Compile-Time Debugging of C Programs Working on Trees

Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach

BRICS, University of Aarhus
{elgaard,amoeller,mis}@brics.dk

Abstract. We exhibit a technique for automatically verifying the safety of simple C programs working on tree-shaped data structures. We do not consider the complete behavior of programs, but only attempt to verify that they respect the shape and integrity of the store. A verified program is guaranteed to preserve the tree-shapes of data structures, to avoid pointer errors such as NULL dereferences, leaking memory, and dangling references, and furthermore to satisfy assertions specified in a specialized store logic.

A program is transformed into a single formula in WSRT, an extension of WS2S that is decided by the MONA tool. This technique is complete for loop-free code, but for loops and recursive functions we rely on Hoare-style invariants. A default well-formedness invariant is supplied and can be strengthened as needed by programmer annotations. If a program fails to verify, a counterexample in the form of an initial store that leads to an error is automatically generated.

This extends previous work that uses a similar technique to verify a simpler syntax manipulating only list structures. In that case, programs are translated into WS1S formulas. A naive generalization to recursive data-types determines an encoding in WS2S that leads to infeasible computations. To obtain a working tool, we have extended MONA to directly support recursive structures using an encoding that provides a necessary state-space factorization. This extension of MONA defines the new WSRT logic together with its decision procedure.

1 Introduction

Catching pointer errors in programs is a difficult task that has inspired many assisting tools. Traditionally, these come in three flavors. First, tools such a Purify [3] and Insure++ [17] instrument the generated code to monitor the runtime behavior thus indicating errors and their sources. Second, traditional compiler technologies such as program slicing [21], pointer analysis [7], and shape analysis [19] are used in tools like CodeSurfer [8] and Aspect [10] that conservatively detect known causes of errors. Third, full-scale program verification is attempted by tools like LCLint [6] and ESC [5], which capture runtime behavior as formulas and then appeal to general theorem provers.

All three approaches lead to tools that are either incomplete or unsound (or both), even for straight-line code. In practice, this may be perfectly acceptable if a significant number of real errors are caught.

In previous work [11], we suggest a different balance point by using a less expressive program logic for which Hoare triples on loop-free code is decidable when integer arithmetic is ignored. That work is restricted by allowing only a `while`-language working on linear lists. In the present paper we extend our approach by allowing recursive functions working on recursive data-types. This generalization is conceptually simple but technically challenging, since programs must now be encoded in WS2S rather than the simpler WS1S. Decision procedures for both logics are provided by the MONA tool [13, 18] on which we rely, but a naive generalization of the previous encoding leads to infeasible computations. We have responded by extending MONA to directly support a logic of recursive data-types, which we call WSRT. This logic is encoded in WS2S in a manner that exploits the internal representation of MONA automata to obtain a much needed state-space factorization.

Our resulting tool catches all pointer errors, including NULL dereferences, leaking memory, and dangling references. It can also verify assertions provided by the programmer in a special store logic. The tool is sound and complete for loop-free code including `if`-statements with restricted conditions: it will reject exactly the code that may cause errors or violate assertions when executed in some initial store. For `while`-loops or functions, the tool relies on annotations in the form of invariants and pre- and post-conditions. In this general case, our tool is sound but incomplete: safe programs exist that cannot be verified regardless of the annotations provided. In practical terms, we provide default annotations that in many cases enable verification.

Our implementation is reasonably efficient, but can only handle programs of moderate sizes, such as individual operations of data-types. If a program fails to verify, a counterexample is provided in the form of an initial store leading to an error. A special simulator is supplied that can trace the execution of a program and provide graphical snapshots of the store. Thus, a reasonable form of compile-time debugging is made available. While we do not detect all program errors, the verification provided serves as a finely masked filter for most bugs.

As an example, consider the following recursive data-type of binary trees with red, green, or blue nodes:

```
struct RGB {
  enum {red,green,blue} color;
  struct RGB *left;
  struct RGB *right;
};
```

The following non-trivial application collects all green leaves into a right-linear tree and changes all the blue nodes to become red:

```
/**data**/ struct RGB *tree;
/**data**/ struct RGB *greens;

enum bool {false,true};

enum bool greenleaf(struct RGB *t) {
  if (t==0) return false;
  if (t->color!=green) return false;
  if (t->left!=0 || t->right!=0) return false;
  return true;
}

void traverse(struct RGB *t) {
  struct RGB *x;
  if (t!=0) {
    if (t->color==blue) t->color = red;
    if (greenleaf(t->left)==true /**keep: t!=0 **/) {
      t->left->right = greens;
      greens = t->left;
      t->left=0;
    }
    if (greenleaf(t->right)==true /**keep: t!=0 **/) {
      t->right->right = greens;
      greens = t->right;
      t->right=0;
    }
    traverse(t->left); /**keep: t!=0 **/
    traverse(t->right); /**keep: t!=0 **/
  }
}

/**pre: greens==0 **/
main()  { traverse(tree); }
```

The special comments are assertions that the programmer must insert to specify the intended model (`/**data**/`), restrict the set of stores under consideration (`/**pre**/`), or aid the verifier (`/**keep**/`). They are explained further in Section 2.4.

Without additional annotations, our tool can verify this program (in 33 seconds on a 266MHz Pentium II PC with 128 MB RAM). This means that no pointer errors occur during execution from any initial store. Furthermore, both `tree` and `greens` are known to remain well-formed trees. Using the assertion:

```
  all p: greens(->left + ->right)*==p => (p!=0 => p->color==green)
```

we can verify (in 74 seconds) that `greens` after execution contains only green nodes. That `greens` is right-linear is expressed through the assertion:

```
  all p: greens(->left + ->right)*==p => (p!=0 => p->left==0)
```

In contrast, if we assert that `greens` ends up empty, the tool responds with a minimal counterexample in the form of an initial store in which `tree` contains a green leaf.

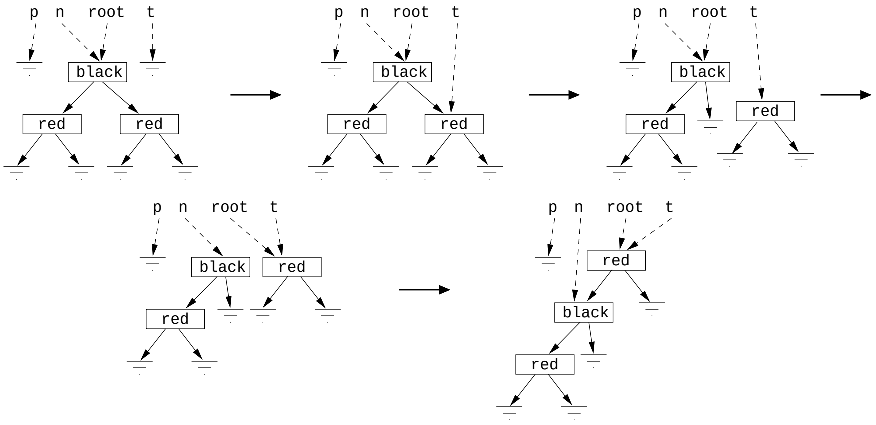
An example of the simulator used in conjunction with counterexamples comes from the following fragment of an implementation of red-black search trees. Consider the following program, which performs a left rotation of a node `n` with parent `p` in such a tree:

```
struct Node {
  enum {red, black} color;
  struct Node *left;
  struct Node *right;
};

/**data**/ struct Node *root;

/**pre: n!=0 & n->right!=0 &
        (p!=0 => (p->left==n | p->right==n)) &
        (p==0 => n==root) **/
void left_rotate(struct Node *n, struct Node *p) {
  struct Node *t;
  t = n->right;
  n->right = t->left;
  if (n==root) root = t;
  else if (p->left==n) p->left = t;
  else p->right = t;
  t->left = n;
}
```

In our assertion language, we cannot express the part of the red-black data-type invariant that each path from the root to a leaf must contain the same number of black nodes; however we can capture the part that the root is black and that a red node cannot have red children:

```
root->color==black &
all p: p->color==red =>
       (p->left->color!=red & p->right->color!=red)
```

If we add the above assertion as a data-type invariant, we are (in 18 seconds) given a counterexample. If we apply the simulator, we see the following example run, which shows that we have forgotten to consider that the root may become red (in which case we should add a line of code coloring it black):



Such detailed feedback at compile-time is clearly a useful debugging tool.

2 The Language

The language in consideration is a simple yet non-trivial subset of C. It allows declaration of tree-shaped recursively typed data structures and recursive imperative functions operating on the trees. The subset is chosen such that the verification we intend to perform becomes decidable. Thus, for instance, integer arithmetic is omitted from the language; only finite enumeration types can be expressed. Also, to smoothen presentation, many other C constructs have been omitted although some of them easily could be included directly, and other by approximating their behavior.

We begin by defining the core language. After that, we describe how programs can be annotated with formulas expressing additional requirements for correctness.

2.1 The C Subset

The abstract syntax of the C subset is defined using EBNF notation, where furthermore ⊛ is used to denote comma-separated lists with zero or more elements. The semantics of the language is as known from C.

A program consists of declarations of structures, enumerations, variables, and functions:

$$program \;\rightarrow\; (\; struct \mid enum \mid var \mid function\;)^*$$

A structure contains an enumeration denoting its value and a union of structures containing pointers to its child structures. An enumeration is a list of identifiers:

```
struct   →   struct id {
               enum id id;
               union {
                 ( struct {
                   ( struct id * id; )*
                   } id; )*
               } id;
             };
enum     →   enum id { id+ };
```

The enumeration values denote the *kind* of the structure, and the kind determines which is the *active* union member. The association between enumeration values and union members is based on their indices in the two lists. Such data structures are typical in real-world C programs and exactly define recursive data-types. One goal of our verification is to ensure that only active union members are accessed.

For abbreviation we allow declarations of structures and enumerations to be inlined. Also, we allow (`struct` *id* `*` *id*`;` $)^*$ in place of `union` {. . . }, implicitly meaning that all union members are identical. A variable is either a pointer to a structure or an enumeration:

$$\begin{array}{lll} \mathit{var} & \rightarrow & \mathit{type}\ \mathit{id}\texttt{;} \\ \mathit{type} & \rightarrow & \texttt{struct}\ \mathit{id}\ \texttt{*} \ \mid\ \texttt{enum}\ \mathit{id} \end{array}$$

A function can contain variable declarations and statements:

$$\begin{array}{lll} \mathit{function} & \rightarrow & (\ \texttt{void} \ \mid\ \mathit{type}\ )\ \mathit{id}\texttt{(}\ (\ \mathit{type}\ \mathit{id}\ )^{\circledast}\ \texttt{)}\ \texttt{\{} \\ & & \quad \mathit{var}^{*}\ \mathit{stm}^{?} \\ & & \quad (\ \texttt{return}\ \mathit{rvalue}\texttt{;}\ )^{?} \\ & & \texttt{\}} \end{array}$$

A statement is a sequence, an assignment, a function call, a conditional statement, a `while`-loop, or a memory deallocation:

$$\begin{array}{lll} \mathit{stm} & \rightarrow & \mathit{stm}\ \mathit{stm} \ \mid \\ & & \mathit{lvalue}\ \texttt{=}\ \mathit{rvalue}\texttt{;} \ \mid \\ & & \mathit{id}\texttt{(}\ (\ \mathit{rvalue}\ )^{\circledast}\ \texttt{);} \ \mid \\ & & \texttt{if (}\ \mathit{cond}\ \texttt{)}\ \mathit{stm}\ (\ \texttt{else}\ \mathit{stm}\ )^{?} \ \mid \\ & & \texttt{while (}\ \mathit{cond}\ \texttt{)}\ \mathit{stm} \ \mid \\ & & \texttt{free(}\ \mathit{lvalue}\ \texttt{);} \end{array}$$

A condition is a boolean expression evaluating to either true or false; the expression `?` represents non-deterministic choice and can be used in place of those C expressions that are omitted from our subset language:

$$\mathit{cond} \quad \rightarrow \quad \mathit{cond}\ \texttt{\&}\ \mathit{cond} \ \mid\ \mathit{cond}\ \texttt{|}\ \mathit{cond} \ \mid\ \texttt{!}\ \mathit{cond} \ \mid\ \mathit{rvalue}\ \texttt{==}\ \mathit{rvalue} \ \mid\ \texttt{?}$$

An *lvalue* is an expression designating an enumeration variable or a pointer variable. An *rvalue* is an expression evaluating to an enumeration value or to a pointer to a structure. The constant `0` is the NULL pointer, `malloc` allocates memory on the heap, and *id*`(...)` is a function call:

$$\begin{array}{lll} \mathit{lvalue} & \rightarrow & \mathit{id}\ (\ \texttt{->}\ \mathit{id}\ (\ \texttt{.}\ \mathit{id}\ )^{?}\ )^{*} \\ \mathit{rvalue} & \rightarrow & \mathit{lvalue} \ \mid\ \texttt{0} \ \mid\ \texttt{malloc(sizeof(}\ \mathit{id}\ \texttt{))} \ \mid\ \mathit{id}\texttt{(}\ \mathit{rvalue}^{\circledast}\ \texttt{)} \end{array}$$

The nonterminal *id* represents identifiers.

The presentation of our verification technique is based on C for familiarity reasons only—no intrinsic C constructs are utilized.

2.2 Modeling the Store

During execution of a program, structures located in the heap are allocated and freed, and field variables and local variables are assigned values. The state of an execution can be described by a model of the heap and the local variables, called the *store*.

A store is modeled as a finite graph, consisting of a set of *cells* representing structures, a distinguished *NULL cell*, a set of *program variables*, and *pointers* from cells or program variables to cells. Each cell is labeled with a *value* taken from the enumerations occurring in the program. Furthermore, each cell can have a *free* mark, meaning that it is currently not allocated.

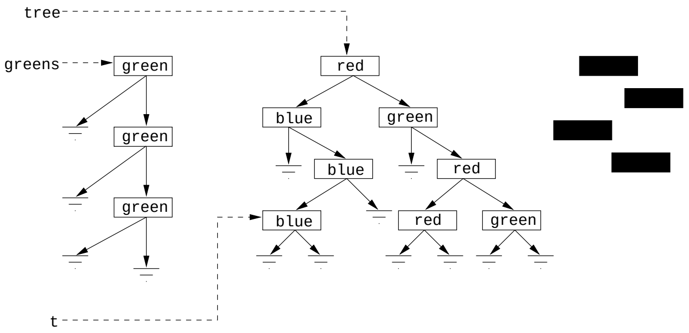
Program variables are those that are declared in the program either globally or inside functions. To enable the verification, we need to classify these variables as either *data* or *pointer* variables. A variable is classified as a data variable by prefixing its declaration in the program with the special comment `/**data**/`; otherwise, it is considered a pointer variable.

A store is *well-formed* if it satisfies the following properties:

- the cells and pointers form disjoint tree structures (the NULL cell may be shared, though);
- each data variable points either to the root of a tree or to the NULL cell;
- each pointer variable points to any cell (including the NULL cell);
- a cell is marked as free if and only if it is not reachable from a program variable; and
- the type declarations are respected—this includes the requirement that a cell representing a structure has an outgoing pointer for each structure pointer declared in its active union member.

With the techniques described in the remainder of this paper, it is possible to automatically verify whether well-formedness is preserved by all functions in a given program. Furthermore, additional user defined properties expressed in the logic presented in Section 2.3 can be verified.

The following illustrates an example of a well-formed store containing some `RGB`-trees as described in Section 1. Tree edges are solid lines whereas the values of pointer variables are dashed lines; free cells are solid black:



2.3 Store Logic

Properties of stores can conveniently be stated using logic. The declarative and succinct nature of logic often allows simple specifications of complex requirements. The logic presented here is essentially a first-order logic on finite tree structures [20]. It has the important characteristic of being decidable, which we will exploit for the program verification.

A formula ϕ in the store logic is built from boolean connectives, first-order quantifiers, and basic propositions. A term t denotes either an enumeration value or a pointer to a cell in the store. A path set P represents a set of paths, where a path is a sequence of pointer dereferences and union member selections ending in either a pointer or an enumeration field. The signature of the logic consists of dereference functions, path relations, and the relations `free` and `root`:

$$\begin{array}{lll} \phi & \rightarrow & \texttt{!}\ \phi \ \mid\ \phi\ \texttt{\&}\ \phi \ \mid\ \phi\ \texttt{|}\ \phi \ \mid\ \phi\ \texttt{=>}\ \phi \ \mid\ \phi\ \texttt{<=>}\ \phi \ \mid \\ & & \texttt{ex}\ id\ \texttt{:}\ \phi \ \mid\ \texttt{all}\ id\ \texttt{:}\ \phi \ \mid\ \texttt{true} \ \mid\ \texttt{false} \ \mid \\ & & id\ \texttt{(}\ P\ \texttt{)}^{?}\ \texttt{==}\ t \ \mid\ \texttt{free(}\ t\ \texttt{)} \ \mid\ \texttt{root(}\ t\ \texttt{)} \end{array}$$

A path relation, $id\ P$ `==` t, compares either enumeration values or cell pointers. The identifier id may be either a bound quantified variable, a program variable, or an enumeration value, and t is a term.

If both id and t denote cell pointers, a path relation is true for a given store if there is a path in P from the cell denoted by id to the cell denoted by t in the store. If P is omitted, the relation is true if id and t denote the same cell.

If id denotes a cell pointer and t is an enumeration value, a path relation is true for a given store if there is a path satisfying P from the cell denoted by id to an enumeration field with the value t in the store.

The relation `free(`t`)` is true in a given store if the cell denoted by t is marked as not allocated in the store. The relation `root(`t`)` is true if t denotes the root of some tree.

A term is a sequence of applications of the dereference function and union member selections or the constant `0` representing the special NULL cell:

$$t \quad \rightarrow \quad id\ \texttt{(}\ \texttt{->}\ id\ \texttt{(}\ \texttt{.}\ id\ \texttt{)}^{?}\ \texttt{)}^{*} \ \mid\ \texttt{0}$$

A path set is a regular expression:

$$P \quad \rightarrow \quad \texttt{->}\ id\ \texttt{(}\ \texttt{.}\ id\ \texttt{)}^{?} \ \mid\ P\ \texttt{+}\ P \ \mid\ P\ P \ \mid\ P\ \texttt{*}$$

The path set defined by `->`id_1`.`id_2 consists of a single dereference of id_1 and subsequent selection of the member id_2. The expressions P `+` P, $P\ P$, and P `*` respectively denote union, concatenation, and Kleene star.

2.4 Program Annotations and Hoare Triples

The verification technique is based on Hoare triples [9], that is, constructs of the form $\{\phi_1\}stm\{\phi_2\}$. The meaning of this triple is that executing the statement stm in a store satisfying the pre-condition ϕ_1 always results in a store satisfying the post-condition ϕ_2, provided that the statement terminates. Well-formedness is always implicitly included in both ϕ_1 and ϕ_2. We can only directly decide such triples for loop-free code. Programs containing loops—either as `while`-loops or as function calls—must be split into loop-free fragments.

A program can be annotated with formulas expressing requirements for correctness using a family of designated comments. These annotations are also used to split the program into a set of Hoare triples that subsequently can be verified separately.

`/**pre:` ϕ `**/` and `/**post:` ϕ `**/` may be placed between the signature and the body of a function. The `pre` formula expresses a property that the verifier may assume initially holds when the function is executed. The `post` formula expresses a property intended to hold after execution of the function. The states before and after execution may be related using otherwise unused variables.

/**inv: ϕ **/ may be placed between the condition and the body of a while-loop. It expresses an invariant property that must hold before execution of the loop and after each iteration. It splits the code into three parts: the statements preceding the while-loop, its body, and the statements following it.

/**keep: ϕ **/ may be placed immediately after a function call. It expresses a property that must hold both before and after the call. It splits the code into two parts: the statements before and after the call. The keep formulas can specify invariants for recursive function calls just as inv formulas can specify invariants for while-loops.

/**assert: ϕ **/ may be placed between statements. It splits the statement sequence, such that a Hoare triple is divided into two smaller triples, where the post-condition of the first and the pre-condition of the second both are ϕ. This allows modular analysis to be performed. The variant /**assert: ϕ assume: ϕ **/ allows the post-condition and the pre-condition to be different, and thereby to weaken the verification requirements. This is needed whenever a sufficiently strong property either cannot be expressed or requires infeasible computations.

/**check: ϕ **/ *stm* informally corresponds to "if (!ϕ) fail; else *stm*", where fail is some statement that fails to verify. This can be used to check that a certain property holds without creating two Hoare triples incurring a potential loss of information.

Whenever a pre- or post-condition, an invariant, or a keep-formula is omitted, the default formula true is implicitly inserted. Actually, many interesting properties can be verified with just these defaults. As an example, the program:

```
/**data**/ struct RGB *x;
struct RGB *p;
struct RGB *q;

p = x;
q = 0;
while (p!=0 & q==0) /**inv: q!=0 => q->color==red **/ {
  if (p->color==red) q = p;
  else if (p->color==green) p = p->left;
  else /**assert: p->color==blue **/ p = p->right;
}
```

yields the following set of Hoare triples and logical implications to be checked:

```
{ true } p = x; q = 0; { I }
( I & !B ) => true
{ I & B & B1 } q = p; { I }
{ I & B & !B1 & B2 } p = p->left; { I }
( I & B & !B1 & !B2 ) => ( p->color==blue )
{ I & B & !B1 & !B2 & p->color==blue } p = p->right; { I }
```

where `B` is the condition of the `while`-loop, `I` is the invariant, `B1` is the condition of the outer `if`-statement and `B2` that of the inner `if`-statement. Note that the generated Hoare triples are completely independent of each other—when a triple is divided into two smaller triples, no information obtained from analyzing the first triple is used when analyzing the second.

3 Deciding Hoare Triples

The generated Hoare triples and logical implications—both the formula parts and the program parts—can be encoded in the logic WS2S which is known to be decidable. This encoding method follows directly from [11] by generalizing from list structures to tree structures in the style of [16]. The MONA tool provides an implementation of a decision procedure for WS2S, so in principle making a decision procedure for the present language requires no new ideas.

As we show in the following, this method will however lead to infeasible computations making it useless in practice. The solution is to exploit the full power of the MONA tool: usually, WS2S is decided using a correspondence with ordinary tree automata—MONA uses a representation called *guided tree automata*, which when used properly can be exponentially more efficient than ordinary tree automata. However, such a gain requires a non-obvious encoding.

We will not describe how plain MONA code directly can be generated from the Hoare triples and logical implications. Instead we introduce a logic called *WSRT*, *weak monadic second-order logic with recursive types*, which separates the encoding into two parts: the Hoare triples and logical implications are first encoded in WSRT, and then WSRT is translated into basic MONA code. This has two benefits: WSRT provides a higher level of abstraction for the encoding task, and, as a by-product, we get an efficient implementation of a general tree logic which can be applied in many other situations where WS2S and ordinary tree automata have so far been used.

3.1 Weak Monadic Second-Order Logic with Recursive Types

A *recursive type* is a set of recursive equations of the form:

$$T_i = v_1(c_{1,1} : T_{j_{1,1}}, \ldots, c_{1,m_1} : T_{j_{1,m_1}}), \ldots, v_n(c_{n,1} : T_{j_{n,1}}, \ldots, c_{n,m_n} : T_{j_{n,m_n}})$$

Each T denotes the name of a type, each v is called a *variant*, and each c is called a *component.* A tree conforms to a recursive type T if its root is labeled with a variant v from T and it has a successor for each component in v such that the successor conforms to the type of that component. Note that types defined by `structs` in the language in Section 2.1 exactly correspond to such recursive types.

The logic WSRT is a weak monadic second-order logic. Formulas are interpreted relative to a set of trees conforming to recursive types. Each node is labeled with a variant from a recursive type. A tree variable denotes a tree conforming to a fixed recursive type. A first-order variable denotes a single node. A second-order variable denotes a finite set of nodes.

A formula is built from the usual boolean connectives, first-order and weak monadic second-order quantifiers, and the special WSRT basic formulas:

$type(t, T)$ which is true iff the the first-order term t denotes a node which is labeled with some variant from the type T; and

$variant(t, x, T, v)$ which is true iff the tree denoted by the tree variable x at the position denoted by t is labeled with the T variant v.

Second-order terms are built from second-order variables and the set operations union, intersection and difference. First-order terms are built from first-order variables and the special WSRT functions:

$tree\_root(x)$ which evaluates to the root of the tree denoted by x; and

$succ(t, T, v, c)$ which, provided that the first-order term t denotes a node of the T variant v, evaluates to its c component.

This logic corresponds to the core of the FIDO language [16] and is also reminiscent of the LISA language [1]. It can be reduced to WS2S and thus provides no more expressive power, but we will show that a significantly more efficient decision procedure exists if we bypass WS2S.

3.2 Encoding Stores and Formulas in WSRT

The idea behind the decision procedure for Hoare triples is to encode well-formed stores as trees. The effect of executing a loop-free program fragment is then in a finite number of steps to transform one tree into another. WSRT can conveniently be used to express regular sets of finite trees conforming to recursive types, which turns out to be exactly what we need to encode pre- and post-conditions and effects of execution.

We begin by making some observations that simplify the encoding task. First, note that NULL pointers can be represented by adding a “NULL kind” with no successors to all structures. Second, note that memory allocation issues can be represented by having a “free list” for each `struct`, just as in [11]. We can now represent a well-formed store by a set of WSRT variables:

- each data variable is represented by a WSRT tree variable with the same recursive type, where we use the fact that the types defined by `struct`s exactly correspond to the WSRT notion of recursive types; and
- each pointer variable in the program is represented by a WSRT first-order variable.

For each program point, a set of WSRT predicates called *store predicates* is used to express the possible stores:

- for each data variable d in the program, the predicate $root_d(t)$ is true whenever the first-order term t denotes the root of d;

- for each pointer variable p, the predicate $pos_p(t)$ is true whenever t and p denote the same position;
- for each pointer field f occurring in a union u in some structure s, the predicate $succ_{f,u,s}(t_1, t_2)$ is true whenever the first-order term t_1 points to a cell of type s having the value u, and the f component of this cell points to the same node as the first-order term t_2;
- for each possible enumeration value e, the predicate $kind_e(t)$ is true whenever t denotes a cell with value e; and
- to encode allocation status, the predicate $free_s(t)$ is true whenever t denotes a non-allocated cell.

A set of store predicates called the *initial store predicates* defining a mapping of the heap into WSRT trees can easily be expressed in the WSRT logic. For instance, the initial store predicates *root*, *succ*, and *kind* simply coincide with the corresponding basic WSRT constructs.

Based on a set of store predicates, the well-formedness property and all store-logic formulas can be encoded as other predicates. For well-formedness, the requirements of the recursive types are expressed using the *root*, *kind*, and *succ* predicates, and the requirement that all data structures are disjoint trees is a simple reachability property. For store-logic formulas, the construction is inductive: boolean connectives and quantifiers are directly translated into WSRT; terms are expressed using the store predicates *root*, *kind*, and *succ*; and the basic formulas `free(`t`)` and `root(`t`)` can be expressed using the store predicates $free$ and *root*. Only the regular path sets are non-trivial; they are expressed in WSRT using the method from [14] (where path sets are called "routing expressions"). Note that even though the logic in Section 2.3 is a first-order logic, we also need the weak monadic second-order fragment of WSRT to express well-formedness and path sets.

3.3 Predicate Transformation

When the program has been broken into loop-free fragments, the Hoare triples are decided using the transduction technique introduced in [15]. In this technique, the effect of executing a loop-free program fragment is simulated, step by step, by transforming store predicates accordingly, as described in the following.

Since the pre-condition of a Hoare triple always implicitly includes the well-formedness criteria, we encode the set of *pre-stores* as the conjunction of well-formedness and the pre-condition, both encoded using the initial store predicates, and we initiate the transduction with the initial store predicates. For each step, a new set of store predicates is defined representing the possible stores after executing that step. This predicate transformation is performed using the same ideas as in [11], so we omit the details.

When all steps in this way have been simulated, we have a set of *final* store predicates which exactly represents the changes made by the program fragment. We now encode the set of *post-stores* as the conjunction of well-formedness and the post-condition, both encoded using the final store predicates. It can be shown that the resulting predicate representing the post-stores coincides with the weakest precondition of the code and the post-condition. The Hoare triple is satisfied if and only if the encoding of the pre-stores implies the encoding of the post-stores.

Our technique is sound: if verification succeeds, the program is guaranteed to contain no errors. For loop-free Hoare triples, it is also complete. That is, every effect on the store can be expressed in the store logic, and this logic is decidable. In general, no approximation takes place—all effects of execution are simulated precisely. Nevertheless, since not all true properties of a program containing loops can be expressed in the logic, the technique is in general not complete for whole programs.

4 Deciding WSRT

As mentioned, there is a simple reduction from WSRT to WS2S, and WS2S can be decided using a well-known correspondence between WS2S formulas and ordinary tree automata. The resulting so-called *naive* decision procedure for WSRT is essentially the same as the ones used in FIDO and LISA and as the "conventional encoding of parse trees" in [4]. The naive decision procedure along with its deficiencies is described in Section 4.1. In Section 4.2 we show an efficient decision procedure based on the more sophisticated notion of guided tree automata.

4.1 The Naive Decision Procedure

WS2S, the weak monadic second-order logic of two successors, is a logic that is interpreted relative to a binary tree. A first-order variable denotes a single node in the tree, and a second-order variable denotes a finite set of nodes. For a full definition of WS2S, see [20] or [13].

The decision procedure implemented in MONA inductively constructs a tree automaton for each sub-formula, such that the set of trees accepted by the automaton is the set of interpretations that satisfy the sub-formula. This decision procedure not only determines validity of formulas; it also allows construction of counterexamples whenever a formula is not valid.

Note that the logic WSnS, where each node has n successors instead of just two, easily can be encoded in WS2S by replacing each node with a small tree with n leaves. The idea in the encoding is to have a one-to-one mapping from nodes in a WSRT tree to nodes in a WSnS tree, where we choose n as the maximal fanout of all recursive types.

Each WSRT tree variable x is now represented by b second-order variables $v_1, \ldots, v_b$ where b is the number of bits needed to encode the possible type variants. For each node in the n-ary tree, membership in $v_1 \ldots v_b$ represents some binary encoding of the label of the corresponding node in the x tree.

Using this representation, all the basic WSRT formulas and functions can now easily be expressed in WSnS. We omit the details. For practical applications, this method leads to intractable computations requiring prohibitive amounts of time and space. Even a basic concept such as *type well-formedness* yields immense automata. Type well-formedness is the property that the values of a given set of WS2S variables do represent a tree of a particular recursive type.

This problem can be explained as follows. The WS2S encoding is essentially the same as the "conventional encoding of parse trees" in [4], and type well-formedness corresponds to grammar well-formedness. In that paper, it is shown that the number of states in the automaton corresponding to the grammar well-formedness predicate is linear in the size of the grammar, which in our case corresponds to the recursive types. As argued e.g. in [12], tree automata are at least quadratically more difficult to work with than string automata, since the transition tables are two-dimensional as opposed to one-dimensional. This inevitably causes a blowup in time and space requirements for the whole decision procedure.

By this argument, it would be pointless making an implementation based on the described encoding. This claim is supported by experiments with some very simple examples; in each case, we experienced prohibitive time and space requirements.

4.2 A Decision Procedure using Guided Tree Automata

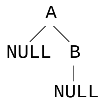
The MONA implementation of WS2S provides an opportunity to factorize the state-space and hence make implementation feasible. To exploit this we must, however, change the encoding of WSRT trees, as described in the following.

The notion of guided tree automata (GTA) was introduced in [2] to combat state-space explosions and is now fully implemented in MONA [13]. A GTA is a tree automaton equipped with separate state spaces that—independently of the labeling of the tree—are assigned to the tree nodes by a top-down automaton, called the *guide*. The secret behind a good factorization is to create the right guide.

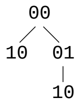
A recursive type is essentially also a top-down automaton, so the idea is to derive a guide from the recursive types. This is however not possible with the naive encoding, since the type of a WSnS node depends on the actual value of its parent node.

Instead of using the one-to-one mapping from WSRT tree nodes to WSnS tree nodes labeled with type variants, we represent a WSRT tree entirely by the *shape* of a WSnS tree, similarly to the "shape encoding" in [4]. Each node in the WSRT tree is represented by a WSnS node with a successor node for each variant, and each of these nodes have themselves a successor for each component in the variant. A WSRT tree is then represented by a *single* second-order WSnS variable whose value indicates the active variants.

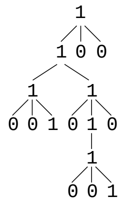
The following illustrates an example of a tree conforming to the recursive type `Tree=A(left:Tree,right:Tree),B(next:Tree),NULL` and its encodings:



(a) a tree



(b) its naive encoding



(c) its shape encoding

This encoding has the desired property that a WSnS tree position always is assigned the same type, independently of the tree values, so a GTA guide can directly be derived from the types. This guide factorizes the state space such that all variants and components in the recursive types have their own separate state spaces. Furthermore, the intermediate nodes caused by the WSnS to WS2S transformation can now also be given separate state spaces, causing yet a degree of factorization.

One consequence is that type well-formedness now can be represented by a GTA with a constant number of states in each state space. The size of this automaton is thus reduced from quadratic to linear in the size of the type. Similar improvements are observed for other predicates.

With these obstacles removed, implementation becomes feasible with typical data-type operations verified in seconds. In fact, for the linear sub-language, our new decision procedure is almost as fast as the previous WS1S implementation; for example, the programs `reverse` and `zip` from [11] are now verified in 2.3 and 29 seconds instead of the previous times of 2.7 and 10 seconds (all using the newest version of MONA). This is remarkable, since our decision procedure suffers a quadratic penalty from using tree automata rather than string automata.

5 Conclusion

By introducing the WSRT logic and exploiting novel features of the MONA implementation, we have built a tool that catches pointer errors in programs working on recursive data structures. Together with assisting tools for extracting counterexamples and graphical program simulations, this forms the basis for a compile-time debugger that is sound and furthermore complete for loop-free code. The inherent non-elementary lower bound of WSnS will always limit its applicability, but we have shown that it handles some realistic examples.

Among the possible extensions or variations of the technique are allowing parent and root pointers in all structures, following the ideas from [14], and switching to a finer store granularity to permit casts and pointer arithmetic. A future implementation will test these ideas. Also, it would be interesting to perform a more detailed comparison of the technique presented here with pointer analysis and shape analysis techniques.

References

[1] Abdelwaheb Ayari, David Basin, and Andreas Podelski. LISA: A specification language based on WS2S. In *Proceedings of CSL'97*. BRICS, 1997.

[2] 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*. Springer Verlag, 1996.

[3] Rational Software Corporation. Purify. URL: `http://www.rational.com/`.

[4] Niels Damgaard, Nils Klarlund, and Michael I. Schwartzbach. YakYak: Parsing with logical side constraints. In *Proceedings of DLT'99*, 1999.

[5] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. URL: `http://www.research.digital.com/SRC/esc/Esc.html`.

[6] David Evans. LCLint user's guide. URL: `http://www.sds.lcs.mit.edu/lclint/guide/`.

[7] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of POPL'98*, 1998.

[8] GrammaTech Inc. CodeSurfer user guide and reference manual. URL: `http://www.grammatech.com/papers/`.

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

[10] D. Jackson. Aspect: an economical bug-detector. In *Proceedings of 13th International Conference on Software Engineering*, 1994.

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

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

[13] Nils Klarlund and Anders Møller. *MONA Version 1.3 User Manual.* BRICS Notes Series NS-98-3 (2.revision), Department of Computer Science, University of Aarhus, October 1998. URL: `http://www.brics.dk/mona/manual.html`.

[14] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc. 20th Symp. on Princ. of Prog. Lang.*, pages 196–205. ACM, 1993.

[15] Nils Klarlund and Michael I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In *Proc. CAAP' 94, LNCS 787*, 1994.

[16] Nils Klarlund and Michael I. Schwartzbach. A domain-specific language for regular sets of strings and trees. *IEEE Transactions on Software Engineering*, 25(3), 1997.

[17] Adam Kolawa and Arthur Hicken. Insure++: A tool to support total quality software. URL: `http://www.parasoft.com/products/insure/papers/tech.htm`.

[18] Anders Møller. MONA project homepage. URL: `http://www.brics.dk/mona/`.

[19] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[20] 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.

[21] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.