Putting Static Analysis to Work for Verification

A Case Study

Tomasz Dudziak

Based on a paper by T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm

23 June 2010
A very capable **shape analysis** that can describe and discover complex properties of program heap structure, such as:

- An output of a program is a sorted list.
- One list is a permutation of the other.

- We can verify partial correctness of simple implementations of: bubble sort, insertion sort, merge, and destructive list reversal.
- For incorrect programs the analysis gives meaningful output and eases debugging.
- Analysis is based on the parametric framework for shape analysis using 3-valued logic (last meeting).
A very capable shape analysis that can describe and discover complex properties of program heap structure, such as:

- An output of a program is a sorted list.
- One list is a permutation of the other.

We can verify partial correctness of simple implementations of: bubble sort, insertion sort, merge, and destructive list reversal.

For incorrect programs the analysis gives meaningful output and eases debugging.

Analysis is based on the parametric framework for shape analysis using 3-valued logic (last meeting).
A very capable **shape analysis** that can describe and discover complex properties of program heap structure, such as:

- An output of a program is a sorted list.
- One list is a permutation of the other.

We can verify partial correctness of simple implementations of: bubble sort, insertion sort, merge, and destructive list reversal.

For incorrect programs the analysis gives meaningful output and eases debugging.

Analysis is based on the parametric framework for shape analysis using 3-valued logic (last meeting).
A very capable **shape analysis** that can describe and discover complex properties of program heap structure, such as:

- An output of a program is a sorted list.
- One list is a permutation of the other.

- We can verify partial correctness of simple implementations of: bubble sort, insertion sort, merge, and destructive list reversal.

- For incorrect programs the analysis gives meaningful output and eases debugging.

- Analysis is based on the parametric framework for shape analysis using 3-valued logic (last meeting).
1. How to approach verification of program correctness using shape analysis?
3. Analysis of singly-linked lists.
4. Extending the analysis to express properties related to ordering.
5. Actual verification.
A program is **partially correct** iff its output is correct every time it terminates.

- A program is **totally correct** iff it is partially correct and terminates for every input.
- In this approach we tackle only partial correctness.
- Total correctness is typically harder to prove (although both are undecidable in general case).
In traditional verification we need to supply a specification and lots of program-dependent information (loop invariants).

In this approach specification consists of:

1. description of input 3-VLS
2. acceptability criterion on output 3-VLS

The analysis (predicates and actions) is program-independent.
Recap: Core ideas of the framework

1. By encoding stores as logical structures questions about properties of stores can be answered by evaluating formulæ. (*Property-Extraction Principle*)

2. We can express the effect of statements on program store using predicate-update formulæ.

3. By explicitly storing values of interesting formulæ with the structure we can achieve greater precision. (*Instrumentation Principle*)

4. Same predicate-update formulæ that give concrete semantics of a program when used with 2-VLS give us transfer functions for abstract semantics when used with 3-VLS. (*Reinterpretation Principle*)
1. By encoding stores as logical structures questions about properties of stores can be answered by evaluating formulæ. (*Property-Extraction Principle*)

2. We can express the effect of statements on program store using predicate-update formulæ.

3. By explicitly storing values of interesting formulæ with the structure we can achieve greater precision. (*Instrumentation Principle*)

4. Same predicate-update formulæ that give concrete semantics of a program when used with 2-VLS give us transfer functions for abstract semantics when used with 3-VLS. (*Reinterpretation Principle*)
Recap: Core ideas of the framework

1. By encoding stores as logical structures questions about properties of stores can be answered by evaluating formulæ. *(Property-Extraction Principle)*

2. We can express the effect of statements on program store using predicate-update formulæ.

3. By explicitly storing values of interesting formulæ with the structure we can achieve greater precision. *(Instrumentation Principle)*

4. Same predicate-update formulæ that give concrete semantics of a program when used with 2-VLS give us transfer functions for abstract semantics when used with 3-VLS. *(Reinterpretation Principle)*
Recap: Core ideas of the framework

1. By encoding stores as logical structures, questions about properties of stores can be answered by evaluating formulæ. (*Property-Extraction Principle*)

2. We can express the effect of statements on program store using predicate-update formulæ.

3. By explicitly storing values of interesting formulæ with the structure, we can achieve greater precision. (*Instrumentation Principle*)

4. Same predicate-update formulæ that give concrete semantics of a program when used with 2-VLS give us transfer functions for abstract semantics when used with 3-VLS. (*Reinterpretation Principle*)
Recap: Theoretical landscape

- $f$ is a naïve conversion
- $blur$ merges nodes with same values on abstraction predicates.
- $\text{rng}(blur)$ is a tractable sublattice of $\mathcal{P}(3\text{-VLS})$. 

\[ \mathcal{P}(2\text{-VLS}) \quad \text{and} \quad \mathcal{P}(3\text{-VLS}) \]
Recap: Performing analyses

TVLA = Three-Valued-Logic Analyzer, implementation of the general framework

TVLA

possible states for each program point (as 3-VLS)

program

list of core predicates

definitions of instrumentation predicates

program Control Flow Graph

semantic of CFG actions (predicate-update formulae)

possible program inputs (as 3-VLS)

T. Dudziak

Putting Static Analysis to Work for Verification
We already have an analysis that can:

- Analyze programs manipulating singly-linked lists.
- Compute reachability, cyclicity, and sharedness information.
- Detect memory leaks and NULL-pointer dereferences.

We shall extend it to express ordering information.
Predicates

Core predicates

- \( x(v) = "v \) is pointed to by variable \( x" \)
- \( n(v_1, v_2) = "v_1 \cdot n \) points to \( v_2" \)

Instrumentation predicates

- \( r[n, x] = \exists v_1. x(v_1) \land n^*(v_1, v) \) (reachability)
- \( c[n](v) = n^+(v, v) \) (cyclicity)
- \( is[n](v) = \exists v_1 v_2. n(v_1, v) \land n(v_2, v) \land v_1 \neq v_2 \) (sharing)
Different TVLA actions represent different kinds of statements.

Every CFG edge has an associated action.

Actions are conceptually equivalent to transfer functions.
### Defining actions

<table>
<thead>
<tr>
<th>Part of action specification</th>
<th>TVLA syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>predicate-update formulæ</td>
<td>{ \ldots }</td>
</tr>
<tr>
<td>focus formulæ</td>
<td>%f{ \ldots }</td>
</tr>
<tr>
<td>precondition</td>
<td>%p formula</td>
</tr>
<tr>
<td>do we need to create nodes</td>
<td>%new</td>
</tr>
<tr>
<td>which nodes should we remove</td>
<td>%retain</td>
</tr>
<tr>
<td>textual representation</td>
<td>%t string</td>
</tr>
<tr>
<td>message specification</td>
<td>%message formula \rightarrow \text{string}</td>
</tr>
</tbody>
</table>
Corresponds to a statement: \( \text{lhs} = \text{rhs} \rightarrow \text{n} \).

\[
\begin{align*}
%\text{action} & \quad \text{Get}_-\text{Next}_-\text{L}(\text{lhs}, \text{rhs}) \{} \\
%\text{f} & \quad \{ \ E(\text{v}_1, \text{v}_2) \ \text{rhs}(\text{v}_1) \ & \ \& \ n(\text{v}_1, \text{v}_2) \ & \ \& \ t[n](\text{v}_2, \text{v}) \ \} \\
%\text{message} & \quad (\neg E(\text{v}) \ \text{rhs}(\text{v})) \ \rightarrow \\
& \quad \ " \text{Illegal dereference to} \ n" \ + \ n \ + \ " \text{component of} \ + \ \text{rhs} \\
& \quad \{ \ \text{lhs}(\text{v}) = E(\text{v}_1) \ \text{rhs}(\text{v}_1) \ & \ \& \ n(\text{v}_1, \text{v}) \ \} \\
\}
\end{align*}
\]

- Default predicate-update formula is identity.
- There is one additional "technical" instrumentation predicate \( t[n](\text{v}_1, \text{v}_2) = n^*(\text{v}_1, \text{v}_2) \).
### Other actions

<table>
<thead>
<tr>
<th>C expression</th>
<th>corresponding action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lhs = NULL</code></td>
<td><code>Set_Null_L(lhs)</code></td>
</tr>
<tr>
<td></td>
<td><code>Copy_Var_L(lhs, rhs)</code></td>
</tr>
<tr>
<td><code>lhs = rhs</code></td>
<td></td>
</tr>
<tr>
<td><code>lhs-&gt;n = NULL</code></td>
<td><code>Set_Next_Null(lhs)</code></td>
</tr>
<tr>
<td><code>lhs-&gt;n = rhs</code></td>
<td><code>Set_Next_L(lhs, rhs)</code></td>
</tr>
<tr>
<td><code>lhs != NULL</code></td>
<td><code>Is_Not_Null_Var(lhs)</code></td>
</tr>
<tr>
<td><code>lhs == NULL</code></td>
<td><code>Is_Null_Var(lhs)</code></td>
</tr>
<tr>
<td><code>lhs == rhs</code></td>
<td><code>Is_Eq_Var(lhs, rhs)</code></td>
</tr>
<tr>
<td><code>lhs != rhs</code></td>
<td><code>Is_Not_Eq_Var(lhs, rhs)</code></td>
</tr>
<tr>
<td><code>lhs = (L) malloc(...)</code></td>
<td><code>Malloc_L(lhs)</code></td>
</tr>
<tr>
<td><code>free(lhs)</code></td>
<td><code>Free_L(lhs)</code></td>
</tr>
</tbody>
</table>
ADT of singly-linked lists

```c
typedef struct node {
    int d;
    struct node *n;
} *L;
```

- We aim to analyze simple singly-linked lists with a single data field.
- The analysis does not depend on \( d \) being an integer; it generalizes to any comparable type.
The Analysis

- We start by taking the analysis of singly linked lists given with our framework.
- We extend this analysis with additional predicates to obtain a more expressive language for describing program states.
- We adjust existing actions to take new predicates into account.
- We create new actions to express more complex programs.
- We encode specified behavior of the program to the analysis using additional actions and predicates.
We start by taking the analysis of singly linked lists given with our framework.

We extend this analysis with additional predicates to obtain a more expressive language for describing program states.

We adjust existing actions to take new predicates into account.

We create new actions to express more complex programs.

We encode specified behavior of the program to the analysis using additional actions and predicates.
We start by taking the analysis of singly linked lists given with our framework.

We extend this analysis with additional predicates to obtain a more expressive language for describing program states.

We adjust existing actions to take new predicates into account.

We create new actions to express more complex programs.

We encode specified behavior of the program to the analysis using additional actions and predicates.
We start by taking the analysis of singly linked lists given with our framework.

We extend this analysis with additional predicates to obtain a more expressive language for describing program states.

We adjust existing actions to take new predicates into account.

We create new actions to express more complex programs.

We encode specified behavior of the program to the analysis using additional actions and predicates.
The Analysis

- We start by taking the analysis of singly linked lists given with our framework.
- We extend this analysis with additional predicates to obtain a more expressive language for describing program states.
- We adjust existing actions to take new predicates into account.
- We create new actions to express more complex programs.
- We encode specified behavior of the program to the analysis using additional actions and predicates.
Core Predicates

Scheme

- \( x(v) = "v\) is pointed to by variable \( x" \)
- \( n(v_1, v_2) = \"v_1.n\) points to \( v_2\)"
- \( dle(v_1, v_2) = \"v_1.d \leq v_2.d\"

- One \( x\)-predicate for every program variable.
- \( n\) and \( dle\) are ADT-dependent.
- For our ADT: only one \( n\)-predicate and \( dle\)-predicate.
- Analysis exploits the fact that \( dle\) is a total order.
Instrumentation Predicates

Borrowed from previous analysis

- \( r[n, x] = \exists v_1. \ x(v_1) \land n^*(v_1, v) \) (reachability)
- \( c[n](v) = n^+(v, v) \) (cyclicity)
- \( is[n](v) = \exists v_1 v_2. \ n(v_1, v) \land n(v_2, v) \land v_1 \neq v_2 \) (sharing)

New, data-related

- \( inOrder[dle, n](v) = \forall v_1. \ n(v, v_1) \Rightarrow dle(v, v_1) \)
- \( inROrder[dle, n](v) = \forall v_1. \ n(v, v_1) \Rightarrow dle(v_1, v) \)

Abstraction predicates: \( r[n, x] \), \( c[n] \), and \( is[n] \).
Predicates define language in which we describe program states – we need it to be precise enough.

For our task, we need to describe at least states like "x points to a sorted list".

Some intermediate states in sorting algorithms may require more complex descriptions.
Example: Expressiveness of abstract domain

Single variable $x$ points to a list $[1, 4, 3, 2]$.

\[ \text{inOrder : } u_1, u_2, \text{ inROrder : } u_4, u_3, u_2 \]

What is the corresponding abstract state?

- Abstraction predicates: $x, r[n, x], c[n], is[n]$.
- Nodes modulo equivalence on abstraction predicates: $\{ \{ u_1 \}, \{ u_4, u_3, u_2 \} \}$. 
Example: Expressiveness of abstract domain

Single variable $x$ points to a list $[1, 4, 3, 2]$.

\[
\text{inOrder} : u_1, u_2, \text{inROrder} : u_4, u_3, u_2
\]

What is the corresponding abstract state?

- Abstraction predicates: $x, r[n, x], c[n], is[n]$.
- Nodes modulo equivalence on abstraction predicates: $\{\{u_1\}, \{u_4, u_3, u_2\}\}$. 
Example: Expressiveness of abstract domain

\[ \begin{array}{c|cc}
\text{inOrder} & u_1 & u \\
\text{inROrder} & 1 & 1/2 \\
\text{r}[n,x] & 0 & 1 \\
is[n] & 0 & 0 \\
c[n] & 0 & 0 \\
\end{array} \]

Note that:

- First element is the minimum of the list.
- Remaining elements are sorted in descending order.
- Taking \textit{inOrder} and \textit{inROrder} as abstraction predicates would split arbitrary unsorted lists into sequences of sorted sublists.
Updating language semantic

- Predicate-update formulæ for existing predicates can be borrowed from previous analyses.
- We need new actions to represent conditions comparing data fields.
- We will need to represent boolean variables and define actions for them.
- We introduce several verification-related pseudo-actions.
void sort2 (L x) {
    L y, t;
    y = x->n;
    if (y.d <= x.d) {
        t = y->n;
        x->n = NULL;
        x->n = t;
        y->n = NULL;
        y->n = x;
        x = y;
    }
}
%action Less_Equal_Data_L(lhs, rhs) {
  %f { lhs(v_1) & rhs(v_2) & dle(v_1, v_2) }
  %p E(v_1, v_2) lhs(v_1) & rhs(v_2) & dle(v_1, v_2)
}

- Corresponds to condition \( \text{lhs.d} \leq \text{rhs.d} \).
- No predicate-update formulæ (identity on all predicates).
- Single precondition: \( \exists v_1, v_2. \text{lhs}(v_1) \land \text{rhs}(v_2) \land \text{dle}(v_1, v_2) \).
- Focus formula guarantees that after focusing, the ordering between nodes is always defined.
We would like to verify that first two elements of the list returned by `sort2` are always ordered.

TVLA’s messages mechanism can be used to define such actions.
Example: sort2.c – Demo

Input passed to TVLA:

1. sort2.tvp: contains predicate definitions, actions, and CFG generated from sort2.c

2. sort2.in.tvp: describes possible inputs as 3-VLS (arbitrary lists of length 2 and more)

TVLA produces:

1. sort2.out.tvp: all possible outputs as 3-VLS

2. sort2.out.ps: visualization of program CFG and 3-VLS at points of interest

To retrieve the result of verification we examine messages attached to output 3-VLS.
Verifying insertion-sort

- List pointed by $x$ is sorted iff:

$$\forall v. \ r[n, x](v) \Rightarrow inOrder[dle, n](v)$$

- Ordering alone is not enough – we need to verify that the list pointed by $x$ at the end is a permutation of the list at the beginning.

- We need "historical" reachability information:

$$\forall v. \ r[n, x](v) \iff \text{"v was reachable from x at L0"}$$
A special "origin" predicate stores historical reachability information:

\[ or[n, x, l](v) = "v was reachable from x at program point l" \]

Storing this information for every program point and variable would be very redundant.

We define action \texttt{Copy\_Reach\_L(lhs, l)} to store state of the part of the current heap. It has a single predicate-update formula:

\[ or[n, lhs, l](v) = r[lhs, x](v) \]
%action Copy_Reach_L(lhs, l) {
  \{ \text{or}[n, lhs, l](v) = r[n, lhs](v) \}
}

%action Assert_Permutation_L(lhs, label) {
  \%message !(A(v) (r[n,lhs](v) \leftrightarrow \text{or}[n,lhs,label](v)))) ->
  "Unable to prove that the list pointed to by " + lhs + 
  " is a permutation of the original list at program label " + label
}

%action Assert_Sorted_L(lhs) {
  \%message !(A(v) (r[n,lhs](v) \rightarrow \text{inOrder}[dle,n](v)))) ->
  "Unable to prove that the list pointed to by " + lhs + " is sorted"
}
Inputs (distributed with TVLA in example/sll_sorting):

1. `insertSort.tvp`: CFG and program-dependent information (list of variables, etc.)
2. `predicates.tvp`: Predicate definitions and integrity constraints; included by `insertSort.tvp`
3. `actions.tvp`: Operational semantic of CFG nodes; included by `insertSort.tvp`
4. `unsorted.tvs`: 3-VLS describing arbitrary unsorted lists
File insertSort_bug2.tvp replicates the analysis for a different implementation of insertion-sort.

This implementation contains a subtle bug that cause it to ignore the first element.

After running the analysis:

- We know that we failed to prove that the program is correct.
- We can examine produced 3-VLS descriptions to check what possible outputs can occur.
A different implementation of insertion-sort

- File insertSort_bug2.tvp replicates the analysis for a different implementation of insertion-sort.

- This implementation contains a subtle bug that causes it to ignore the first element.

After running the analysis:

- We know that we failed to prove that the program is correct.

- We can examine produced 3-VLS descriptions to check what possible outputs can occur.
Buggy insertion-sort implementation: output

- Analysis failed to verify that first two elements are properly ordered.
- The rest of the list is always sorted.
Verifying bubble-sort

```c
L bubbleSort(L x) {
    if (x == NULL) return;
    change = TRUE;
    while (change) {
        ...
        change = FALSE;
        ...
        while (yn != NULL) {
            ...
            change = TRUE;
            ...
        }
    }
    return x;
}
```
Modeling boolean variables

Solution

Every boolean variable in the program can be modelled as a nullary core predicate. For our bubble-sort program, we define additional predicate change().

- In every concrete state (a 2-VLS) change() must have either value 0 or 1.
- Abstract states can also model situations where the value of a variable is unknown.
- We define actions for manipulating boolean variables: Set_True, Set_False Is_True, Is_False.
Operations on boolean variables

%action Set_True(lhs) {
  { lhs() = 1 }
}

%action Set_False(lhs) {
  { lhs() = 0 }
}

%action Is_True(lhs) {
  %p lhs()
}

%action Is_False(lhs) {
  %p !lhs()
}
Inputs (distributed with TVLA in example/sll_sorting):

1. bubbleSort.tvp: CFG and program-dependent information (list of variables, etc.)

2. predicates.tvp: Predicate definitions and integrity constraints; included by insertSort.tvp

3. actions.tvp: Operational semantic of CFG nodes; included by insertSort.tvp

4. unsorted.tvs: 3-VLS describing arbitrary unsorted lists

Only bubbleSort.tvp is different than in previous example.
Condition for swapping elements is changed from $y->data > yn->data$ to $y->data \geq yn->data$.

The procedure does not terminate if the input contains two elements with equal $d$-fields.

But this does not break partial correctness!

After running the analysis:

- We do not get any messages: partial correctness has been proven.
- But by examining output structures we can figure out that something is wrong.
Condition for swapping elements is changed from \( y \rightarrow \text{data} > y_n \rightarrow \text{data} \) to \( y \rightarrow \text{data} \geq y_n \rightarrow \text{data} \).

The procedure does not terminate if the input contains two elements with equal \( d \)-fields.

But this does not break partial correctness!

After running the analysis:

- We do not get any messages: partial correctness has been proven.
- But by examining output structures we can figure out that something is wrong.
Only nontrivial structure we get is:

What is wrong here?
For every input list we obtain output that:

- is a permutation of the input
- is sorted
- first element is the minimum of the list?
DataIsNEqual[dle, n](v) = \forall v_1. n(v, v_1) \Rightarrow \neg (dle(v, v_1) \land dle(v_1, v))

- We can try to examine the rest of the list.
- Re-running the analysis with additional instrumentation predicate proves that no two elements in the output are equal!
Summary of the analysis

We have extended the singly-linked lists analysis with following:

1. Ordering-aware predicates: \( dle, \text{inOrder}, \text{inROrder} \).

2. New conditions: Less\_Equal\_Data\_L, Greater\_Equal\_Data\_L, ... 

3. Actions for manipulating boolean variables (modelled as nullary predicates): Set\_True, Is\_True, ... 

4. Origin predicate \( \text{or}[n, x, l] \) and action Copy\_Reach\_L that implement heap snapshots.

5. Actions that implement assertions: Assert\_Sorted\_L, Assert\_Permutation\_L, ...
We are able to verify that merging two sorted lists results in a sorted list.

For merge, heap snapshots are not used. Instead, we verify that no elements are lost by using `Assert_No_Leak`:

```plaintext
%action Assert_No_Leak(lhs) {
  %f { lhs(v) }
  %p E(v) !r[n,lhs](v)
  %message ( E(v) !r[n,lhs](v) ) ->
          "There may be a list element not " +
          "reachable from variable " + lhs + ")" 
}
```
We are able to verify that reversing a sorted list results in a reversely sorted list.

In the file reverse.tvp distributed with TVLA the problem of losing elements is not addressed.

But I was able to establish that the output is a permutation of the input using heap snapshots.
Replicating the results

- TVLA is available at http://www.math.tau.ac.il/~tvla/.
- Directory examples/sll_sorting contains TVLA programs from the paper.
- Analyzing all the programs takes about 30 seconds.