?Rb\ ?B$
proof
  fix x **assume** a: $x \in R \wedge \neg\ m\ x$
  — First, a disjunction on $p\hat{}.r$ used later in the proof
  have $pDisj$:$p\hat{}.r = Null \vee (p\hat{}.r \neq Null \wedge p\hat{}.r\hat{}.m)$ **using** $poI1\ poI2$
    by $auto$
  — x belongs to the left hand side of $subset$:
  have $incl$: $x \in ?Ra^*$ ``$addrS\ ?A$ **using** $a\ i4$ **by** $reachable\text{-}def$
  have $excl$: $x \notin ?Rb^*$ `` $addrS\ ?T$ **using** $pDisj\ ifB1\ a$ **by** $addrS\text{-}def$
  — And therefore also belongs to the right hand side of $subset$,
  — which corresponds to our goal.
  from $incl\ excl\ subset$ **show** $x \in reachable\ ?Rb\ ?B$ **by** $reachable\text{-}def$
qed
moreover

Since m is unchanged through the pop arm, the next subgoal is identical to its counterpart in the original invariant.

— If it is marked, then it is reachable
from $i5$ **have** $poI5$: $\forall x.\ m\ x \longrightarrow x \in R$ **.**
moreover

The next part of the invariant is what is used to prove that the l and r are finally restored. As expected, the major part of this proof follows from $i7$, the assertion involving *stkOk*, expressing what it means for a graph to be reconstructible.

— If it is not on the stack, then its l and r fields are unchanged
from $i7\ i6\ ifB2$
have $poI6$: $\forall x.\ x \notin set\ stack\text{-}tl \longrightarrow (r(p \rightarrow t))\ x = iR\ x \wedge l\ x = iL\ x$
  by $addr\text{-}p\text{-}eq\ stack\text{-}eq$
moreover

The last part of the invariant involves the *stkOk* predicate. The only thing the pop arm changes here is the r mapping at p. The goal is automatically proven using the following simplification rule:

$x \notin set\ xs \Longrightarrow$
$stkOk\ c\ l\ (r(x := g))\ iL\ iR\ (Ref\ x)\ xs = stkOk\ c\ l\ r\ iL\ iR\ (Ref\ x)\ xs$

— If it is on the stack, then its l and r fields can be reconstructed
from $p\text{-}notin\text{-}stack\text{-}tl\ i7$ **have** $poI7$: $stkOk\ c\ l\ (r(p \rightarrow t))\ iL\ iR\ p\ stack\text{-}tl$
  by $stack\text{-}eq\ addr\text{-}p\text{-}eq$

The proof of the pop arm was in the style of an Isabelle "calculation", with **have** statements separated by **moreover**, which can **ultimately** be put together to show the goal at hand. At this point we have proved the individual conjuncts of *?popInv stack-tl*. We will now piece them together and introduce an existential quantifier, thus arriving exactly at what came out of the verification condition generator:

ultimately show *?popInv stack-tl* **by** *simp*
qed
hence $\exists$ *stack. ?popInv stack* **..**

We similarly prove preservation of the invariant in the swing and push arms and combine these results to complete the proof.

References

1. Richard Bornat. Proofs of pointer programs in Jape. http://www.dcs.qmul.ac.uk/~richard/pointers/.
2. Richard Bornat. Proving pointer programs in Hoare Logic. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lect. Notes in Comp. Sci.*, pages 102–126. Springer-Verlag, 2000.
3. Richard Bornat and Bernard Sufrin. Animating formal proofs at the surface: the Jape proof calculator. *The Computer Journal*, 43:177–192, 1999.
4. Rod Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.
5. M.C.J. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.
6. Jacob L. Jensen, Michael E. Joergensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI '97*, 1997.
7. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. http://www.in.tum.de/~nipkow/pubs/cade03.html.
8. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
9. Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lect. Notes in Comp. Sci.*, pages 259–278. Springer-Verlag, 2003.
10. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.
11. John C. Reynolds. Intuitionistic reasoning about shared mutable data structures. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millenial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.
12. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, 2002.
13. Norihisa Suzuki. *Automatic Verification of Programs with Complex Data Structures*. PhD thesis, Stanford University, 1976. Garland Publishing, 1980.
14. Markus Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.
15. H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, Urbana-Champaign, 2001.