Automatic Verification of Pointer Programs using Monadic Second-Order Logic*

Jakob L. Jensen, Michael E. Jørgensen, Michael I. Schwartzbach
BRICS, University of Aarhus
{u820406,medgar,mis}@brics.dk

Nils Klarlund
AT&T Research
klarlund@research.att.com

Abstract

We present a technique for automatic verification of pointer programs based on a decision procedure for the monadic second-order logic on finite strings.

We are concerned with a `while`*-fragment of Pascal, which includes recursively-defined pointer structures but excludes pointer arithmetic.*

We define a logic of stores with interesting basic predicates such as pointer equality, tests for `nil` *pointers, and garbage cells, as well as reachability along pointers.*

We present a complete decision procedure for Hoare triples based on this logic over loop-free code. Combined with explicit loop invariants, the decision procedure allows us to answer surprisingly detailed questions about small but non-trivial programs. If a program fails to satisfy a certain property, then we can automatically supply an initial store that provides a counterexample.

Our technique has been fully and efficiently implemented for linear linked lists, and extends in principle to tree structures. The resulting system can be used to verify extensive properties of smaller pointer programs and could be particularly useful in a teaching environment.

*`detex paper.tex | wc | cut -d' ' -f2 = 4821`

1 Introduction

Background

Programming with pointers is difficult and risky. This has motivated a huge body of work concerned with analyzing pointer programs and verifying their properties.

Traditional pointer analyses accept preexisting programs and provide approximate and conservative answers to a fixed collection of questions concerning pointer aliases, `nil` dereferences, dangling references, and unclaimed memory.

We present an approach based on a first-order *store logic* in which such questions can be stated as simple formulas. But the store logic allows pointer analysis to be taken to a higher symbolic level, where more general properties expressed by assertions can be verified by a complete decision procedure. In this way, our verifier works as an oracular, symbolic debugger for pointer code.

Our approach is made practical through a strong connection to finite state regularity.

Contributions

In this paper we present the following results.

- A first-order logic of memory cells and their contents. The logic specifies *regular* (finite-state) languages of stores, and we show that it subsumes the scope of traditional pointer analyses.
- A complete decision procedure for Hoare triples using this logic over loop-free code without arithmetic. For full programs our technique is of course approximative, but we can in this manner clearly characterize its power.
- An automatic technique for generating concrete counterexamples whenever a program fails to verify. These can be used to interactively explain programming errors.
- A full implementation for a `while`-fragment of Pascal that considers linear linked lists only. However, theoretical results guarantee that our approach generalizes to tree structures.

- A collection of small but non-trivial programs for which we verify interesting properties.
- A sketch of a method for developing pointer programs, where debugging is replaced with attempts to verify strategically placed formulas.

A more abstract contribution is to identify and exploit an important niche of finite state regularity in programming language semantics.

Related Work

Our work does not follow the established tradition of conventional heap-based pointer analysis [7, 8, 15, 4, 5, 14, 13, 17, 2] which develops specialized algorithms for answering specific questions about preexisting programs without annotations. We are more general in providing a full, decidable logic in which one may phrase a broad range of questions, and in providing concrete counterexamples whenever a question is answered in the negative. Also, the use of Hoare triples allows a modular analysis of programs. However, we are less general in requiring programs to be explicitly annotated with formulas and invariants; also, the present implementation handles only list structures.

Most similar in spirit is the ECS system [3], which also uses a restricted specification logic, requires explicit annotation with formulas and invariants, and generates counterexamples. A principal difference is in the questions that can be phrased. We are concerned with non-arithmetic properties of pointers in the heap, whereas the ESC logic includes array subscript errors and deadlocks in concurrent programs. The two approaches are incomparable in their ambitions. For loop-free code we provide a complete, model-theoretic BDD-based decision procedure, whereas ESC relies on an incomplete theorem prover.

The system LCLint [6] uses simple annotations and a fast, incomplete decision procedure to detect certain dynamic memory errors in C programs. In comparison, our technique is more detailed but restricted to a simpler store model.

Our previous work on decidable graph transductions [11] describes the theoretical foundations for our current approach and for several generalizations.

2 The Pascal Subset

We consider a subset of the Pascal language, which has been restricted for reasons of both presentation and necessity. First, we have chosen Pascal rather than e.g. C to reflect that we cannot handle pointer arithmetic. For simplicity, we consider only a *while*-fragment; however, recursive procedures are easily accommodated. Furthermore, our present implementation only supports lists rather than trees; however, our theoretical foundations extend cleanly to tree structures. Finally, our verification technique does not consider integer arithmetic, and to make this approximation explicit, our language includes only enumeration types as basic values.

Syntax

We allow exclusively declarations of enumeration types, record types with variants, and types of pointers to records. All declared variables are required to have pointer types. A *pointer variable expression* is defined as:

V	::=	`x`	variable
		V`^.n`	pointer traversal

A *pointer value expression* is defined as:

P	::=	**V**	variable expression
		nil	the *nil* constant

A *boolean expression* is quite restricted since we do not allow arithmetic:

B	::=	$\mathbf{P}_1$`=`$\mathbf{P}_2$	pointer equality
		$\mathbf{P}_1$`<>`$\mathbf{P}_2$	pointer inequality
		V`^.t=v`	variant test
		$\mathbf{B}_1$ `and` $\mathbf{B}_2$	conjunction
		$\mathbf{B}_1$ `or` $\mathbf{B}_2$	disjunction
		`not` **B**	negation

The collection of *statements* is fairly complete:

S	::=	**V**`:=`**P**	assignment
		begin **S** *end*	block
		$\mathbf{S}_1$ `;` $\mathbf{S}_2$	sequence

if **B** *then* **S**	conditional
if **B** *then* $\mathbf{S}_1$ *else* $\mathbf{S}_2$	conditional
while **B** *do* **S**	loop
new(**V**,v)	allocation
dispose(**V**,v)	deallocation

We do not consider input and output explicitly; rather, we assume that values are communicated through the global variables.

An Example Program

An example of a program in our language is the following, which performs an in-situ reversal of a linked list with colored elements.

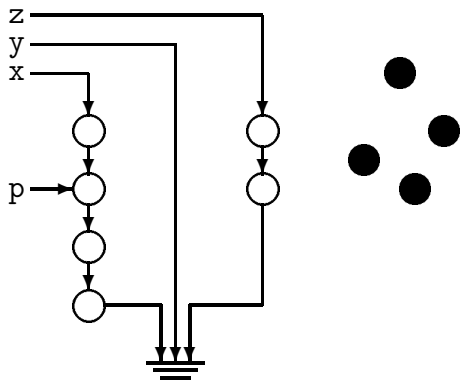
```
program reverse;
type Color = (red,blue);
     List = ^Item;
     Item = record
               case tag: Color of
               red,blue: (next: List)
             end;
var x,y,p: List;
begin
    while x<>nil do
    begin
       p:=x^.next;
       x^.next:=y;
       y:=x;
       x:=p
    end
end.
```

Let us illustrate the scope of our ambitions on this small example. With our system, we can automatically verify that the resulting structure is still a linked list conforming to the type `List`. We can also verify that no pointer errors have occurred, such as dangling references or unclaimed memory cells. However, we cannot verify that the resulting list contains the same colors in reversed order. Still, our partial verification will clearly serve as a finely masked filter for many common programming errors.

3 Stores and Formulas

Stores

We are interested in *stores* consisting of *cells* and *pointers*:



The white circles are *record cells*, which are labeled with record types and variants. The black circles are *garbage cells* corresponding to deallocated records. The ground symbol is a distinguished ***nil*** cell. A record cell may have an outgoing pointer (several, if we consider trees). The named handles on a store are either *data variables* (`x`, `y`, `z`) or *pointer variables* (`p`). It should be clear that the state of a program in our Pascal subset can be modeled as such a store, provided we classify the program variables as either data or pointer variables.

We are particularly interested in *well-formed* stores, which satisfy the following properties:

- the record cells and their pointers form disjoint lists;
- each data variable points either to ***nil*** or to the root of a unique list;
- a pointer variable may point to ***nil*** or to any record cell;
- garbage cells have no incoming pointers; and
- the Pascal type system is respected.

Logic

We now define a logic of stores in which one may state interesting properties; for example, well-formedness is a property that can be expressed as a formula.

The logic is a first-order formalism in which terms denote cells in the store. A *cell term* is of the form:

C ::=	`x`	data variable
	`p`	pointer variable
	C`^.n`	pointer traversal
	`nil`	the ***nil*** cell
	α, β, ...	cell term variables

A *formula* is built from basic predicates and the usual connectives:

Φ ::=	$\mathbf{C}_1$ `=` $\mathbf{C}_2$	cell equality
	$\mathbf{C}_1$ `<R>` $\mathbf{C}_2$	routing relation
	`~`Φ	negation
	Φ_1 `&` Φ_2	conjunction
	`ex` α`:` Φ	quantification over cells

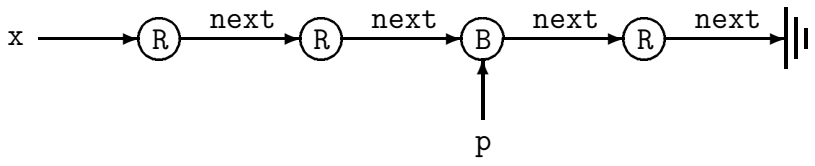
A *routing relation* (introduced in [10]) is a binary relation on cells:

R ::=	`n`	traverse an `n`-pointer
	`(T:v)?`	test for type and variant
	`nil?`	test for the ***nil*** cell
	`garb?`	test for a garbage cell
	$\mathbf{R}_1$ `.` $\mathbf{R}_2$	concatenation
	$\mathbf{R}_1$ `+` $\mathbf{R}_2$	union
	R`*`	Kleene star

The relation c`<`**R**`>`d holds if the regular language denoted by **R** contains a sequence that leads from the cell c to the cell d and in which all pointer traversals are possible and all tests are successful as they are encountered. The individual tests are decided as follows:

- the test `(T:v)?` is true if the cell has record type `T` and variant `v`;
- the test `nil?` is true if the cell is the ***nil*** cell; and
- the test `garb?` is true if the cell is a garbage cell.

Thus, in the following store containing a list with red and blue nodes:



the relation `x<next.next.(List:blue)?>p` is true, whereas the relation `p<next*>x` is false. We allow the usual syntactic sugar, such as `true`, `|`, `=>`, `<>`, and `all`. Furthermore, the unary relation **<R>***c* abbreviates *c*<**R**>*c*. Some examples of general formulas on stores are:

- if `p` is not red, then it can be reached from `x` through a number of `next` pointers: `~<(List:red)?>p => x<next*>p`;
- no garbage cells have incoming `next` pointers: `all c,d: c<next>d => ~<garb?>d`; and
- no non-*nil* cell has two distinct incoming `next` pointers: `all c,p,q: (c<>nil and p<next>c and q<next>c) => (p=q)`.

All of the above formulas are true for the example store. This first-order logic may be extended to a monadic second-order logic which additionally allows *sets* of cells; however, this extension is not needed for the current presentation. Note that neither version of the logic permits us to mention arithmetic properties, such as the lengths of lists.

4 Deciding Hoare Triples

Given a loop-free statement **S** and any two formulas Φ_1 and Φ_2, we can automatically decide *validity* of the Hoare triple: $\{\Phi_1\}$ **S** $\{\Phi_2\}$. We define validity as follows: if we start our computation in a *well-formed* store (with sufficient available memory cells) that satisfies the precondition Φ_1, then the execution of the statement **S** will always result in a *well-formed* store that satisfies the postcondition Φ_2. The inclusion of the well-formedness predicates is an essential technical requirement for our decision procedure. A simple example of a valid triple is:

```
{ x<next*>p & p^.next=nil }
new(q,blue);
q^.next:=nil;
p^.next:=q
{ x<next*>q & q^.next=nil & p<>q }
```

This triple expresses that if `p` points to the last element of the list `x` in a well-formed store, then the three lines of code result in a well-formed store,

where `q` points to the last element in `x` and where `p` is different from `q`. Our logic defines the semantics of the formula `p^.next=nil` such that it holds only if `p^.next` is well-defined and is equal to `nil`.

The key insight behind our decision procedure is to encode a store as a string. Clearly, the effect of a loop-free program is then to transform one string into another. The set of stores that satisfy a given formula Φ in our logic can be shown to always form a *regular* set of strings $\mathcal{L}(\Phi)$. Furthermore, formulas in the store logic can be shown to be closed under the weakest precondition transformations induced by loop-free code.

Thus, our decision procedure applied to the triple $\{\Phi_1\}\ \mathbf{S}\ \{\Phi_2\}$ can be roughly sketched as follows:

- compute the weakest precondition $wp(\mathbf{S}, \Phi_2)$ describing those well-formed stores that under the transformation induced by $\mathbf{S}$ will produce well-formed stores that satisfy Φ_2;
- compute a predicate $alloc(\mathbf{S})$ that describes the number of cells that $\mathbf{S}$ may need to allocate; and
- decide if $\mathcal{L}(\Phi_1) \cap \mathcal{L}(alloc(\mathbf{S})) \subseteq \mathcal{L}(wp(\mathbf{S}, \Phi_2))$.

There are of course subtle details to this approach that are not explained here; however, the formal foundations for the general case of trees are presented in [11].

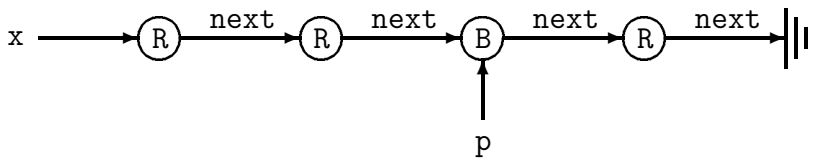
Encoding Stores

To encode a single store as a string, we need a suitable alphabet. A single alphabet symbol will consist of both a *label* and a *bitmap*. The label is either **nil**, **garb**, **lim**, or a pair **(T:v)** where `v` is a variant of the record type `T` in the current program. The bitmap indicates a position for each declared data variable and pointer variable. The encoding of a store is now defined as follows:

- the first position (and no other) is labeled **nil**;
- following the first position is a sequence of encodings of the lists;
- each list is encoded (in the declared order) as a sequence of cells followed by a **lim** symbol, where each cell is followed by its successor in the list;
- each cell is labeled with its type and variant;

- following the encodings of lists are the garbage cells;
- each variable occurs in exactly one bitmap;
- a data variable occurs in the bitmap of the root of its list, or in **nil** if it is empty; and
- a pointer variable occurs in the bitmap of its destination.

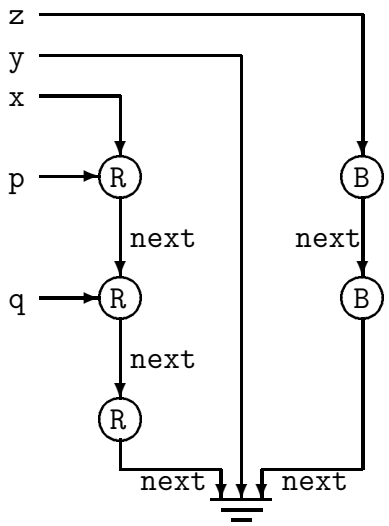
For example, the store:



is encoded as the string of six symbols:

[**nil**,∅] [(`List:red`),{`x`}] [(`List:red`),∅] [(`List:blue`),{`p`}] [(`List:red`),∅] [**lim**,∅]

and the more complicated store:

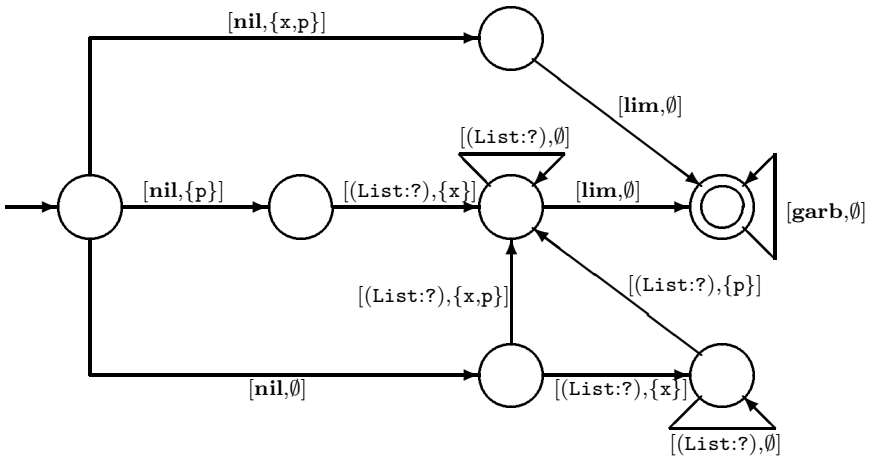


is similarly encoded as the string of nine symbols:

[**nil**,{`y`}] [(`List:red`),{`x`,`p`}] [(`List:red`),{`q`}] [(`List:red`),∅] [**lim**,∅] [**lim**,∅]
[(`List:blue`),{`z`}] [(`List:blue`),∅] [**lim**,∅]

Encoding Formulas

We claim without proof that the set of well-formed stores satisfying a given formula corresponds to a regular language of string encodings. As an example, consider the formula `x<next*>p`. It corresponds to the set of strings accepted by the following deterministic, partial automaton:



We have used the symbol [(`List:?`),...] to indicate that a transition is possible for both colors. As indicated, even fairly simple formulas may yield very complicated automata, since they make explicit all the special cases; for example, the string:

[**nil**,{`x`,`p`}] [**lim**,∅]

corresponds to the case where `x` is an empty list, and the string:

[**nil**,{`p`}] [(`List:red`),{`x`}] [**lim**,∅]

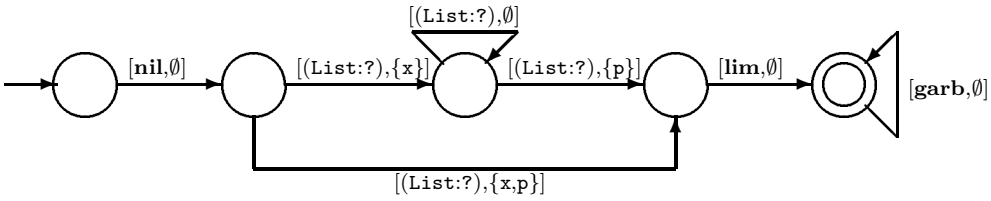
to the case where `x` is a red singleton list and `p` points to the final ***nil*** cell.

Deciding Triples

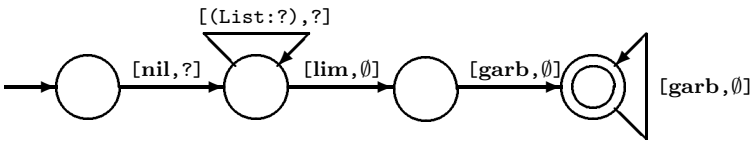
We claim, again without proof, that the *wp*-transformer and the *alloc*-predicate are computable. Let us illustrate with an example based on the supposedly valid triple:

```
{ x<next*>p & p^.next=nil }
new(q,blue);
q^.next:=nil;
p^.next:=q
{ x<next*>q & q^.next=nil & p<>q }
```

The precondition formula describes the stores accepted by the automaton A_{pre}:



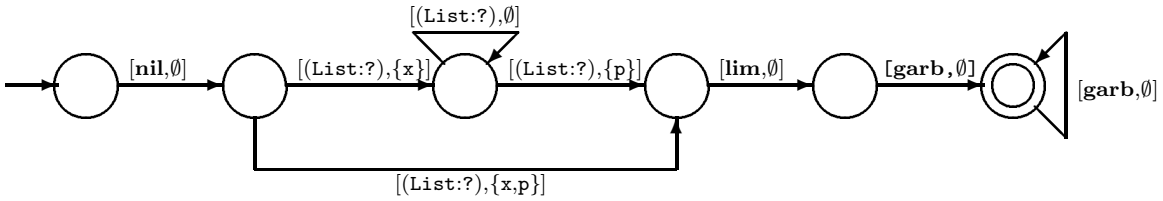
The *alloc*-predicate requires at least one available garbage cell and corresponds to the automaton A_{alloc}:



The weakest precondition is calculated by the technique of *transduction*, where the postcondition is syntactically transformed according to the effect of the program statements. In this technique all basic relationships, such as the successor relation between cells, are accounted for in a predicate after each program statement. The effect of a statement is to transform this collection of predicates. The resulting collection is then used to rewrite the postcondition (and the well-formedness formula), where each reference to a basic relationship is replaced by the corresponding final transformed predicate. For this particular example, a rather large formula results; but it will be equivalent to:

```
x<next*>p & (ex g: <garb?>g) & p^.next=nil
```

which corresponds to the automaton A_{wp}:



A simple computation will now confirm that $A_{pre} \cap A_{alloc} \subseteq A_{wp}$ (for this trivial case they are in fact equal). It follows that the triple is indeed valid.

For general programs this task is quite intricate, since it is also necessary to consider the effects of conditionals and possible type errors and run-time errors. A full version of this paper will contain further details of the predicate transformation.

5 Verifying Programs

Our decision procedure can clearly be used to answer all possible questions about a loop-free program that can be phrased in our logic. This provides a very general tool for analyzing such programs, since it is not limited to answering single, fixed questions such as the absence of dangling references or unclaimed memory cells. It demonstrates that such programs can be completely understood as transformations on regular sets.

Using Invariants

Most interesting programs contain loops. For these we can verify the same class of properties as for loop-free programs, provided we can phrase loop invariants in our logic. Our decision procedure is then not entirely automatic, nor is it complete since it is well-known that a true property of a loop cannot necessarily be proven by an invariant. In practice, however, it is quite easy to phrase useful invariants in our logic.

Let us recapitulate the required proof technique for loops. To verify the triple $\{\Phi_1\}$ *while* **B** *do* **S** $\{\Phi_2\}$ we phrase a loop invariant I and prove validity of the formula $\Phi_1 \Rightarrow I$, the triple $\{I$ & **B**$\}$ **S** $\{I\}$, and the formula I & ~**B** $\Rightarrow \Phi_2$. The generation and verification of these three subgoals is handled automatically by our decision procedure, when the invariant has been given.

As a default, our system uses the basic well-formedness predicate for the invariant. In many cases, this turns out to be sufficient.

Positive Examples

We now show a number of example programs that are successfully verified by our decision procedure. They are ordinary Pascal programs, except that we annotate them with occasional formulas and classify the declared variables as data or pointer variables. In each case we verify that the resulting store is well-formed. This guarantees that we encounter no run-time errors, and that we leave no dangling references or unclaimed memory cells. In some cases, we use the power of our logic to verify further properties. The first example is the program that reverses a list.

```
program reverse;
type Color = (red,blue);
     List = ^Item;
     Item = record
              case tag: Color of
              red,blue: (next: List)
            end;
{data}var x,y: List; {pointer}var p: List;
begin
    {y=nil}
    while x<>nil do
    begin
       p:=x^.next;
       x^.next:=y;
       y:=x;
       x:=p
    end
    {x=nil}
end.
```

This program is particularly well-suited for our analysis, since we do not need to specify an invariant beyond the implicit well-formedness formula. After the loop, we are assured that `x` is empty and that `y` contains a list.

The next program performs a cyclic rotation of a list `x` where `p` points to the last element. We omit the type declarations which are the same in all our examples.

```
program rotate;
{data}var x: List; {pointer}var p: List;
begin
    {x<next*>p & (x<>nil => p^.next=nil)}
    if x<>nil then
    begin
       p^.next:=x;
       x:=x^.next;
       p:=p^.next;
       p^.next:=nil
    end
    {x<next*>p & (x<>nil => p^.next=nil)}
end.
```

Note how the precondition is used to specify the assumptions under which the program works. The postcondition assures that this data type invariant is preserved by the operation.

The following program inserts a red node into a possibly empty list `x` (cyclically) after the position indicated by `p`, taking care of all the special cases.

```
program insert;
{data}var x: List; {pointer}var p,q: List;
begin
    {x<next*>p & (x=nil <=> p=nil)}
    if p<>nil then
    begin
       if p^.next=nil then
       begin
          q:=x^next;
          new(p,red);
          x^.next:=p
       end
       else
       begin
          q:=p^.next;
          new(p^.next,red);
          p:=p^.next
       end
    else
       q:=nil;
```

```
       new(p,red);
       x:=p
    end;
    p^.next:=q
end.
```

We now dually consider a program that deletes the node after p.

```
program delete;
{data}var x: List; {pointer}var p,q: List;
begin
    {x<next*>p & (x=nil <=> p=nil)}
    if p<>nil then
    begin
       if p^.next=nil then
       begin
          q:=x^.next;
          if x^.tag=red then
             dispose(x,red)
          else
             dispose(x,blue);
          x:=q
       end
       else
       begin
          q:=p^.next^.next;
          if x^.tag=red then
             dispose(x,red)
          else
             dispose(x,blue);
          p^.next:=q
       end
    end
    {p<>nil => (ex q: <garb?>q & (all r: <garb?>r => r=q)) &
     p=nil => ~(ex q: <garb?>q)
    }
end.
```

The postcondition verifies that if the list was not empty, then exactly one record has been deallocated; if the list was empty, then no deallocation took place.

We now turn our attention to a program that searches for the first occurrence of a blue node in a list.

```
program search;
{data}var x: List; {pointer}var p: List;
begin
    p:=x;
    while p<>nil and p^.tag<>blue do
       {x<next*>p & (all q: (x<next*>q & q<next*>p) => <(List:red)?>q)}
       p:=p^.next
    end
    {x<next*>p & (p=nil | <(List:blue)?>p) &
     (all q: (x<next*>q & q<next.next*>p) => <(List:red)?>q)
    }
end.
```

To merely verify well-formedness, we do not have to specify an explicit invariant. However, by providing a rich invariant we can automatically verify the *behavior* of this program which is expressed by the postcondition: to find the first blue node if one exists. This illustrates that we can verify properties well beyond standard pointer analyses.

The final program zips two lists into one, by performing a strict shuffle of their elements and appending the tail of the longer list.

```
program zip;
{data}var x,y,z: List; {pointer}var p,t: List;
begin
    if x=nil then
    begin
       t:=x; x:=y; y:=t
    end;
    z:=nil; pz:=nil;
    while x<>nil do
       {(x=nil => y=nil) & z<next*>p & (z<>nil => p^.next=nil)}
    begin
       if z=nil then
       begin
          z:=x; p:=x;
       end
       else
       begin
```

```
          p^.next:=x;
          p:=p^.next
       end;
       x:=x^.next;
       p^.next:=nil;
       if y<>nil then
       begin
          t:=x; x:=y; y:=t
       end
    end
end.
```

For this example we need a seemingly involved invariant to establish well-formedness. However, it merely states that `x` is only empty if `y` is empty, and that `p` points to the last element of `z`.

Negative Examples

We next turn our attention to faulty programs that cannot be verified. Consider first the `reverse` program in which we perform a likely mistake by accidentally switching the second and third program line in the loop body.

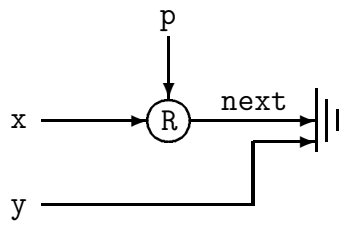
```
program fumble;
{data}var x,y: List; {pointer}var p: List;
begin
    {y=nil}
    while x<>nil do
    begin
       p:=x^.next;
       y:=x;        (* line 3 *)
       x^.next:=y;  (* line 2 *)
       x:=p
    end
    {x=nil}
end.
```

The program is clearly no longer correct, which our decision procedure detects since it is not the case that $\mathcal{L}(\Phi_1) \cap \mathcal{L}(\mathit{alloc}(\mathbf{S})) \subseteq \mathcal{L}(\mathit{wp}(\mathbf{S}, \Phi_2))$. Significantly, we may now obtain more information besides this bare fact, since the set $(\mathcal{L}(\Phi_1) \cap \mathcal{L}(\mathit{alloc}(\mathbf{S}))) \setminus \mathcal{L}(\mathit{wp}(\mathbf{S}, \Phi_2))$ is also a (non-empty) regular language

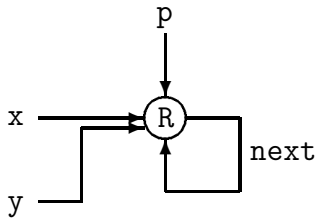
from which we can automatically extract a shortest string. For the `fumble` program such a string is:

[**nil**,{p}] [(List:red),∅] [**lim**,∅] [**lim**,∅]

which corresponds to a particular store:



This initial store is a concrete counterexample on which the program will expose its faulty behavior. We may simulate the program on this store, and after the first iteration we see the error:



We envision a tool in which a programming error will generate and play a small cartoon of store modifications that explains the faulty behavior.

As another example, consider a program that swaps the first two elements of a list.

```
program swap;
{data}var x: List; {pointer}var p: List;
begin
   if x<>nil then
   begin
      p:=x;
      x:=x^.next;
      p^.next:=x^.next;
      x^.next:=p
   end
end.
```

The program is essentially correct, except that it fails by dereferencing a *nil*-pointer in the special case of a list of length one. Correspondingly, our decision procedure responds with the string:

[**nil**,{p}] [(List:red),∅] [**lim**,∅]

which corresponds to a store containing a list of length one. To confirm the hypothesis that this is the only fatal case, we may introduce the precondition {x^.next<>nil} and then successfully verify the program.

6 Implementation

An implementation of our decision procedure needs to:

- compute and represent the regular set of stores that satisfy a given formula;
- compute the predicate transformer *wp* and the predicate *alloc*; and
- decide properties of regular sets.

We have a unifying framework for expressing all these tasks.

Using Monadic Second-Order Logic

Our implementation is based on the monadic second-order logic on finite strings (M2L), which is an inordinately succinct notation for specifying regular sets [16]. It uses formulas similar to but more general than those of our store logic.

It turns out to be a straightforward task to inductively translate formulas of our store logic into equivalent formulas of M2L. The regular set is then represented by an M2L formula.

Also *wp* and *alloc* are elegantly captured through formulas in M2L, where the effect of each program line is simulated with all the appropriate type and run-time tests. The formulas look vaguely like the code for an interpreter.

Finally, all the required properties of regular sets correspond to simple connectives in M2L; for example, set inclusion is implication of the representing formulas.

The net effect is to produce a (possibly huge) M2L formula whose validity coincides with validity of the given triple.

Using Fido and Mona

The M2L approach is fruitful because of the Fido and Mona tools that implement this logic for finite strings and trees [9, 12]. Mona is an engine that reduces an M2L formula to an equivalent finite state automaton. Fido is a high-level specification notation that generates primitive Mona formulas.

The implementation of Mona is feasible because of a special representation of automata, where transition functions are encoded as binary decision diagrams [1] (BDDs). Corresponding to this representation, specialized algorithms for the basic automata operations have been developed. As a result, Mona may efficiently reduce automata with very large alphabets, such as those we encounter in our application.

The implementation of our decision procedure is a pipeline: from the annotated Pascal program we generate a Fido specification, which is translated into a voluminous Mona formula, which is then reduced to a finite state automaton. A complete documentation of the example programs, including the Fido and Mona formulas that are generated by the decision procedure, is available at `http://www.brics.dk/~mis/pointers/`.

Complexity

The theoretical worst-case complexity of our decision procedure is non-elementary, i.e. not bounded by any finite stack of exponentials. This lower bound is inherited from M2L. Fortunately, the worst-case scenario hinges on the use of complex formulas, which are not likely to occur in practice. Using the current implementation of the Fido and Mona tools, we obtain the following statistics for our example programs. The time is measured on a SparcServer 1000; the size of the formula is that of the raw Mona input; and for the largest automaton encountered during the Mona reduction we give the number of states and the number of BDD-nodes in the representation of its transition function.

Program	Time	Formula	States	BDD-nodes
`reverse`	25 seconds	56K	74	297
`insert`	54 seconds	91K	323	1,916
`rotate`	36 seconds	65K	156	981
`delete`	55 seconds	96K	131	623
`search`	52 seconds	92K	167	1,072
`zip`	94 seconds	107K	876	5,611
`fumble`	25 seconds	56K	56	215
`swap`	21 seconds	51K	35	156

These measurements are merely intended to give a rough idea of the complexities of the verification problems. Note how seemingly innocuous pointer manipulations are revealed to possess large state spaces when all possible executions and special cases are considered.

7 Conclusions

We have demonstrated that small programs in our Pascal subset can be verified with great accuracy. Our decision procedure exploits a new approach to pointer analysis by modeling stores as strings and reducing the problem to validity of formulas in monadic second-order logic. Our use of a decidable specification logic for properties of stores with pointers may also be of independent interest.

How may we use and extend this technique? We present the answers to a number of pertinent questions.

Can we include trees? The well-formedness requirement insists on disjoint lists. However, our decision procedure is based on encoding stores as finite strings for which the monadic second-order logic is decidable. While we cannot encode trees in this string logic, we can use the fact that also the monadic second-order logic of trees is decidable and implemented by Fido and Mona. While this extension is conceptually and computationally more involved, all our theoretical results still hold [11].

Can we go beyond trees? Some extensions are possible, specifically *graph types* [10] which include doubly-linked and cyclic structures. However, this extension requires an invasive modification of the Pascal syntax. Also, there are clear limitations; for example, on grid structures the store logic is no longer decidable.

What about modularity? It is very easy to perform a modular analysis of programs if the interface between two code fragments can be specified as an intermediate formula. In this case, a triple breaks into two smaller triples.

What about real programs? We have restricted the allowed syntax in several ways. However, the only non-trivial one is the exclusion of integers and arithmetic. Our decision procedure will never be complete for arithmetic, but a canonical approximative analysis may be performed by abstracting the type of integers into a singleton enumeration type, or even into a finite range with modulo arithmetic.

Where will this be used? It is unlikely that our technique will verify a huge, preexisting program. However, when implementing an abstract data type for a library, it should be possible to state the required invariants to obtain an automatic verification of the operations. Also, our tool seems ideal for a teaching environment, since it encourages formal reasoning and provides counterexamples for faulty programs.

References

[1] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, August 1986.

[2] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *of the 7th International on Languages and Compilers for Parallel Computing*, number 892 in Lecture Notes in Computer Science, August 1994.

[3] David L. Detlefs. An overview of the extended static checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice*. ACM SIGSOFT, January 1996.

[4] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *of the ACM SIGPLAN '94 on Programming Language Design and Implementation*, June 1994.

[5] Alain Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In *of the ACM SIGPLAN on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, June 1995.

[6] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of Symposium on the Foundations of Software Engineering*. ACM SIGSOFT, December 1994.

[7] Rakesh Ghiya and Laurie J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A shape analysis for heap-directed pointers in C. In *the 23rd ACM SIGPLAN-SIGACT on Principles of Programming Languages*, January 1996.

[8] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEETPDS*, 1(1):35–47, January 1990.

[9] Jesper Gulmann Henriksen, Michael Jørgensen, Jakob Jensen, Nils Klarlund, Bob Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings TACAS'95, LNCS 1019*, May 1995.

[10] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *the Twentieth Annual ACM SIGPLAN-SIGACT on Principles of Programming Languages*, January 1993.

[11] Nils Klarlund and Michael I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In *Proc. CAAP' 94 (TAPSOFT)*, 1994.

[12] Nils Klarlund and Michael I. Schwartzbach. Regularity = Logic + Recursive Data Types. BRICS, University of Aarhus, October 1996.

[13] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *of the ACM SIGPLAN '92 on Programming Language Design and Implementation*, June 1992.

[14] John Plevyak, Andrew A. Chien, and Vijay Karamcheti. Analysis of dynamic structures for efficient parallel execution. In *of the 6th International on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, August 1993.

[15] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *the 23rd*

ACM SIGPLAN-SIGACT on Principles of Programming Languages, January 1996.

[16] Wolfgang Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

[17] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *of the ACM SIGPLAN '95 on Programming Language Design and Implementation*, June 1995.