Reasoning about Mutable Data Structures in First-Order Logic with Arithmetic: Lists and Binary Trees

Fangzhen Lin
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Bo Yang
School of Electronic and Communication Engineering
Guiyang University, Guiyang, China

April 2015

(DRAFT)

Abstract

We describe a method for translating programs with loops and mutable data structures to first-order theories with quantifiers over natural numbers. We illustrate by considering a list reversing program and a version of the Deutsch-Schorr-Waite marking algorithm on binary trees. We show how the resulting first-order theories can be used to reason about properties of the programs by induction, without the need of loop invariants.

1 Introduction

How to reason about mutable data structures is a challenging problem in formal methods, and there has been much work about it (e.g. [13, 22, 9, 3, 7, 4, 18, 19, 12]). In this paper, we show how this can be done in a simple and natural way in first-order logic by quantifying over natural numbers. Our approach is based on a recent proposal by Lin [10] for translating a program to a first-order theory with quantifiers over natural numbers. One key feature of this approach is that it does not rely on loop invariants [8]. Once translated to a first-order theory, properties about the program can then be proved using induction and other methods. Lin showed how this method works for Algol-like programs with assignments on program variables with values such as integers. In this paper we show how it can work with assignments that involve singly linked lists and binary trees implemented by C-like `struct`

data structures. We consider two benchmark programs on mutable data structures: an in-place reversal of a list, and Deutsch-Schorr-Waite (DSW) marking algorithm on binary trees. We show how the correctness of these two programs can be stated and proved in first-order logic. The proof of the correctness of the in-place reversing of an acyclic list is not difficult in Hoare-style logic like separation logic. However, for cyclic lists, it is much more involved and we have not seen anyone giving a loop invariant for this program. For the DSW algorithm, Bornat [1] claimed that "(it) is the first mountain that any formalism for pointer analysis should climb." Reynolds [18] mentioned that "the most ambitious application of separation logic has been Yangs proof of the Schorr-Waite algorithm for marking structures that contain sharing and cycles" [23]. In 2001, Bornat did a proof with the help of the Jape proof editor which took 152 pages[1]. More recently, Loginov *et. al.* [12] used an abstract-interpretation system called TVLA to automate a proof of the correctness of the DSW algorithm on binary trees. In this paper, we follow Loginov *et. al.* [12] to consider binary trees as well, to illustrate how our approach works on tree structures. The general idea should work for graph structures as well as our proof is based on induction similar to the manual proof by Gries [5]. In short, our proofs do not need complicated loop invariants. In fact, they are very much direct formalizations in first-order logic how one would prove the correctness of these algorithms in natural language.

This paper is organized as follows. Section 2 defines the class of programs that we are considering. Section 3 describes the translation from these programs to first-order theories. Section 4 considers the list reversing program, and section 5 the DSW marking algorithm on binary trees. Section 6 discusses related work and Section 7 concludes the paper.

2 Programs

We consider programs that manipulate mutable data structures, in particular singly linked lists and binary trees:

```
struct List {                  struct Tree {
  int data;                      int data;
  struct List *next;             struct Tree *left;
}                                struct Tree *left; }
```

Notice that the above `struct` datatype `Tree` in fact defines directed graphs with at most two outgoing edges for each node. Some restrictions need to be put on these graphs for them to be trees.

In context-free grammar, the class of programs that we consider in this paper can be defined as follows:

[1]R. Bornat. Proofs of pointer programs in Jape. Available at `http://www.dcs.qmul.ac.uk/erichard/pointers/`.

```
E ::= operator(E,...,E) | data(List) | data(Tree)
B ::= E = E | boolean-op(B,...,B)
List ::= list-name | next(List)
Tree ::= tree-name | left(Tree) | right(Tree)
P ::=  List = null | List = List |
       data(List) = E | data(Tree) = E |
       Tree = null | Tree = Tree |
       if B then P else P |
       P; P | while B do P
```

where "`list-name`" and "`tree-name`" are tokens denoting program variables of the respective types.

For example, the following program P_{rev} reverses a given list I:

```
J = null;
while I != null do {
  K = next(I);
  next(I) = J;
  J=I;
  I=K;
  }
I=J;
```

3 Translating programs to first-order theories

3.1 Preliminaries

We consider a state of a program (before, during, and after its execution) to be a first-order structure. The semantics of a program is thus a theory about how these states are related to each other. For instance, for a sequential program that is supposed to terminate, one may be interested in only the relationship between the beginning and terminating states of the program. This relationship can be axiomatized in many ways, for example, by Hoare's triples [8], modal logic such as dynamic logic [6], or situation calculus [14]. In this paper we axiomatize it in first-order logic by using different vocabularies to describe different states.

We will use a many-sorted first-order language to model a state. For the programs considered in this paper, we assume the following sorts: Nat for natural numbers, $List$ for lists, and $Tree$ for trees. We assume *null* (no value) is a special value in $List$ and $Tree$. We assume that the domain for Nat is the set of natural numbers and the usual constant symbols such as 0 and 1, and the usual operators like $+$ and $-$ have their intended interpretations.

This many-sorted language will be our *base* language. In the following, given a set $\mathcal{D}$ of axioms, we use $\mathcal{D} \models_{Nat} \varphi$ to mean that the formula φ is true in all models of $\mathcal{D}$ that interpret Nat as described above.

Given a program, for each identifier in it, we extend the base language by a new constant of appropriate sort, typically written in a string of letters beginning with an upper case one like X, to denote its value when the program starts. Its value at the end of the program will be denoted by the same string but "primed", for example, X'. Its values during the execution of the program will be denoted by unique names. If it occurs inside a loop, then we will extend it by a new natural number argument, like $X(n)$, to keep track of its values during the iterations of the loop. How this is done will become apparent later, and is one of the keys in our method.

To model lists and trees, we also extend our base language by the following functions: $data : List \cup Tree \to Nat$, $next : List \to List$, $left : Tree \to Tree$, $right : Tree \to Tree$. These functions are used to define lists and trees, and are dynamic. Thus if a is a list, then

$$data(a) = 1 \wedge data'(a) = 2$$

means that for the list a, its data value is 1 when the program starts, but becomes 2 when the program terminates. Similar to program variables, we may extend these functions with additional natural number arguments to model the changes of lists during loops. Thus when we write $next(x, n)$, the first argument x must be of sort $List$, and the second n of sort Nat.

To simplify the presentation of our formulas, we omit the sort information when it is apparent from the context. For example, the formula $\exists x.x = next(x)$ stands for $\exists(x : List)x = next(x)$ because the first argument of $next$ must be of sort $List$.

We will reserve i, j, k, n and m for variables ranging over sort Nat (natural numbers). Thus $\forall n \exists x.x = next(x, n)$ stands for

$$\forall(n\!:\!Nat)\exists(x\!:\!List)x = next(x, n).$$

In the following, we use a shorthand (macro) to say that e is the smallest natural number that satisfies $\varphi(n)$:

$$smallest(e, n, \varphi)$$

stands for the following formula:

$$\varphi(n/e) \wedge \forall m.m < e \to \neg\varphi(n/m),$$

where n is a natural number variable in φ, m a new natural number variable not in e or φ, $\varphi(n/e)$ the result of replacing n in φ by e, similarly for $\varphi(n/m)$. For example, $smallest(M, k, k < N \wedge found(k))$ says that M is the smallest natural number such that $M < N \wedge found(M)$:

$$M < N \wedge found(M) \wedge \forall n.n < M \to \neg(n < N \wedge found(n)).$$

Some useful properties about this macro can be found in the paper [10].

3.2 The translation

With the many sorted first-order language described above, we now describe our translation from programs to first-order theories.

Let P be a program, $\vec{L}$ a set of list variables and $\vec{T}$ a set of tree variables. We assume that all variables used in the program P are in $\vec{L} \cup \vec{T}$. We construct a set of axioms according to the procedure described in [10]. This set of axioms, denoted by $A(P; \vec{L}, \vec{T})$, can be constructed to record the changes of program variables during the execution of the program. However, in this paper we focus only on how the output of a program. This set of axioms is constructed inductively according to the structure of P.

1. If P is `L = L1`, where `L` is a program variable in $\vec{L}$, and `L1` a list expression or null, then $A(P; \vec{L}, \vec{T})$ consists of following axioms:

$$\begin{aligned} L' &= L_1, \\ X' &= X, \quad X \in \vec{L} \cup \vec{T} \text{ and } X \text{ different from } L \\ \xi'(x) &= \xi(x), \quad \xi \in \{data, next, left, right\}, \end{aligned} \tag{1}$$

   where free variables are universally quantified from the outside over the appropriate sorts. Notice that (1) generates four equations:

$$\begin{aligned} &\forall x.data'(x) = data(x), \quad \forall x.next'(x) = next(x), \\ &\forall x.left'(x) = left(x), \quad \forall x.right'(x) = right(x). \end{aligned}$$

2. If P is `next(L) = L1`, where `L` is a list expression and `L1` is a list expression or *null*, then $A(P; \vec{L}, \vec{T})$ consists of following axioms:

$$\begin{aligned} X' &= X, \quad X \in \vec{L} \cup \vec{T} \\ next'(x) &= \text{ if } x = L \text{ then } L_1 \text{ else } next(x), \\ \xi'(x) &= \xi(x), \quad \xi \in \{data, left, right\}, \end{aligned}$$

   where, in general, the conditional expression

$$e_1 = \text{ if } \varphi \text{ then } e_2 \text{ else } e_3$$

   is a shorthand for the following formula:

$$\forall \vec{x}[(\varphi \to e_1 = e_2) \wedge (\neg\varphi \to e_1 = e_3)],$$

   where $\vec{x}$ are the free variables in the conditional expression.

3. If P is `data(L) = e`, where `L` is a list expression and `e` an integer expression, then $A(P; \vec{L}, \vec{T})$ consists of following axioms:

$$\begin{aligned} X' &= X, \quad X \in \vec{L} \cup \vec{T}, \\ data'(x) &= \text{ if } x = L \text{ then } e \text{ else } data(x), \\ \xi'(x) &= \xi(x), \quad \xi \in \{next, left, right\}. \end{aligned}$$

4. The cases for the statements on trees are similar, and we omit them here.

5. If P is `if B then P1 else P2`, then $A(P; \vec{L}, \vec{T})$ is constructed from $A(P_1; \vec{L}, \vec{T})$ and $A(P_2, \vec{L}, \vec{T})$ as follows:

$$\begin{aligned} &B \to \varphi, \text{ for each } \varphi \in A(P_1; \vec{L}, \vec{T}), \\ &\neg B \to \varphi, \text{ for each } \varphi \in A(P_2; \vec{L}, \vec{T}). \end{aligned}$$

6. If P is `P1; P2`, then $A(P; \vec{L}, \vec{T})$ is constructed from $A(P_1; \vec{L}, \vec{T})$ and $A(P_2; \vec{L}, \vec{T})$ by connecting the outputs of P_1 with the inputs of P_2 as follows: get a new constant X_{new} for each $X \in \vec{L} \cup \vec{T}$, and new functions $data_{new}$, $next_{new}$, $left_{new}$ and $right_{new}$, and generate the following axioms:

$$\begin{aligned} &\varphi(Output/New), \text{ for each } \varphi \in A(P_1; \vec{L}, \vec{T}), \\ &\varphi(Input/New), \text{ for each } \varphi \in A(P_2; \vec{L}, \vec{T}), \end{aligned}$$

   where

   - $\varphi(Output/New)$ is the result of replacing in φ each occurrence of $data'$ by $data_{new}$, $next'$ by $next_{new}$, $left'$ by $left_{new}$, $right'$ by $right_{new}$, and X' by X_{new}, where $X \in \vec{L} \cup \vec{T}$;
   - $\varphi(Input/New)$ is the result of replacing in φ each occurrence of $data$ by $data_{new}$, $next$ by $next_{new}$, $left$ by $left_{new}$, $right$ by $right_{new}$, and X by X_{new}, where $X \in \vec{L} \cup \vec{T}$.

   We assume that, by renaming if necessary, $A(P_1; \vec{L}, \vec{T})$ and $A(P_2, \vec{L}, \vec{T})$ have no common temporary function names.

7. If P is `while B do P1`, Then $A(P; \vec{L}, \vec{T})$ is constructed by adding an index parameter n to every function in $A(P_1; \vec{L}, \vec{T})$ to record its value after the body P_1 has been executed for n times. Notice that P_1 may contain while loops, so some index parameters n_i may have already been added to these functions. Formally, it consists of the

following axioms:

$$\begin{array}{l}
\varphi[n], \text{ for each } \varphi \in A(P_1; \vec{L}, \vec{T}), \\
X = X(0), \text{ for each } X \in \vec{L} \cup \vec{T}, \\
\xi(x) = \xi(x, 0), \quad \xi \in \{data, next, left, right\}, \\
smallest(N, n, \neg B[n]), \\
X' = X(N), \text{ for each } X \in \vec{L} \cup \vec{T}, \\
\xi'(x) = \xi(x, N), \quad \xi \in \{data, next, left, right\},
\end{array}$$

where n is a new natural number variable not already in φ, N a new constant not already used in $A(P_1; \vec{L}, \vec{T})$ and denoting the number of iterations for the loop to terminate, and for each formula or term α, $\alpha[n]$ denotes the value of α after the body P_1 has been executed n times, and is obtained from α by performing the following recursive substitutions:

- for each non-primed function X not in the base language, replace every occurrence of $X(e, n_1, ..., n_k)$ in α by $X(e[n], n_1, ..., n_k, n)$. Notice that this replacement is for every non-primed function X, including those that are not in $\vec{L} \cup \vec{T}$, but introduced during the construction of $A(P_1; \vec{L}, \vec{T})$.
- for each X in $\vec{L} \cup \vec{T} \cup \{data, next, left, right\}$, replace all occurrences of $X'(e, n_1, ..., n_k)$ by
  $X(e[n], n_1, ..., n_k, n + 1)$.

For examples, $(L_1 = L_2)[n]$ is $L_1(n) = L_2(n)$, $(1+data'(L))[n]$ is $1+data(L(n), n+1)$, and $next(L(m), m)[n]$ is $next(L(m, n), m, n)$.

The above axioms assume that no runtime errors such as *null* access and divide by 0 will occur. This can be handled by adding some more axioms. For example, for the statement `L = L1`, where `L` and `L1` are list expressions, we can add the axiom $I \neq null$ for every list expression I such that `next(I)` is a sub-expression of `L` or `L1`.

These axioms can be used to prove both partial and total correctness of a program. For partial correctness, one proves some properties without establishing the existence of the new constants N in the axioms for loops, which means that the theory may not be consistent for some inputs. For total correctness, one also shows that for each loop, the new constant N indeed exists. For example, the theory for `while true do {}` is inconsistent. However, the theory for `while L=null do {}` is consistent but inconsistent with $L = null$, meaning that it in fact entails that $L \neq null$. To avoid inconsistency coming from non-terminating loops, we can add a guard to loop axioms, like

$$(\exists n \neg B[n]) \rightarrow smallest(N, n, \neg B[n]).$$

Notice that we write $A(P; \vec{L}, \vec{T})$ as if there is a unique set of axioms constructed using the above procedure. As one can see, it is unique only up to the choice of temporary functions when constructing axioms for sequences $P_1; P_2$. Given that in this paper we only care about the input/output behavior of a program, we will eliminate these temporary functions whenever possible. For example, according to our construction above, the axiom set $A(P; (A, B), \emptyset)$ for the program P: `next(A) = B; next(B) = null` is:

$$
\begin{aligned}
&A_1 = A \wedge B_1 = B,\\
&next_1(x) = \text{if } (x = A) \text{ then } B \text{ else } next(x),\\
&\xi_1(x) = \xi(x), \quad \xi \in \{data, left, right\},\\
&A' = A_1 \wedge B' = B_1,\\
&next'(x) = \text{if } (x = B_1) \text{ then } null \text{ else } next_1(x),\\
&\xi'(x) = \xi_1(x), \quad \xi \in \{data, left, right\},
\end{aligned}
$$

where $A_1, B_1, next_1, data_1, left_1, right_1$ are temporary functions. We can eliminate these temporary functions and use the following axioms for $A(P; (A, B), \emptyset)$:

$$
\begin{aligned}
&A' = A \wedge B' = B,\\
&next'(x) = \text{if } (x = A) \text{ then } B \text{ else}\\
&\qquad\qquad \text{if } (x = B) \text{ then } null \text{ else } next(x),\\
&\xi'(x) = \xi(x), \xi \in \{data, left, right\}.
\end{aligned}
$$

We will do these simplifications whenever possible to get simpler sets of axioms.

In addition to the axioms that we obtain from a program, we may also need to have some general axioms about the domain that the program is about. For lists and trees, we need to assume that they are finite. Furthermore, as we shall see, it is convenient to introduce some additional functions and predicates besides *data*, *next*, *left* and *right* that are directly manipulated by the programs.

3.3 Axioms about lists

First of all, while there can be an infinite number of lists, we assume that each of them is finite, meaning that for each list, the chain of the *next* links will either terminate at *null* or loop in a finite number of steps. Given that we have natural numbers in our language, an easy way to formalize this is to represent lists, effectively, as arrays: let $item(n, x)$ denote the nth list down the *next* link from x:

$$item(0, x) = x, \tag{2}$$

$$item(n + 1, x) = next(item(n, x)), \tag{3}$$

$$item(n, null) = null, \tag{4}$$

then the assumption that each list be finite can captured by the following axiom:

$$\forall(x\!:\!List)\exists m,n.m<n\wedge item(m,x)=item(n,x). \tag{5}$$

If $item(m,x)=item(n,x)=null$, then list x is non-cyclic and has at most m nodes. If $item(m,x)=item(n,x)\neq null$, then x has a cycle and has at most n distinct nodes.

Notice that (4) is equivalent to $next(null)=null$. The correctness of a program requires that $null$ never be accessed. We postulate that $next(null)=null$ so that cyclic and non-cyclic lists can be handled uniformly.

In the following, let $\mathcal{D}_{list}$ be the set of axioms (2) - (5). It is easy to show that by (5), each list has a smallest index at which it either terminates at $null$ or starts to loop ($\exists!m$ stands for that there is a unique m):

$$\mathcal{D}_{list}\models_{Nat}\forall x\exists!m\exists!n.loop(m,n,x), \tag{6}$$

where $loop(m,n,x)$ is a shorthand for the following formula:

$$\begin{aligned}&m<n\wedge item(m,x)=item(n,x)\wedge\\&\forall i,j(0\leq i<n\wedge 0\leq j<n\wedge i\neq j\rightarrow item(i,x)\neq item(j,x)).\end{aligned}$$

For example, $loop(0,1,null)$. If $a\rightarrow b\rightarrow null$, then $loop(2,3,a)$. If $a\rightarrow a$, then $loop(0,1,a)$. If $a\rightarrow b\rightarrow c\rightarrow b$, then $loop(1,3,a)$. If $loop(m,n,x)$ and $item(m,x)=null$, then $n=m+1$ and there are m nodes in x. On the other hand, if $loop(m,n,x)$ but $item(m,x)\neq null$, then there are n nodes in x.

Notice that $item(n,x)$ is defined using $next$, which defines the lists at the input. We may also need a corresponding $item$ function to describe lists at the output or at a point during the program execution. We will use a similar notation as we have done with $next$. For example, at the output, we'll use $item'(n,x)$ which is defined via $next'(x)$, at the end of the mth iteration of a loop, we'll use $item(n,x,m)$ which is defined via $next(x,m)$, and so on. When these functions are used, the corresponding axioms (2) - (4) will be added to $\mathcal{D}_{list}$.

3.4 Axioms about binary trees

As in the case of lists, while the domain of $Tree$ can be infinite, each tree needs to be finite. Again using natural numbers, an easy way to do this is by inductive definition. Let $tree(n,x)$ denotes that x is a binary tree with at most n levels:

$$tree(0,x)\equiv x=null, \tag{7}$$

$$tree(n+1,x)\equiv[tree(n,x)\vee(tree(n,left(x))\wedge tree(n,right(x)))]. \tag{8}$$

We then postulate that every tree must be finite:

$$\forall(x{:}Tree)\exists n.tree(n,x). \tag{9}$$

Notice that if $tree(n,x)$ for some n, then x cannot contain a loop. But it does not rule out node sharing such as $left(b) = right(b) = a \neq null$. To rule out node sharing we add the following axioms:

$$left(x) = right(y) \rightarrow left(x) = null, \tag{10}$$
$$left(x) = left(y) \rightarrow [left(x) = null \vee x = y], \tag{11}$$
$$right(x) = right(y) \rightarrow [right(x) = null \vee x = y]. \tag{12}$$

To describe properties about trees, it is convenient to have a predicate representing the transitive closure of $left$ and $right$. We use $desc(x,y)$ to denote that y is a descendant of x:

$$desc(null,x) \equiv x = null, \tag{13}$$
$$desc(x,y) \equiv [x = y \vee desc(left(x),y) \vee desc(right(x),y)]. \tag{14}$$

It follows that $\forall x.desc(x,null)$. To prove properties about trees, we can do induction on either the levels of trees or the numbers of nodes in trees. Here we use the numbers of nodes and define the following count function:

$$count(null) = 0, \tag{15}$$
$$count(x) = count(left(x)) + count(right(x)) + 1. \tag{16}$$

In the following we denote by $\mathcal{D}_{tree}$ the set of axioms about trees that we have so far: (7) - (16).

Like the case for lists, the new predicates and functions introduced can have variants during the execution of a program. For instance, just as we use $left(x,n)$ and $right(x,n)$ to denote the two children pointers of x during the execution of a loop, we can have a corresponding $desc(x,y,n)$:

$$desc(null,x,n) \equiv x = null,$$
$$desc(x,y,n) \equiv x = y \vee desc(left(x,n),y,n) \vee desc(right(x,n),y,n).$$

When these functions and predicates are used, the corresponding axioms (13) - (16) will be added to $\mathcal{D}_{tree}$. Notice that no variants of (7) - (12) need to be added because they are used to ensure that when a program starts, all trees are finite and proper.

4 List reversing program

Now consider the list reversing program P_{rev} given in Section 2. We consider list variables $\vec{L} = (I, J, K)$ and omit tree variables and tree functions $left$ and $right$ as there are no trees in this program. We also omit $data$ as it is not used. They will come out as frame axioms like $data'(x) = data(x)$ anyway. The axioms for the body of the while loops are:

$$\begin{aligned}
&I' = next(I),\\
&J' = I,\\
&K' = next(I),\\
&next'(x) = \text{ if } x = I \text{ then } J \text{ else } next(x).
\end{aligned}$$

The axioms for the while loop are then:

$$\begin{aligned}
&I(0) = I \wedge J(0) = J \wedge K(0) = K,\\
&next(x, 0) = next(x),\\
&I(n+1) = next(I(n), n),\\
&J(n+1) = I(n),\\
&K(n+1) = next(I(n), n),\\
&next(x, n+1) = \text{ if } x = I(n) \text{ then } J(n) \text{ else } next(x, n),\\
&smallest(N, n, I(n) = null),\\
&I' = I(N) \wedge J' = J(N) \wedge K' = K(N),\\
&next'(x) = next(x, N),
\end{aligned}$$

where N is a natural number constant denoting the number of times that the body of the loop is performed. Of course, N depends on the values of the inputs, I, J, and K. By expanding the smallest macro, and connecting the input and output of the while loop with the output of the first assignment and the input of the last assignment in the program, respectively, we have the following axioms for the P_{rev}:

$$I(0) = I \wedge J(0) = null \wedge K(0) = K, \tag{17}$$
$$next(x, 0) = next(x), \tag{18}$$
$$I(n+1) = next(I(n), n), \tag{19}$$
$$J(n+1) = I(n), \tag{20}$$
$$K(n+1) = next(I(n), n), \tag{21}$$
$$next(x, n+1) = \text{ if } x = I(n) \text{ then } J(n) \text{ else } next(x, n), \tag{22}$$
$$I(N) = null \wedge \forall n(n < N \rightarrow I(n) \neq null), \tag{23}$$
$$I' = J(N) \wedge J' = J(N) \wedge K' = K(N), \tag{24}$$
$$next'(x) = next(x, N). \tag{25}$$

Now consider proving some properties about this program. Informally, this program does the following: if the input list I is not cyclic, then it will be reversed, with I points to the new head (the last node of I at the input). If I is cyclic: $I = I_0 \to I_1 \to \cdots \to I_k \to \cdots \to I_k$, then at the output, I points to the same node, the nodes in between I_0 and I_k are not changed (they were first reversed when traveling from I_0 to I_k and then reversed back when traveling from I_k to I_0), but the cyclic section from I_k back to I_k will be reversed. For example, if I initially points to the list: $a \to b \to c \to d \to e \to c$, then after the program terminates, it points to $a \to b \to c \to e \to d \to c$.

Using $item(n, x)$, the main property of the list reversing program can then be specified as follows:

$$I = null \to I' = null, \tag{26}$$

$$loop(m, n, I) \wedge item(m, I) = null \to [I' = item(m-1, I) \wedge next'(I) = null \wedge \forall n.0 < i \leq m \to next'(item(i, I)) = item(i-1, I)], \tag{27}$$

$$loop(m, n, I) \wedge item(m, I) \neq null \to I' = I \wedge \forall i.0 \leq i < m \to item(i, I) = item'(i, I) \wedge \forall i.m < i \leq n \to next'(item(i, I)) = item(i-1, I). \tag{28}$$

Notice $item(i, I) = item'(i, I)$ means that for the list pointed to by I at the beginning of the program, its ith item at the beginning of the program is the same as its ith item at the end of the program. This is different from $item(i, I) = item'(i, I')$ which would be about two different lists I and I': the values of the program variable `I` at the beginning and the end of the program, respectively. Similarly, $next'(item(i, I)) = item(i-1, I)$ says that for the list pointed to by I at the beginning of the program, when the program ends, the next pointer of its ith item points to its $(i-1)$th item.

Let $\mathcal{D}_{rev}$ be the set of axioms (17) - (25) about the list reversing program. We then need to show that

$$\mathcal{D}_{rev} \cup \mathcal{D}_{list} \models_{Nat} (26) \wedge (27) \wedge (28).$$

Whether this can be proved automatically with one of the current theorem provers is an interesting question (see e.g. [21] for a list of provers, some of them designed for proving properties of computer programs.) In the following, we outline a proof whose steps can be verified using many of the provers. It follows closely the informal descriptions about the program given above, by computing closed-form solutions of $I(i)$ and $next(x, i)$ using recurrences (19) - (22).

By (6), suppose $loop(m, n, I)$ for some $m < n$. Then for $i < n$, we can compute $I(i)$ and $next(x, i)$ by induction on i using (19) - (22): if $0 < i < n$, then $I(i) = item(i, I)$ and

$$next(x, i) = \begin{cases} null, & \text{if } x = item(0, I) \\ item(j-1, I), & \text{if } x = item(j, I) \text{ for some } 0 < j < i \\ next(x), & \text{otherwise} \end{cases}$$

These are enough to prove the cases (26) and (27) when I is non-cyclic: if $item(m, I) = item(n, I) = null$, then $n = m + 1$. Thus N in D_{rev}, which denotes the number of iterations of the while loop, equals m. This shows also that the program terminates when I is non-cyclic.

For proving (28) when I is cyclic, we'll need to determine the values of $I(i)$ and $next(x, i)$ when $i \geq n$: if $loop(m, n, I)$ and $item(m, I) \neq null$, then

1. $I(k) = item(m + n - k, I)$ for any $n \leq k \leq m + n$;
2. $next(item(0, I), n) = null$;
3. $next(item(i, I), n) = item(i - 1, I)$ for any $0 < i < n$;
4. $next(item(0, I), k) = null$ for any $n < k \leq m + n$;
5. $next(item(0, I), m + n + 1) = next(I)$;
6. $next(item(i, I), k) = item(i - 1, I)$ for any $m < i \leq n$ and $n < k \leq m + n + 1$;
7. $next(item(i, I), k) = item(i-1, I)$ for any $0 < i < m+n+1-k$ and $n < k \leq m+n+1$;
8. $next(item(i, I), k) = item(i+1, I)$ for any $m+n+1-k \leq i < m$ and $n < k \leq m+n+1$.

These can again be proved by using recurrences (19) - (22). From these, it will follow that $N = m + n + 1$, thus (28).

Of course, the question is how one is expected to come up with closed-form solutions for $I(i)$ and $next(x, i)$. In general, this is a difficult task, but there are many techniques that we can use, see [16]. For this example, it is easy to come up with them by tracing (19) - (22) for some small examples of I.

5 DSW marking algorithm

We now consider Deutsch-Schorr-Waite (DSW) algorithm for marking nodes in a graph [20]. While the algorithm works on general graphs, we consider Lindstrom's version [11] on trees. Our code in Figure 1 is adapted from [12]. When translated to a first-order theory using the method described above, we obtain the following axioms (we omit axioms about lists and *data* as this program does not mention lists and does not change the *data* field of any of the nodes):

$$Prev(0) = Sentinel \wedge Cur(0) = Root \wedge Next(0) = Next, \tag{29}$$

$$left(x, 0) = left(x) \wedge right(x, 0) = right(x), \tag{30}$$

$$Prev(n + 1) = \text{if } left(Cur(n), n) = null \text{ then } null \text{ else } Cur(n), \tag{31}$$

```
Prev = Sentinel;
Cur = Root;
while (Cur != Sentinel) {
  Next = left(Cur);
  left(Cur) = right(Cur);
  right(Cur) = Prev;
  Prev = Cur;
  Cur = Next;
  if (Cur == null) {
     Cur = Prev;
     Prev = null; }
}
```

Figure 1: Deutsch-Schorr-Waite (DSW) algorithm on binary trees: Root is a tree not equal to null, and Sentinel is a node not inside Root.

$$Cur(n+1) = \text{if } left(Cur(n), n) = null \text{ then } Cur(n) \text{ else } left(Cur(n), n), \tag{32}$$
$$Next(n+1) = left(Cur(n), n), \tag{33}$$
$$left(x, n+1) = \text{if } x = Cur(n) \text{ then } right(Cur(n), n) \text{ else } left(x, n), \tag{34}$$
$$right(x, n+1) = \text{if } x = Cur(n) \text{ then } Prev(n) \text{ else } right(x, n), \tag{35}$$
$$Cur(N) = Sentinel, \tag{36}$$
$$n < N \rightarrow Cur(n) \neq Sentinel, \tag{37}$$
$$Root' = Root \wedge Sentinel' = Sentinel, \tag{38}$$
$$Prev' = Prev(N) \wedge Cur' = Cur(N) \wedge Next' = Next(N), \tag{39}$$
$$left'(x) = left(x, N) \wedge right'(x) = right(x, N), \tag{40}$$

where N is a constant of sort Nat and denotes the number of times that the body of the while loop is performed. In the following, we denote by $\mathcal{D}_{DSW}$ the set of the above axioms. The trademark property of the DSW algorithm is that it does not change the structure of the trees:

$$Root \neq null \wedge \neg desc(Root, Sentinel) \rightarrow \\ \forall x(left'(x) = left(x) \wedge right'(x) = right(x)). \tag{41}$$

We show that

$$\mathcal{D}_{DSW} \cup \mathcal{D}_{tree} \models_{Nat} (41),$$

Again we give a manual proof which follows how one would justify this algorithm informally.

The DSW algorithm is basically a recursive procedure. The key insight is that when entering the loop, if $Prev$ is not part of the original tree at Cur, then Cur will only visit nodes in the original tree before eventually swaps its value with $Prev$, and when it does, all nodes in the original tree will have been marked properly. In our language, this can be expressed as follows: for any n, if

$$Cur(n) \neq null \wedge \neg desc(Cur(n), Prev(n), n), \tag{42}$$

then for some $m > n$,

$$Cur(m) = Prev(n) \wedge Prev(m) = Cur(n), \tag{43}$$
$$\forall i.n \leq i < m \rightarrow desc(Cur(n), Cur(i), n), \tag{44}$$
$$\forall x.left(x, m) = left(x, n) \wedge right(x, m) = right(x, n). \tag{45}$$

Formally, we prove

$$\mathcal{D}_{DSW} \cup \mathcal{D}_{tree} \models_{Nat} (42) \rightarrow \exists m.m > n \wedge (43) \wedge (44) \wedge (45) \tag{46}$$

by induction on the number of nodes in $Cur(n)$ (the value of $count(Cur(n), n)$). This will also show that the program always terminates. Notice that for $count(Cur(n), n)$ to be well-defined, $Cur(n)$ must really be a tree at n. This can be verified as all trees are proper when the program starts, and they remain so at the end of each iteration of the while loop.

For the base case, suppose $count(Cur(n), n) = 1$, then $left(Cur(n), n) = null$ and $right(Cur(n), n) = null$. We can then let $m = n + 3$, because by expanding recurrences (31) - (35), we have:

$$Cur(n+2) = Cur(n+1) = Cur(n),$$
$$Cur(n+3) = Prev(n), \quad Prev(n+3) = Cur(n),$$
$$left(Cur(n), n+3) = null, \quad right(Cur(n), n+3) = null,$$
$$x \neq Cur(n) \rightarrow (left(x, n+3) = left(x, n) \wedge right(x, n+3) = right(x, n)).$$

Inductively, suppose the result holds for all n such that $count(Cur(n), n) < k$ for some $k > 1$. We show that the result holds when $count(Cur(n), n) = k$. There are four cases to consider, depending on whether $left(Cur(n), n)$ and/or $right(Cur(n), n)$ are $null$. The following is the proof for the case when
$left(Cur(n), n) \neq null$ and $right(Cur(n), n) \neq null$. The other cases are similar.

Since $left(Cur(n), n) \neq null$, we have

$$Cur(n+1) = left(Cur(n), n),$$
$$Prev(n+1) = Cur(n),$$
$$Next(n+1) = left(Cur(n), n),$$

$$
\begin{aligned}
&left(Cur(n), n+1) = right(Cur(n), n),\\
&right(Cur(n), n+1) = Prev(n),\\
&x \neq Cur(n) \rightarrow (left(x, n+1) = left(x, n) \wedge right(x, n+1) = right(x, n)).
\end{aligned}
$$

Thus we have that $\neg desc(Cur(n+1), Cur(n), n+1)$, and $count(Cur(n+1), n+1) < k$. So we can apply our inductive assumption at $n+1$: for some $m_1 > n+1$,

$$
\begin{aligned}
&Cur(m_1) = Prev(n+1) = Cur(n),\\
&Prev(m_1) = Cur(n+1) = left(Cur(n), n),\\
&n+1 \leq i < m_1 \rightarrow desc(Cur(n+1), Cur(i), n+1),\\
&left(x, m_1) = left(x, n+1) \wedge right(x, m_1) = right(x, n+1).
\end{aligned}
$$

Thus

$$
\begin{aligned}
left(Cur(m_1), m_1) &= left(Cur(m_1), n+1)\\
&= left(Cur(n), n+1)\\
&= right(Cur(n), n),\\
right(Cur(m_1), m_1) &= right(Cur(m_1), n+1)\\
&= right(Cur(n), n+1) = Prev(n).
\end{aligned}
$$

Since $right(Cur(n), n) \neq null$, we have

$$
\begin{aligned}
&Cur(m_1+1) = left(Cur(m_1), m_1) = right(Cur(n), n),\\
&Prev(m_1+1) = Cur(m_1) = Cur(n),\\
&Next(m_1+1) = left(Cur(m_1), m_1) = right(Cur(n), n),\\
&left(Cur(n), m_1+1) = left(Cur(m_1), m_1+1) = right(Cur(m_1), m_1) = Prev(n),\\
&right(Cur(n), m_1+1) = right(Cur(m_1), m_1+1) = Prev(m_1) = left(Cur(n), n),\\
&x \neq Cur(n) \rightarrow (left(x, m_1+1) = left(x, m_1) \wedge right(x, m_1+1) = right(x, m_1)).
\end{aligned}
$$

We now apply the inductive assumption on $m_1 + 1$ as $count(Cur(m_1+1), m_1+1) = count(right(Cur(n), n), m_1+1) = count(right(Cur(n), n), n) < k$. Thus there is an $m_2 > m_1 + 1$ such that

$$
\begin{aligned}
&Cur(m_2) = Prev(m_1+1) = Cur(n),\\
&Prev(m_2) = Cur(m_1+1) = right(Cur(n), n),\\
&m_1+1 \leq i < m_2 \rightarrow desc(right(Cur(n), n), Cur(i), m_1),\\
&left(x, m_2) = left(x, m_1+1) \wedge right(x, m_2) = right(x, m_1+1).
\end{aligned}
$$

So $left(Cur(m_2), m_2) = left(Cur(m_2), m_1 + 1) = left(Cur(n), m_1 + 1) = Prev(n) \neq null$. Thus we have

$$
\begin{aligned}
&Cur(m_2 + 1) = left(Cur(m_2), m_2) = Prev(n),\\
&Prev(m_2 + 1) = Cur(m_2) = Cur(n),\\
&left(Cur(n), m_2 + 1) = left(Cur(m_2), m_2 + 1) = right(Cur(m_2), m_2) =\\
&\qquad right(Cur(n), m_1 + 1) = left(Cur(n), n),\\
&right(Cur(n), m_2 + 1) = right(Cur(m_2), m_2 + 1) = Prev(m_2) = right(Cur(n), n),\\
&x \neq Cur(n) \rightarrow left(x, m_2 + 1) = left(x, m_2) \wedge right(x, m_2 + 1) = right(x, m_2).
\end{aligned}
$$

Thus

$$
\begin{aligned}
&Cur(m_2 + 1) = Prev(n), \quad Prev(m_2 + 1) = Cur(n),\\
&left(x, m_2 + 1) = left(x, n), \quad right(x, m_2 + 1) = right(x, n).
\end{aligned}
$$

So we can let $m = m_2 + 1$: the only thing left to check is the following:

$$n \leq i < m_2 + 1 \rightarrow desc(Cur(n), Cur(i), n).$$

This can be shown by dividing i into two cases: $n \leq i < m_1$ and $m_1 \leq i \leq m_2$, by the inductive assumptions on $n + 1$ and $m_1 + 1$, respectively.

6 Related work

This work extends the method in [10] to include assignments about mutable data structures which is a non-trivial task given, for example, many different attempts in extending Hoare's logic to pointers. As discussed in [10], this approach differs from Hoare's logic [8] and dynamic logic [6] in that it translates programs to classical logic rather than introduces new rules and logics for them. In particular, we do not need loop invariants. We use ordinary induction to prove properties like (46). It differs from temporal logic approach [17] in that it considers programs at their source code level and is compositional on program constructs. Furthermore, our language is more fine grained. For example, it allows sentences like $desc(Cur(n), Cur(i), n)$ (according to the tree at time n, the node pointed to by Cur at time i is a descendant of the node pointed to by Cur at time n) which are essential for proving properties about the DSW algorithm here.

Interestingly, while done independently, the axioms that we have for assignments like `L = next(K)` are very similar to those that are used to characterize similar statements in [12]. However, in [12], program semantics is defined using fixed-points.

For the list reversing program, the proof of correctness for the case when the given list is non-cyclic should be easy for most approaches. However, the case for cyclic lists could be challenging for many approaches.

For the DSW marking algorithm, Loginov *et al.* [12] gave an excellent survey of various approaches to formally proving the correctness of the algorithm. Like [23, 15], our proof here is manual. However, our proof establishes total correctness (termination) as well.

7 Conclusion remarks

We have described a translation from programs that manipulate mutable data structures with loops to first-order theories with quantifiers over natural numbers. While the translated first-order theories give a declarative and axiomatic semantics to the programs, the models of the theories are in fact transition systems. Thus this approach combines axiomatic semantics and operational semantics in a single framework, and it should not be hard to prove the "correctness" of the translation described in this paper under some operational semantics. We instead concentrate on showing the effectiveness of our approach by showing how properties about programs can be proved naturally once they are translated to first-order theories. While our proofs were done manually, key steps in them can be verified using theorem provers and systems like sage, mathematica, ACL [2] and many others [21].

We believe that by providing a translation from programs to first-order theories, we lay the foundation for a new approach to formal analysis of programs. There are many promising directions for continuing this work. We have already implemented a translator and a proof assistant based on mathematica for programs on integers. We are also working on extending our translation to richer classes of programs, in particular, those with concurrency.

References

[1] R. Bornat. Proving pointer programs in Hoare logic. In *Proc. Mathematics of Program Construction*, page 102126, 2000.

[2] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[3] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. SIGPLAN'90 Conf. Programming Languages Design and Implementation*, pages 296 – 310, 1991.

[4] R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proc. 23rd Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pages 1 – 15, 1996.

[5] D. Gries. The schorr-waite graph marking algorithm. *Acta Informatica*, pages 223–232, 1979.

[6] D. Harel. *First-Order Dynamic Logic*. Springer-Verlag: Lecture Notes in Computer Science 68, New York, 1979.

[7] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proc. SIGPLAN '92 Conf. Programming Language Design and Implementation*, pages 249 – 260, 1992.

[8] C. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, pages 576–580, 1969.

[9] H.-K. Hung and J. I. Zucker. Semantics of pointers, referencing and dereferencing with intensional logic. In *Proceedings of the Sixth Symposium on Logic in Computer Science*, pages 127–136, 1991.

[10] F. Lin. A formalization of programs in first-order logic with a discrete linear order. In *Proceedings of KR 2014*, 2014.

[11] G. Lindstrom. Scanning list structures without stacks or tag bits. *Information Processing Letters*, 2(2):4751, 1973.

[12] A. Loginov, T. Reps, and M. Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In K. Yi, editor, *Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 261–279. Springer Berlin Heidelberg, 2006.

[13] Z. Manna and R. Waldinger. Problematic features of programming languages: A situational-calculus approach. *Acta Informatica*, 16:371–426, 1981.

[14] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[15] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin Heidelberg, 2003.

[16] M. Petkovsek, H. S. Wilf, and D. Zeilberger. $A = B$. Wellesley, Mass. : A K Peters, 1996.

[17] A. Pnueli. The temporal semantics of concurrent programs. *Theor. Comput. Sci.*, 13:45–60, 1981.

[18] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[19] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 24(3):217298, 2002.

[20] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501506, 1967.

[21] F. Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[22] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. Conf. Programming Language Design and Implementation*, pages 1 – 12, 1995.

[23] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, June 2001.