The Pointer Assertion Logic Engine

Anders Møller & Michael I. Schwartzbach

BRICS
Department of Computer Science
University of Aarhus, Denmark
{amoeller,mis}@brics.dk

Abstract

We present a new framework for verifying partial specifications of programs in order to catch type and memory errors and check data structure invariants. Our technique can verify a large class of data structures, namely all those that can be expressed as *graph types*. Earlier versions were restricted to simple special cases such as lists or trees. Even so, our current implementation is as fast as the previous specialized tools.

Programs are annotated with partial specifications expressed in Pointer Assertion Logic, a new notation for expressing properties of the program store. We work in the logical tradition by encoding the programs and partial specifications as formulas in monadic second-order logic. Validity of these formulas is checked by the MONA tool, which also can provide explicit counterexamples to invalid formulas.

To make verification decidable, the technique requires explicit loop and function call invariants. In return, the technique is highly modular: every statement of a given program is analyzed only once.

The main target applications are safety-critical data-type algorithms, where the cost of annotating a program with invariants is justified by the value of being able to automatically verify complex properties of the program.

1 Introduction

We present a new contribution to the area of *pointer verification*, which is concerned with verifying partial specifications of programs that make explicit use of pointers. In practice, there is an emphasis on catching type and memory errors and checking data structure invariants.

For data-type implementations, standard type-checking systems, as in C or Java, are not sufficiently expressive. For example, the type of binary trees is identical to the one for doubly-linked lists. Both are just records with pairs of pointers, which makes the type checker fail to catch many common bugs. In contrast, pointer verification can validate the underlying data structure invariants, for instance, to guarantee that doubly-linked lists maintain their shapes after pointer manipulations. Memory errors, such as dereference of `null` pointers or dangling references, and creation of memory leaks are also beyond the scope of standard type checking.

There have been several different approaches to pointer verification, but not many that are as expressive as the one we propose in this paper. Clearly there is a trade-off between expressiveness and complexity, since less detailed analyses will be able to handle larger programs. Our approach is designed to verify a single abstract data type at a time. Since such implementations often contain intricate pointer manipulations and are trusted implicitly by programmers, they are a fair target for detailed scrutiny.

We work in the logical tradition by encoding programs and partial specifications as formulas in monadic second-order logic. Formulas are processed by the MONA tool [26, 34] which reduces them to equivalent tree automata from which it is simple to conclude validity or to extract concrete counterexamples. Translated back into the underlying programming language, a counterexample is an initial store that causes the given program to fail. Program annotations, in the form of assertions and invariants, are allowed and may prove necessary to obtain the desired degree of precision. This approach can be viewed both as lightweight program verification, since the full behavior of the program is not considered, and as heavyweight type checking, since properties well beyond the expressiveness of standard type systems can be checked.

We have reported on our approach in two earlier works. In the first we introduce the basic technique applied to linear lists [24]. In the second we provide a generalization to tree-shaped data structures and introduce a new encoding to make the analysis feasible [14]. The current paper takes a leap forward in generalizing the class of data structures that can be considered, without sacrificing precision or efficiency. Our new framework can handle all data structures that can be described as *graph types* [28]. These include data structures that are well-known from folklore or literature, such as doubly-linked lists, trees with parent pointers, threaded trees, two-dimensional range trees, and endless customized versions such as trees in which all leaves are linked in a cyclic list. Our framework is also designed to handle the common situation where a data structure invariant must be temporarily violated at some program points.

Our contributions are:

- An extension of the results in [24, 14] to the whole class of graph types;

In PLDI '01, Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation.

- a language for expressing data structures and operations along with correctness specifications;
- a full implementation exploiting intricate parts of the MONA tool to obtain an efficient decision procedure, together with a range of non-trivial examples.

To verify a data type implementation, the desired data structure is specified in an abstract notation, and the program is annotated with assumptions and assertions. It is not necessary to customize or optimize the implementation, and no proof obligations are left to be dealt with manually.

We rely on a new formal notation, *Pointer Assertion Logic (PAL)* to specify the structural invariants of graph types, to state pre- and post-conditions for procedures, and to formulate invariants and assertions that are given as hints to the system. The PAL notation is essentially a monadic second-order logic in which the universe of discourse contains records, pointers, and booleans. Programs with PAL annotations are verified with the tool PALE, the *Pointer Assertion Logic Engine*. The "secret" behind the PALE implementation is using the MONA tool to decide validity of Hoare triples based on PAL over loop-free code. Code with loops or recursion is handled by splitting it into loop-free fragments using invariants, as in classical Hoare logic. While the MONA logic has an inherent non-elementary complexity [33], we demonstrate that it can efficiently handle many real programs. Furthermore, the ability to insert assertions to break larger triples into smaller ones suggests that the overall approach is modular and thus can scale reasonably.

A framework for pointer verification, such as ours, should be evaluated on four different criteria. First, how precise is the analysis? Second, it is fast and scalable? Third, does it allow or require programs to be annotated? Fourth, which data structures can be considered and how are they described? In the following sections, we will describe a programming language that uses Pointer Assertion Logic for expression of store properties, describe the decision procedure based on Hoare logic and MONA, and through a number of experiments argue that the Pointer Assertion Logic approach provides a productive compromise between expressibility and efficiency.

A Tiny Example

Consider the type of linked lists with tail pointers, which as a graph type is expressed as:

```
type Head = {
  data first: Node;
  pointer last:
    Node[this.first<next*.[pos.next=null]>last];
}
type Node = {
  data next: Node;
}
```

The notation is explained in the following section, but intuitively the `last` pointer is annotated with a formula that constrains its destination to be the last `Node` in the list. A candidate for verification is the following procedure which concatenates two such structures:

```
proc concat(data l1,l2: Head): Head
{
  if (l1.last!=null) { l1.last.next = l2.first; }
  else { l1.first = l2.first; }
  if (l2.first!=null) { l1.last = l2.last; }
  l2.first = null;
  l2.last = null;
  return l1;
}
```

These are tedious pointer manipulations that are easy to get wrong. However, if we annotate the procedure with the pre-condition that `l1` and `l2` are not `null` and run PALE, it will in half a second report that no memory errors occur and, importantly, that the data structure invariant is maintained.

Related Work

General theorem provers, such as HOL [5], may consider the full behavior of programs but are often slow and not fully automated. Tools such as ESC [12] and LCLint [15] consider memory errors among other undesirable behaviors but usually ignore data structure invariants or only support a few predefined properties. Also, they trade soundness or completeness for efficiency and hence may flag false errors or miss actual errors.

Model checkers such as Bebop [2] and Bandera [9] abstract away the heap and only verify properties of control flow. The JPF [20] model checker verifies simple assertions for a subset of Java, but does not consider structural invariants.

The constraint solver Alloy has been used to verify properties about bounded initial segments of computation sequences [23]. While this is not a complete decision procedure even for straight-line code, it finds many errors and can produce counterexamples. With this technique, data structure invariants can be expressed in first-order logic with transitive closure. However, since it assumes computation bounds, absence of error reports does not imply a guarantee of correctness, and the technique does not appear to scale.

The symbolic executor PREfix [7] simulates unannotated code through possible executions paths and detects a large class of errors. Again, this is not a complete or sound decision procedure, and data structure invariants are not considered. However, PREfix gives useful results on huge source programs.

Verification based on static analysis has culminated with shape analysis. The goals of the shape analyzer TVLA [32, 38, 31] are closest to ours but its approach is radically different. Rather than encoding programs in logic, TVLA performs fixpoint iterations on abstract descriptions of the store. Regarding precision and speed, PALE and TVLA seem to be at the same level. TVLA can handle some data abstractions and hence reason about sorting algorithms; we show in Section 6 that we can do the same. TVLA analyzes programs with only pre- and post-conditions, where PALE often uses loop invariants and assertions. This seems like an undisputed advantage for TVLA; however, not having invariants can cause a loss in precision making TVLA falsely reject a program. Regarding the specification of new data structures we claim an advantage. Once a graph type has been abstractly described with PAL, the PALE tool is ready to analyze programs. In TVLA it is necessary to specify in three-valued logic an operational semantics for a collection of primitive actions specific to the data structure in question. Furthermore, to guarantee soundness of the analysis, this semantics should be proven correct by hand. TVLA is applicable also to data structures that are not graph types, but so far all their examples have been in that class. Unlike PALE, TVLA cannot produce explicit counterexamples when programs fail to verify.

There exists a variety of assertion languages designed to express properties of data structures, such as ADDS [21], L_r [3], and Shape Types [18]. We rely on PAL since it provides a high degree of expressiveness while still having a decision procedure that works in practice.

A drawback of our approach is that detailed, explicitly stated loop invariants often are required. The overhead of adding such annotations can be significant, so the approach is not applicable for verifying large programs. However, the most complex pointer operations often occur in *data-type implementations*, which usually have a manageable size and appear in central libraries. Thus, we primarily aim for the niche of safety-critical data-type implementations. For such programs, it is well known that the effort of constructing loop invariants is comparable to the effort of designing the data-type [19]. Once the program annotations have been added, the PALE tool can automatically decide validity. PALE works by splitting the program into disjoint fragments that are verified separately by analyzing every statement exactly once. That is, verification depends only on locally specified properties and there is no fixpoint iteration involved. In this sense, the approach is highly scalable. On the other hand, the approach relies on a decision procedure with a non-elementary complexity, so there are programs that cannot be verified in practice. The experiments described in Section 7 indicate that the annotation overhead is manageable, that the theoretical complexity is not necessarily a problem in practice, and that quite intricate properties can be expressed and verified.

2 Pointer Assertion Logic

In this section, we informally present the components of our framework. First, we describe the underlying *store model*. Second, we use the notion of *graph types* to describe data structures. Third, we employ a simple *programming language* to express data structure operations. And, finally, we use *program annotations* in the form of Pointer Assertion Logic formulas, for expressing properties of the program store.

The programming language and the annotations have been designed to be simple but at the same time as expressive as the verification technique allows. In the following, we present the framework informally and refer the reader to [35] for formal definitions. To make the expressive power of the framework lucid, we show the complete syntax instead of only describing the main ideas.

Store Model

In our model, the store consists of a *heap* and some *program variables*. The heap contains *records* whose *fields* are either *pointers* or *boolean* values. A pointer either has the value `null` or points to a record. Program variables are either *data variables* or *pointer variables*. A data variable is the root of a data structure, whereas a pointer variable may point to any record in the heap.

This is a very concrete representation. We only abstract away arithmetic values and the actual addresses of records. Memory management is not automatically represented, but as in [24, 14], allocation and deallocation primitives could easily be added along with automatic checks for memory leaks and dangling references.

Graph Types

Collections of records and pointers can form any number of interesting data structures, which are generally expressed through an invariant on the allowed shapes. We wish to explicitly declare such data structures so that their invariants can be verified by our system. For this purpose, we use *graph types* [28] which is an intuitive notation that makes it feasible to describe complex structures. Invariants of graph type structures can be expressed in monadic second-order logic on finite trees, which allows us to use the MONA tool to verify correctness.

A graph type is a tree-shaped data structure with extra pointers. The underlying tree is called the *backbone*. The constituent records have two kinds of fields: *data fields* which define the backbone, and *pointer fields* which may point anywhere in the backbone. To describe a structural invariant, a pointer field is annotated with a *routing expression* which restricts its destination. In the current work, we have generalized the annotations to be arbitrary formulas that may contain routing expressions as basic predicates. Another difference to [28] is that instead of building types from unions and records, we only use records and nullable pointers. Clearly, the two variations can encode each other; we choose the more primitive version, since it turns out to lead to a more efficient decision procedure. Our syntax and semantics of graph type declarations is described in the next section.

Surprisingly many data structures can be described as graph types. As a simple example, consider the type of binary trees where all nodes contain pointers to the root. In our notation, it looks like:

```
type Tree = {
  data left,right:Tree;
  pointer root:Tree[root<(left+right)*>this &
                    empty(root^Tree.left union
                          root^Tree.right)];
}
```

The syntax for formulas is presented below, but the restriction on the source, `this`, and destination, `root`, of the pointer is read as follows: `this` must be reachable from `root` by following a sequence of `left` or `right` pointers, and the set of `Tree` records having `left` or `right` pointers to the `root` must be empty. Another example is doubly-linked lists with boolean values:

```
type Node = {
  bool value;
  data next:Node;
  pointer prev:Node[this^Node.next={prev}];
}
```

Here, the set of of nodes that can reach the `this` node through a `next` pointer must only contain the `prev` node. The convention that {`null`} is interpreted as the empty set handles the first node in the list.

Our benchmark programs cover a variety of data structures expressed as graph types, including singly-linked lists, doubly-linked lists with tail pointers, red-black search trees, and post-order threaded trees with parent pointers. Additional examples are presented in [28].

The Programming Language

A *program* consists of a set of declarations of types, variables, and procedures, specified by the following grammar:

```
typedecl   →  type T = { ( field ; )* }
field      →  data p⊕ : T
           |  pointer p⊕ : T [ form ]
           |  bool b⊕
progvar    →  data p⊕ : T
           |  pointer p⊕ : T
           |  bool b⊕
procedure  →  proc n ( progvar⊛ ) : ( T | void )
              ( logicvar ; )*
              property
              ( { ( progvar ; )* stm } )?
              property
```

We use the notation $\oplus$ and $\circledast$ for comma-separated lists with one-or-more elements and zero-or-more elements, respectively. T, p, b, and n range over names of types, pointer variables or fields, boolean variables or fields, and procedures, respectively. Ignore for now all occurrences of *logicvar* and *property*; they are introduced later. A `type` consists of a number of fields of kind `data`, `pointer`, or `bool`. The `data` fields span the tree value and the `pointer` fields define extra pointers whose destinations are constrained by a formula. The `bool` fields are used to model finite values. A procedure has a name, formal parameters, a return type, and a body consisting of local variable declarations and statements. If the body is omitted, the declaration is considered a *prototype*.

A statement is one of the following constructs; the `assert` and `split` statements are described later:

```
stm  →  stm stm
     |  asn⊕ ;
     |  proccall ;
     |  if ( condexp ) { stm } ( else { stm } )?
     |  while property ( condexp ) { stm }
     |  return progexp ;
     |  assert property ;
     |  split property property ;
asn  →  lbexp = ( condexp | proccall )
     |  lptrexp = ( ptrexp | proccall )
```

The language permits multiple-assignment statements where all right-hand sides are evaluated before assigning—these are useful for certain program transformations. Expressions have the following form:

```
condexp   →  bexp   |   ?   |   [ form ]
bexp      →  ( bexp )           |  ! bexp
          |  bexp & bexp        |  bexp | bexp
          |  bexp => bexp       |  bexp <=> bexp
          |  bexp = bexp        |  ptrexp = ptrexp
          |  bexp != bexp       |  ptrexp != ptrexp
          |  true  |  false     |  lbexp
lbexp     →  b     |  ptrexp . b
ptrexp    →  null  |  lptrexp
lptrexp   →  p     |  ptrexp . p
proccall  →  n ( ( condexp | ptrexp )⊛ ) [ formula ]
```

The "?" operator stands for nondeterministic boolean choice, which is used to model arithmetic conditions that we cannot capture precisely. The operator "." dereferences a pointer, and the other constructs have the expected meanings.

The language does not contain arithmetic, since our approach focuses on the structural aspects of data types. However, as described in a later section, the technique does permit abstractions of arithmetic properties, for instance for specifying certain ordered data structures.

Program Annotations

Pointer Assertion Logic is a *monadic second-order logic on graph types.* It allows quantification over heap records, both of individual elements and of sets of elements, and uses generalized routing expressions [28] for convenient navigation in the heap. Formulas are used in pointer fields to constrain their destinations, in `while` loops and procedure calls as invariants, in procedure declarations as pre- and post-conditions, and in `assert` and `split` statements. The syntax of formulas is as follows:

```
form       →  ( existpos | allpos ) p⊕ of T : form
           |  ( existset | allset ) s⊕ of T : form
           |  ( existptr | allptr ) p⊕ of T : form
           |  ( existbool | allbool ) s⊕ : form
           |  ( form )               |  ! form
           |  form & form            |  form | form
           |  form => form           |  form <=> form
           |  ptrexp in setexp       |  setexp sub setexp
           |  setexp = setexp        |  setexp != setexp
           |  empty ( setexp )       |  bexp
           |  return                 |  n . b
           |  m ( ( form | ptrexp | setexp )⊛ )
           |  ptrexp < routingexp > ptrexp
predicate  →  pred m ( logicvar⊛ ) = form
```

The identifiers m and s denote predicates and set variables, respectively. The `pos` and `ptr` quantifiers differ in that the former range over heap records while the latter also includes the `null` value. A routing expression formula p_1<r>p_2 is satisfied by a given model if there is a path from p_1 to p_2 satisfying r, as defined below. For reuse of formulas, predicates can be defined as top-level declarations.

Logical variables can be associated to procedures to allow the pre- and post-conditions to be related, as commonly seen in the literature [19, 11]. A logical variable is a universally quantified variable that may occur in the pre- and post-conditions of a procedure but not in the procedure body:

```
logicvar  →  pointer p⊕ : T
          |  bool b⊕
          |  set s⊕ : T
```

In formulas, *ptrexp* has two additional forms allowing access in procedure post-condition to the returned value and in procedure call formulas to the logical variables of the called procedure:

```
ptrexp  →  ...  |  return  |  n . p
```

Set expressions can contain the usual set operators, along with the *up* operation x^T.p which denotes the set of records of type T having a p successor to x:

```
setexp  →  s
        |  ptrexp ^ T . p
        |  { ptrexp⊕ }
        |  setexp union setexp
        |  setexp inter setexp
        |  setexp minus setexp
```

The syntax of routing expressions is a slightly generalized version of that in [28]. A routing expression is a regular expression over routing directives, each being a step *down* or *up* a pointer or data field, or a formula with the extra free variable `pos` filtering away those records that cause the formula to evaluate to false when `pos` denotes one of them:

$$
\begin{array}{lcl}
\textit{routingexp} & \rightarrow & p \quad | \quad \hat{}\ T\ .\ p \quad | \quad [\ \textit{form}\ ] \\
 & | & \textit{routingexp}\ .\ \textit{routingexp} \\
 & | & \textit{routingexp}\ +\ \textit{routingexp} \\
 & | & (\ \textit{routingexp}\ ) \quad | \quad \textit{routingexp}\ *
\end{array}
$$

By default, a pointer field must satisfy the formula given in its type declaration. This can be overridden with *pointer directives* of the form:

$$\textit{ptrdirs} \quad \rightarrow \quad \{\ (\ T\ .\ p\ [\ \textit{form}\ ]\ )^{\circledast}\ \}$$

They allows pointer fields to be constrained differently at different program points. This is important because temporary but intentional invalidation of data structure invariants often occurs in imperative programs, as noted for instance in [21]. Pointer directives, both default and overriding, are required to be *well-formed*. This means that in any store and for any record, the directives associated to the pointer fields must denote *exactly one* record. Fortunately, as proved in [28] this is decidable.

A pair consisting of a formula and a set of pointer directives:

$$\textit{property} \quad \rightarrow \quad [\ \textit{form ptrdirs}\ ]$$

is called a *property* and denotes the set of stores where

- the formula *form* is satisfied;
- the `data` variables denote disjoint acyclic backbones spanning the heap; and
- each `pointer` field satisfies its pointer directive (which is either the default from the type declaration or the overriding from the *ptrdirs*).

Properties occur as procedure *pre-* and *post-conditions*, as `while` *loop invariants*, as `split` *assertions* and *assumptions* (`split` contains two properties), and as `assert` *assertions*.

Semantics of Annotations

The program annotations are invariants of the program that must be interpreted as follows:

- The pre-condition of a procedure may be assumed to hold when evaluating the procedure body;
- the post-condition must hold upon termination of the procedure body;
- every `while` loop invariant must hold upon entry and after each iteration, and may be assumed to hold when the loop terminates;
- assertions specified with `assert` must hold at those program points;
- for `split` statements, the assertion properties must hold, and the assumption properties may be assumed to hold (the reason for introducing these statements is explained in Section 4); and
- at every procedure call, the invariant conjoined with the pre-condition of the called procedure must hold for some valuation of its logical variables, and the invariant conjoined with the post-condition may be assumed upon return, also for some valuation of the logical variables.

In later sections, we show that the requirements imposed by the annotations can be verified automatically, provided that valid and sufficiently detailed invariants are given.

3 Example: Threaded Trees

Before describing our decision procedure, we show a larger example of using PAL. A *threaded tree* is a binary tree in which all nodes contain a pointer to its cyclic successor in a post-order traversal. As a further complication, we equip all nodes with a parent pointer as well. This corresponds to the following graph type:

```
type Node = {
  data left,right:Node;
  pointer post:Node[POST(this,post)];
  pointer parent:Node[PARENT(this,parent)];
}
```

where `POST` and `PARENT` are predicates that spell out these relationships. For example, `PARENT(a,b)` abbreviates the formula:

```
a^Node.left union a^Node.right={b}
```

The `POST` predicate is more involved and makes use of auxiliary predicates `LEAF`, `ROOT`, and `LESSEQ`.

We consider a procedure `fix(x)` that assigns the correct value to `x.post` assuming that this field initially contains the value `null` and that `x` is non-`null`. This is a non-trivial operation that looks like:

```
proc fix(pointer x: Node): void
{
  if (x.left=null & x.right=null) {
    if (x.parent=null) { x.post = x; }
    else {
      if (x.parent.right=null | x.parent.right=x) {
        x.post = x.parent;
      }
      else {
        x.post = findsmallest(x.parent.right);
      }
    }
  }
  else { x.post = findsmallest(x); }
}
```

where the auxiliary procedure `findsmallest` is:

```
proc findsmallest(pointer t: Node): Node
  pointer T: Node;
{
  while (t.left!=null | t.right!=null) {
    if (t.left!=null) { t = t.left; }
    else { t = t.right; }
  }
  return t;
}
```

The question is: Does this code verify? Does the resulting tree always satisfy the data structure invariant? Can type or memory errors ever occur? PALE can provide the answers with some help from us. First, since the argument to `fix` is not a proper threaded tree, we must state a suitable precondition as the property:

```
[x!=null {Node.post[ALMOSTPOST(this,post,x)]}]
```

Here we require that the argument is not `null` and that the data structure invariant can be temporarily violated. The `ALMOSTPOST` predicate is:

```
(this!=x => POST(this,post)) & (this=x => post=null)
```

which simply states the exception that we allow. Second, the `while` loop in `findsmallest` needs an invariant, which is the property:

```
[INV {Node.post[ALMOSTPOST(this,post,x)]}]
```

where the pointer directive states that the threaded tree is still messed up, and the proper invariant `INV` equals:

```
T<(left+right)*>t &
allpos c of Node: LESSEQ(c,t,T) => t<(left+right)*>c
```

which states that `t` is a descendant of `T` and all its post-order successors are further descendants. See [35] for the full code with all post-conditions. In total, six annotations are required. In less that 4 seconds PALE verifies that the code contains no errors.

4 Hoare Logic Revisited

Given an annotated program, we wish to decide whether the program is correct with respect to the annotations. The first step in our decision procedure is to split the given program into Hoare triples [22, 1, 11]. The idea of modeling transformations of the heap with Hoare logic has been studied before [37, 17]. The main novelty of our approach is the choice of PAL as assertion language. Our Hoare "triples" have a nonstandard form:

triple $\rightarrow$ *property stm*

The statement *stm* is not allowed to contain `while` loops, `split` statements, or procedure calls. A triple is *valid* if

- executing *stm* in a store where *property* is satisfied cannot violate any assertions specified by `assert` statements occurring in *stm*; and
- the execution always terminates in a store consisting of disjoint, acyclic backbones spanning the heap in which all pointer directives hold.

As opposed to normal Hoare triples, these have no explicit post-condition, but the *stm* part may contain `assert` substatements. This simple generalization allows many assertions to be made without always breaking triples into smaller parts, as was often the case in [24] and [14]. For instance, an `if` statement where both branches end in `assert` statements does not necessarily need to be broken into two parts. Also, using this form of Hoare triples simplifies the encoding in monadic second-order logic described in Section 5.

We define the *cut-points* of a program (according to [16]) as the following set of program points: the beginning and end of procedure bodies and `while` bodies, the `split` statements (these do not affect the computation and are considered single program points), and before each procedure call.

For each cut-point in the given program, we generate a Hoare triple from the property associated with that point and the code that follows until reaching other cut-points. Extra `assert` statements are automatically inserted for these other cut-points, reflecting the assertions they define. In case of `split` statements, we here use the assertion property. For procedure calls, we use the pre-condition property of the called procedure conjoined with the call invariant formula. Recall that we do allow `if` statements in the Hoare triples. However, if one branch contains a cut-point, we require syntactically that the other branch also contains a cut-point or that the `if` statement is immediately followed by one. Typically, `split` statements are used to fulfill this requirement. As a result, the statement part of a Hoare triple in general has a tree shape with one cut-point in the root and one in each leaf. See [35] for more details.

We claim without proof that this reduction is semantically sound, with two exceptions:

- For `split` statements, the assertion property may not be implied by the assumption property, thereby causing a "gap" between the Hoare triples. This is intentional, because it allows to recover from situations where the required properties are beyond what is expressible in Pointer Assertion Logic, such as arithmetical properties. Using `split` statements at a few selected places, one can then still verify properties of the remaining parts of the code. However, none of the examples shown in Section 7 require this feature.
- Procedure calls are known to cause complications for Hoare logic [11]. In our case, there is in general no guarantee that the call invariant is actually a valid invariant. However, in most situations, simple syntactic requirements suffice, since recursive calls in data type operations typically follow the recursive structure of the graph type backbones. A sufficient condition is that the call invariant only accesses variables and record fields that are not assigned to in the procedure. Such requirements ensure that the invariant and the procedure's pre- and post-conditions express properties of disjoint parts of the store, reminiscent of the "independent conjunctions" in [37]. All the examples shown in Section 7 can be handled by simple rules, which we plan to build into PALE.

In PALE, this phase is implemented as a desugaring process reducing all procedures, `while` loops, `split` statements, and procedure calls to *transduction* declarations having the form "`transduce` *triple*". In the following section we describe how validity of these simpler `transduce` constructs can be decided.

In contrast to techniques based on generating the weakest preconditions for all procedures, each program or procedure is not turned into one single verification condition; instead we use the annotations to split the program into Hoare triples that are verified independently. Also, as opposed to [17], we will not rely on fixpoint iterations. This means that detailed invariants may be required; however, it has the advantage that the technique becomes highly modular and hence scalable.

5 Deciding Hoare Triples in MONA

We need to decide validity of a Hoare triple of the form

property stm

where the statement *stm* is without loops and procedure calls. The question is whether every execution of *stm* starting from a store satisfying *property* is guaranteed to satisfy the assertions given by `assert` statements and to result in stores with disjoint, acyclic backbones spanning the heap in which all relevant pointer directives hold. A result in [29] shows in a very general setting that this is a decidable question. In essence, we encode each Hoare triple in the logic *weak monadic second-order theory of 2 successors*, which is decidable using the MONA tool [26, 25, 34].

Similarly to the previous implementations [24, 14] we use a particular *transduction* technique. This idea allows us to avoid an explicit construction of weakest pre-conditions working backwards through the statement sequence. Instead, we directly simulate (transduce) the statements and mirror their effect by updating a fixed collection of *store predicates* which abstractly describes a set of stores. It is shown in [29] that any question about the resulting set of stores can be answered by phrasing it in terms of the transduced store predicates and checking for validity of the resulting formula.

The store predicates describe a set of stores in MONA logic. They can be thought of as an interface for asking questions about a store. There are 11 kinds of predicates:

- `bool_`T`_`b`(v)` gives the value of the `bool` field b in a record `v` of type T;
- `succ_`T`_`d`(v,w)` holds if the record `w` is reachable from the record `v` of type T along a data field named d;
- `null_`T`_`d`(v)` holds if the data field d in the record `v` of type T is `null`;
- `succ_`T`_`p`(v,w)` holds if the record `w` is reachable from the record `v` of type T along a pointer field named p;
- `null_`T`_`p`(v)` holds if the pointer field p in the record `v` of type T is `null`;
- `ptr_`d`(v)` holds if the record `v` is the value of the data variable d;
- `null_`d`()` holds if the data variable d is `null`;
- `ptr_`p`(v)` holds if the record `v` is the destination of the pointer variable p;
- `null_`p`()` holds if the pointer variable p is `null`;
- `bool_`b`()` gives the value of the boolean variable `b`;
- `memfailed()` holds if a null-pointer dereference has occured.

All properties of a store can be expressed using these predicates in MONA logic. The transduction process generates a collection of such store predicates for each program point. For convenience, we describe this by indexing the predicates with program points; for example, for each program point i there is a version of the `bool_`T`_`b(v) predicate called `bool_`T`_`b`_`i`(v)`.

An initial collection of store predicates is defined to reflect the formula and pointer directives that constitute the pre-condition of the Hoare triple. In the encoding into MONA code, the program variables are modeled as free variables, which are universally quantified in the final validity formula that is given to MONA. For example, a `bool` variable is modeled as a boolean variable `_bool_`b in MONA and the corresponding initial store predicate is:

```
bool_b_0() = _bool_b
```

Similarly, a pointer variable p is modeled as a first-order MONA variable `_ptr_`p and the corresponding initial store predicate is:

```
ptr_p_0(v) = v = _ptr_p
```

A `bool` field b in a record of type T is modeled as a second-order variable `_bool_`T`_`b containing the set of records in which b is true. Consequently, the corresponding initial store predicate is:

```
bool_T_b_0(v) = v in _bool_T_b
```

As a final example, we consider pointer fields whose initial store predicate is:

```
succ_T_p_0(this,p) = f
```

where f is the encoding of the formula associated with the p field of T. If the pre-condition of the Hoare triple contains the pointer directive T`.`p`[`*form*`]`, then that formula is *form*, otherwise the default formula from the type definition is used.

Across a simple statement, two collections of store predicates are related in a manner that reflects the semantics of that statement. Consider for example a type of linked lists:

```
type Node = { data next: Node; }
```

and a simple statement involving two pointer variables of type `Node`:

```
p = q.next;
```

If this statement is enclosed by program points i and j, then the store predicates are updated as follows in MONA code:

```
memfailed_j() = memfailed_i() | null_q_i()
ptr_p_j(v) = ex2 ptr_q_i(w) & w: succ_Node_next_i(w,v)
null_p_j() = ex2 w: ptr_q_i(w) & null_Node_next(w)
```

while the other store predicates remain unchanged. The PALE tool generates such store predicate updates for all Hoare triples and subsequently generates formulas to check the required properties. Between conditionals, routing expressions, and various primitive statements this is a complex translation reminiscent of generating machine code in a compiler. The details can be studied in [35]. The way assignments are handled without losing aliasing information, as in the example above, is essentially the same as in [36].

Checking that an assertion property at a given program point cannot be violated can be expressed by encoding the property using the store predicates associated with the program point together with the pre-condition property encoded with the initial store predicates. There is a strong connection between this transduction technique and the more traditional weakest-precondition technique: if the predicate invocations in the MONA formulas are "unfolded", one essentially gets the weakest pre-condition. The main advantage of using the "forward" transduction technique instead of a "backward" weakest-precondition technique is an implicit reuse of intermediate results.

Checking that the resulting backbones are disjoint, acyclic, and span the heap is based on formulas for expressing transitive closure. Checking that a pointer directive holds is in [28] shown to be decidable in monadic second-order logic. This result generalizes easily to our extension of graph types, where arbitrary formulas rather than only routing expressions can be used as pointer directives.

The MONA tool transforms the resulting formulas, which can be quite large, into equivalent minimal Guided Tree Automata [4] represented as BDD structures [6], and from that either deduces validity or generates counterexample models. In the latter case, the PALE tool decompiles that model into a program store which causes the program to fail. The use of Guided Tree Automata rather than ordinary tree automata yields an exponential saving by factorizing the state space according to the recursive structure of the graph type backbones. Compared to the WSRT technique used in [14], our choice of describing the backbones as records with pointers rather than as recursive types allow a simpler and more efficient automaton guide to be constructed. Also for efficiency reasons, we compile directly into MONA logic rather than use a more high-level logic, such as FIDO [30].

Note that a collection of store predicates is vaguely similar to the abstract store descriptions employed by TVLA. Consequently, it might seem that we could follow their approach and use a fixpoint process to transduce a `while` loop. However, this is in general not possible, since such fixpoints may require transfinite induction. Hence, we resort to using invariants to break up loops.

This transduction approach introduces no imprecision; it is both sound and complete for individual Hoare triples.

6 Data Abstractions

In [38, 31], abstractions of the data contained in the heap records can be tracked by specifying suitable *instrumentation predicates*. As an example, a predicate $dle(x, y)$ is used to represent "the data in x is less than or equal to the data in y". To illustrate the power of PAL, we show that a similar approach works for our technique.

As an example, we instrument the ubiquitous linked-list `reverse` example to verify that reversal of a list ordered in increasing order results in a list ordered in decreasing order:

- We associate two boolean fields, `next_dle` and `next_dge`, to the `next` field in the linked-list type, with the intended meaning: `next_dle` is true in a given record if the data in the record denoted by the `next` pointer is certain to be *less than or equal* to the data in the given record – and likewise for `next_dge` with *greater than or equal.*

- Similarly, for each pair of program pointer variables, two boolean variables are added to keep track of the relative order of the records being pointed to. With a subsequent dead-code elimination, a total of three boolean variables suffice.

- For each pointer assignment, the new boolean fields and variables are updated accordingly. For instance,

  ```
  list.next = res;
  ```

  is replaced by the multiple-assignment statement:

  ```
  list.next = res, list.next_dle = res_dle_list;
  ```

  reflecting the change of the `next` field.

If arbitrary PAL formulas are allowed as right-hand sides of the new assignments, even complex reachability properties can be captured. For this example, simple assignments suffice, though. As in [31], this is also sufficient to verify for instance that `bubblesort` actually sorts the elements.

The intellectual effort needed to update the data abstraction bits seems to be the same as to define the required operational semantics in TVLA. As hinted in the example, some degree of automation is possible for our technique; however, we leave that for future work.

Note that many data structures, in particular variations of search trees, can be abstractly described by associating to every node a few of bits of information summarizing properties of the tree. Those data structures can also be verified using techniques like these.

7 Implementation and Evaluation

Our verification technique is implemented in a tool called PALE, the Pointer Assertion Logic Engine. Given an annotated program, PALE checks that:

- the pointer directives are well-formed;
- `null` pointer dereferences cannot occur;
- at each cut-point that the `data` variables contain disjoint, acyclic backbones spanning the heap and that the assertions and pointer directives are satisfied;
- all `assert` assertions are valid; and
- all cut-point properties are satisfiable.

There is not necessarily an error in the program if a cut-point property is unsatisfiable, but it usually indicates an error in the specification. As previously mentioned, memory allocation can easily be expressed such that the tool would also check for memory leaks and dangling references.

Using PALE, we have evaluated the technique on a number of examples dealing with a variety of data structures. In all cases, we check for memory errors and possible violations of the data structure invariants:

- *Singly-linked lists* with the operations `reverse`, `search`, `zip`, `delete`, `insert`, and `rotate`. These examples have been scrutinized before [8, 24, 32]. We also include the `concat` operation on lists with tail pointers from Section 1. We have tried `bubblesort` as in [31] but with various degrees of abstraction of the data: In `bubblesort_simple`, the record values are abstracted away so only `null` pointer dereferences are checked for; in `bubblesort_boolean`, the values are abstracted to booleans which in the post-condition are checked to be properly sorted; and in `bubblesort_full`, the data abstraction technique from Section 6 is used as in [31] to conclude that the resulting lists are sorted. We also use data abstractions in `orderedreverse` to show that `reverse` switches the order of a sorted list. Finally, we try `recreverse`, which is a recursive version of `reverse`.

Example name	Lines of code	Invariants (formulas)	GTA operations	Largest GTA States	Largest GTA BDD nodes	Time (seconds)	Memory (MB)
`reverse`	16	1	1,109	35	142	0.52	2
`search`	12	1	853	27	85	0.25	2
`zip`	33	1	1,753	174	730	4.58	11
`delete`	22	0	973	73	349	1.36	5
`insert`	33	0	1,005	103	443	2.66	7
`rotate`	11	0	590	44	213	0.22	1
`concat`	24	0	1,056	48	177	0.47	3
`bubblesort_simple`	43	1	1,477	373	3,289	2.86	18
`bubblesort_boolean`	43	2	1,737	357	3,922	3.37	12
`bubblesort_full`	43	2	2,069	373	3,291	4.13	19
`orderedreverse`	24	1	1,091	29	100	0.46	3
`recreverse`	15	2	1,019	42	176	0.34	2
`doublylinked`	72	1	4,163	230	796	9.43	13
`leftrotate`	30	0	1,489	165	1,550	4.62	7
`rightrotate`	30	0	1,489	165	1,550	4.68	7
`treeinsert`	36	1	1,989	137	844	8.27	31
`redblackinsert`	57	7	4,279	297	2,419	35.04	44
`threaded`	54	4	3,505	50	248	3.38	7

Figure 1: Statistics from PALE experiments.

- *Doubly-linked lists with tail pointers* [28] with the operations `delete`, `search`, `insert`, and `concat`.
- *Red-black search trees* [10] with the standard operations `leftrotate`, `rightrotate`, `treeinsert`, and `redblackinsert`. We include the non-arithmetic part of the red-black search tree invariant, that is, that the root is black and red nodes have black children:

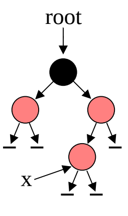
```
BLACK(root) &
allpos q of Node: ROOT<(left+right)*>q  =>
 (RED(q) => BLACK(q.left) & BLACK(q.right));
```

- *Threaded trees* [28], as shown in Section 3, where every node has a pointer to its post-order cyclic successor and a pointer to its parent, with a `fix` operation for reestablishing the correct `post` pointer for a given node.

The resources for translation into MONA code and for the automaton analysis are negligible. Figure 1 shows the time and space consumptions of the MONA automaton operations (on a 466MHz Celeron PC) for the examples, along with the number of GTA operations (here we count only the essential operations: minimization, projection, and product), the size of the largest intermediate minimized automaton (in number of states and in number of BDD nodes). Note that some examples implement individual operations while others implement whole data types. The lines of code measure the underlying program only, thus disregarding the PAL annotations. "Invariants" is the total number of `split` statements, `while` statements, and procedure calls that require explicitly stated invariants. This number is an indication of the effort required by the programmer to make PALE work, in addition to writing the program and its specification. The invariants for `redblackinsert` were admittedly hard to get right. However, the programs that require the most complicated invariants are also those that have the most complicated pointer operations and hence are the ones in most need of verification. The table shows that the examples typically run in seconds despite requiring a quite large number of automaton operations. Since the complexity is non-elementary in the size of the program, intractable examples do exist but they do not seem to occur often in practice. The verification time seems insignificant compared to the time required to design a given data type and specify the invariants, however, it is useful in the design cycle that verification is efficient.

The code for the `bubblesort` examples (excluding annotations) is taken from [31]. Interestingly, PALE discovered a minor bug (a null-pointer dereference) even though the code had allegedly been verified by TVLA, which spent 245 seconds compared to 4 seconds for PALE. This huge speedup shows an instance where using invariants is much faster than performing a fixpoint iteration. This suggests that PALE may be quite scalable. Another noteworthy point discovered by PALE is that in [10], the authors forget to require the root to be initially black in `redblackinsert`. (More precisely, they mention the requirement in the proof of correctness, but not in the specification.)

Versions with plausible bugs planted typically take roughly the same time to process as the correct programs. For such buggy versions, counterexamples are generated, which is crucial for determining whether the error is in the program, the assumptions, or the assertions. As an example, if a conditional in `redblackinsert` erroneously tests for a specific node to be black rather than red, PALE produces the following counterexample store for the Hoare triple containing the conditional:



Here, the root node is black and the others are red, and we omit field names and all pointer fields. Such a counterexample is clearly useful for locating the bug. Notice that for this bug, the approach in [23] would not find the bug for heap bounds of less than four records.

The experiments show that our approach does work in practice for non-trivial data structures, and with time and space requirements which are as good as or better than those for the previous more specialized versions [24, 14] and related approaches with similar goals [31, 23, 13, 17].

8 Conclusion

It is well known that developing formal program specifications is expensive, but for some safety critical applications a guarantee of partial correctness of data type implementations can be worth the effort. A tool such as PALE can be used to verify specifications expressible in Pointer Assertion Logic, and also to guide the programmer by the generation of counterexamples. With verification techniques based on undecidable logics, either the programmer may have to guide a theorem prover to the proofs, not even being certain that they exist, or accept that the reply may be "don't-know". With less expressive techniques, important aspects of the data types may not be expressible and hence not verifiable. In contrast to traditional program analyses, our technique is highly modular: each statement in the given program is analyzed only once. To verify complex properties, the technique often requires detailed invariants to be provided. However, since we primarily aim for data-type implementations, we believe that this annotation overhead is reasonable compared to the effort of creating the program. In conclusion, Pointer Assertion Logic may provide a fruitful compromise between expressibility and usability.

Although facing a non-elementary theoretical complexity, the examples we provide show that logic and automaton based program verification is feasible. Furthermore, we believe that the efficiency of the implementation can be improved by at least an order of magnitude by tuning the MONA tool using heuristics as proposed in [27]. As also suggested in [23, 20] we may benefit from an initial simplification phase that performs program slicing or partial evaluation of the source programs.

Future work will also examine the possibility of incorporating simple arithmetic into the language. The MONA tool can also be used as an efficient decision procedure for Presburger arithmetic [39, 26], which is sufficient for many properties. In [21], abstract data structure descriptions are used to improve program analyses in optimizing compilers. Pointer aliasing, for instance, can be expressed in PAL, so the detailed knowledge of the heap structure provided by PALE might also be useful for optimization. Another idea is to build a translator from C, C++, or Java to PALE to make the tool more practically useful. Finally, it might be interesting to integrate the "independent conjunctions" from [37] into PAL to support local reasoning and make the tool easier to use.

The full source code for the PALE tool, the examples, and a detailed description of the desugaring and code generation to MONA are available from the PALE site at `http://www.brics.dk/PALE/`.

Acknowledgments

This work was done while the first author was visiting UC Berkeley. Thanks to Alex Aiken, Jeff Foster, and Zhendong Zu for valuable discussions and comments.

References

[1] Krzysztof R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Transactions on Programming Languages*, 3(4):431–483, 1981.

[2] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the SPIN Software Model Checking Workshop*, volume 1885 of *LNCS*, 2000.

[3] Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for describing linked data structures. In *European Symposium On Programming, ESOP'99*, volume 1576 of *LNCS*, 1999.

[4] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, WIA'96*, volume 1260 of *LNCS*, 1997.

[5] Paul E. Black and Phillip J. Windley. Inference rules for programming languages with side effects in expressions. In *International Conference on Theorem Proving in Higher Order Logics*, 1996.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

[7] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7), 2000.

[8] Stephen A. Cook and Derek C. Oppen. An assertion language for data structures. In *Principles of Programming Languages, POPL'75*, pages 160–166, 1975.

[9] C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proceedings of the SPIN Software Model Checking Workshop*, volume 1885 of *LNCS*, 2000.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[11] Patrick Cousot. Formal models and semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 841–993. MIT Press/Elsevier, 1990.

[12] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December, 1998.

[13] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Checking cleanness in linked lists. In *Seventh International Static Analysis Symposium, SAS'00*, 2000.

[14] Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach. Compile-time debugging of C programs working on trees. In *European Symposium on Programming Languages and Systems, ESOP'00*, volume 1782 of *LNCS*, 2000.

[15] David Evans. Static detection of dynamic memory errors. In *Programming Language Design and Implementation, PLDI'96*, 1996.

[16] Robert W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science, American Math. Soc.*, 1967.

[17] Pascal Fradet, Ronan Gaugne, and Daniel Le Métayer. Static detection of pointer errors: An axiomatisation and a checking algorithm. In *European Symposium on Programming, ESOP'96*, volume 1058 of *LNCS*, 1996.

[18] Pascal Fradet and Daniel Le Métayer. Shape types. In *Principles of Programming Languages, POPL'97*. ACM, 1997.

[19] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[20] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[21] Laurie Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Program Language Design and Implementation, PLDI'92*, 1992.

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

[23] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis, ISSTA'00*. ACM, 2000.

[24] Jacob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Programming Language Design and Implementation, PLDI'97*, 1997.

[25] Nils Klarlund. Mona & Fido: The logic-automaton connection in practice. In *Computer Science Logic, CSL'97*, volume 1414 of *LNCS*, 1998.

[26] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[27] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. In *Fifth International Conference on Implementation and Application of Automata, CIAA'00*, LNCS, 2000.

[28] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Principles of Programming Languages, POPL'93*. ACM, 1993.

[29] Nils Klarlund and Michael I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In *Trees in Algebra and Programming, CAAP'94*, volume 787 of *LNCS*, 1994.

[30] Nils Klarlund and Michael I. Schwartzbach. A domain-specific language for regular sets of strings and trees. *IEEE Transactions On Software Engineering*, 25(3):378–386, 1999.

[31] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: a case study. In *International Symposium on Software Testing and Analysis, ISSTA'00*. ACM, 2000.

[32] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *Seventh International Static Analysis Symposium, SAS'00*, 2000.

[33] Albert R. Meyer. Weak monadic second-order theory of successor is not elementary recursive. In R. Parikh, editor, *Logic Colloquium, (Proc. Symposium on Logic, Boston, 1972)*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154, 1975.

[34] Anders Møller. MONA project home page. `www.brics.dk/mona`.

[35] Anders Møller. PALE project home page. `www.brics.dk/PALE`.

[36] Joseph M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, 1982.

[37] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, 2000.

[38] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3 valued logic. In *Principles of Programming Languages, POPL'99*, 1999.

[39] Thomas R. Shiple, James H. Kukula, and Rajeev K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *Computer Aided Verification, CAV'98*, volume 1427 of *LNCS*, 1998.