natural proofs for verification of dynamic heaps

P. MADHUSUDAN

UNIV. OF ILLINOIS AT URBANA-CHAMPAIGN

(VISITING MSR, BANGALORE)

WORKSHOP ON MAKING FORMAL VERIFICATION SCALABLE AND USEABLE
CMI, CHENNAI, INDIA
JOINT WORK WITH...

Xiaokang Qiu
.. graduating!

Gennaro Parlato
Andrei Stefanescu
Pranav Garg
GOAL: BUILD RELIABLE SOFTWARE

Build systems with proven reliability and security guarantees.

(Not the same as finding bugs!)

Deductive verification with automatic theorem proving

• Floyd/Hoare style verification
• User supplied modular annotations (pre/post cond, class invariants) and loop invariants
• Resulting verification conditions are derived automatically for each linear program segment (using weakest-pre/strongest-post)

Verification conditions are then proved valid using mostly automatic theorem proving (like SMT solvers)
TARGET: RELIABLE SYSTEMS SOFTWARE

Several success stories:

• Microsoft Hypervisor verification using VCC  [MSR, EMIC]

• A secure foundation for mobile apps  [@Illinois, ASPLOS’13]
A SECURE FOUNDATION FOR MOBILE APPS

In collaboration with King’s systems group at Illinois

First OS architecture that provides verifiable, high-level abstractions for building mobile applications.

Application state is private by default.

- Meta-data on files controls access permissions can access files
- Every page is encrypted before sending to the storage service
- Memory isolation between apps
- A page fault handler serves a file-backed page for a process, the file has to be opened by the same process.
- Only the current app can write to the screen buffer.
- IPC channel correctness
- ...

And works!

VERIFICATION TOOLS ARE MATURE ENOUGH TO BUILD RELIABLE S/W.
KEY CHALLENGE: HEAPS

- **Methodology:**
  - User provides specification using modular annotations (pre/post conditions, class inv)
  - User also provides loop invariants.
  - Automatic generation of verification conditions (pure logic formulas whose validity needs to be verified)

  **Example:** 
  $[[x > i]]$ \(x := x + 2;\) \(i := i + 1;\) $[[x > i]]$

  gives verification condition:

  \[
  \forall x, x', i, i' \ (x > i \land x' = x + 2 \land i' = i + 1) \Rightarrow i' > x'
  \]

- Validity of verification conditions done mostly automatically

- Works well when program uses only static variables (like above)
  *But doesn’t really work when manipulating objects, dynamic heaps, etc.*

- Key challenges: specification language, validity checking
Node avl_insert(Node t, Int v)

//requires “t points to an AVL tree”

//ensures “returns a tree t’, where t’ is a pointer to an AVL tree, keys stored in t’ = keys stored in t U { v } and height increases at most by 1”

//where t points to an AVL-tree if t points to a tree wrt some pointer fields left, right, and tree is almost balanced, and is a binary search tree...

{ ...

  <<code for AVL-insert>>

}
DYNAMIC HEAPS

Fix a finite set of pointer fields PF and a set of data fields DF.
Fix also a set of program variables PV.

A heap is a finite set of locations L and maps

\[ m_p: L \rightarrow L \cup \{nil\} \quad \text{for each } p \in PF \]
\[ m_d: L \rightarrow \mathbb{Z} \quad \text{for each } d \in DF \]
\[ pv: PV \rightarrow L \cup \{nil\} \]

Graph: edge-labels in PF, integers on nodes, PV-labels on some nodes

Example: binary trees

\[ PF = \{l, r\}; \quad DF = \{\text{key}\}; \quad PV = \{x\} \]
(all missing arrows go to nil)
DYNAMIC HEAPS: INHERENT UNDECIDABILITY

Unbdd # nodes => need universal quantification to describe properties

**Example:** Sortedness of lists: $\forall x, y. \ (\text{succ}(x, y) \rightarrow (x.d \leq y.d))$

But this immediately gives undecidability of satisfiability/validity:

We can simulate 2-counter machines using a linked list with two data-fields. Assert consecutive nodes in the list encode the right evolution of counters

$\forall x, y. \ (\text{succ}(x, y) \rightarrow \varphi(x.c1, x.c2, y.c1, y.c2))$

![Diagram of linked list simulating 2-counter machines]
FUNCTIONAL VERIFICATION OF HEAP-MANIPULATING PROGRAMS

Expressive
Expressive Logics: Separation logics, FOL+Ghost-defns, HOL, MatchingLogic, etc.

Decidable Logics: LISBQ, CSL, STRAND, etc.

Natural Proofs
Useful for proving programs correct

Keep expressiveness
Give up decidability
Sound but incomplete
Preserve automaticity

Expressive
Automatic

Natural Proofs

Expressive
Decidable

Natural Proofs

Expressive
Decidable

Natural Proofs

Expressive
Decidable

Natural Proofs
DECIDABLE LOGICS VS NATURAL PROOFS

Formulas

Decision procedure for searching for a proof in $\mathcal{N}$
NATURAL PROOFS

- Handle a logic that is very **expressive** (inevitably undecidable)
- **Identify a class of simple and natural proofs** $\mathcal{N}$ such that
  - Many correct programs can be proved using a proof in class $\mathcal{N}$
  - The class $\mathcal{N}$ is effectively searchable (searching thoroughly for a proof in $\mathcal{N}$ is efficiently decidable)

**Natural proofs**
- **unfold** recursive definitions across the **footprint** that a straight-line program manipulated, E.g.

$$
tree^*(x) \overset{\text{def}}{=} (x = \text{nil \ emp})
$$

$$
(x \mapsto xl, xr)^l,r \ast tree^*(xl) \ast tree^*(xr)
$$
- **formula abstraction** (make recursive definitions uninterpreted)
DRYAD LOGIC

Aim: To provide a single logical framework that supports natural proofs for general properties of structure, data, and separation

DRYAD: A dialect of separation logic

• no explicit quantification, but supports recursive definitions
• admits a “deterministic” quantifier-free translation to classical logic
• Develop natural proofs for this logic using decision procedures (powered by SMT solvers)

Separation logic [Reynolds, O’Hearn, Ishtiaq, Yang]

Key insight: formulas should be defined on local (small) heaplets by default, not the global heap

Separation operator: * to combine heaplets

E.g., $x \rightarrow y$ is true on a heaplet where the pointer fields from only the node $\{x\}$ is defined; not true on larger heaps!

$p * q$ is true on a heaplet $H$ if it can be split into two disjoint heaplets, one satisfying $p$ and one satisfying $q$
SEPARATION LOGIC

\[ x \xrightarrow{p} y \]

\[ x \xrightarrow{p} y \xrightarrow{\text{X}} z \]

\[ x \xrightarrow{p} y \]

\[ x \xrightarrow{p} y \]

\[ x \xrightarrow{p} y \]

\[ x \xrightarrow{p} y \]
SEPARATION LOGIC

\[ x \xrightarrow{P} y \land y \xrightarrow{P} z \]

\((x \xrightarrow{P} y) \land (y \xrightarrow{P} z)\)
SEPARATION LOGIC

def \( \text{tree}^* (x) = (x = \text{nil} \quad \text{emp}) \)

\[
\begin{align*}
(x \xleftarrow{l,r} xl, xr)^* & \quad \text{tree}^* (xl)^* \quad \text{tree}^* (xr)
\end{align*}
\]
BINARY SEARCH TREE:
AN EXAMPLE USING DRYAD

\[
\begin{align*}
\text{def} & \quad \text{bst}(x) = (x = \text{nil} \land \text{emp}) \lor \\
& \quad (x \mapsto x_l, x_r, x_k) \land (\text{bst}(x_l) \land \text{keys}(x_l) < \{x_k\}) \land (\text{bst}(x_r) \land \{x_k\} < \text{keys}(x_r)) \\
\text{def} & \quad \text{keys}(x) = (x = \text{nil}: \emptyset) \lor \\
& \quad (x \mapsto x_l, x_r, x_k) \land \text{true}: \{x_k\} \cup \text{keys}(x_l) \cup \text{keys}(x_r); \\
& \quad \text{default}: \emptyset \\
\psi & \equiv \text{bst}(\text{root}) \land (\text{bst}(\text{curr}) \land \text{true}) \\
& \land (k \in \text{keys}(\text{root}) \iff k \in \text{keys}(\text{curr}) \land \text{true})
\end{align*}
\]

(loop invariant for “bst-search”:

- \text{root} points to a BST and \text{curr} points into the BST;
- \text{k} is stored in the BST iff \text{k} is stored under \text{curr}.)
SYNTAX OF DRYAD

\[ pf \in PF \quad i^* : \text{Loc} \rightarrow \text{Int}_L \quad p^* : \text{Loc} \rightarrow \text{Bool} \quad x \in \text{Loc Variables} \]
\[ df \in DF \quad st^* : \text{Loc} \rightarrow S(\text{Int}) \quad S \in S(\text{Int}) \text{ Variables} \quad j \in \text{Int}_L \text{ Variables} \]
\[ c \in \text{Int}_L \text{ Constant} \quad mst^* : \text{Loc} \rightarrow MS(\text{Int})_L \quad MS \in MS(\text{Int})_L \text{ Variables} \quad q \in \text{Bool Variables} \]

Loc Term: \[ l_t ::= x | \text{nil} \]
Int\_L Term: \[ i_t ::= c | j | i_{<}^* (lt) | i + i | i - i \]

S(Loc) Term: \[ s_{lt} ::= \emptyset_L | L | \{lt\} | s_{<}^* (lt) | s_{\cup} s_{lt} | s_{\cap} s_{lt} | s_{\setminus} s_{lt} \]

S(Int) Term: \[ s_{it} ::= \emptyset_S | S | \{i\} | s_{<}^* (lt) | s_{\cup} s_{it} | s_{\cap} s_{it} | s_{\setminus} s_{it} \]

MS(Int)_L Term: \[ m_{sit} ::= \emptyset_M | M | \{i\}_m | m_{<}^* (lt) | m_{\cup} m_{sit} | m_{\cap} m_{sit} | m_{\setminus} m_{sit} \]

Positive Formula: \[ \phi ::= \text{true} | q (p_{<}^* (lt)) | \text{emp} | \overset{df}{\overset{pf}{\rightarrow}} \langle \tilde{lt}, \tilde{i} \rangle | \rightarrow lt = lt | lt \neq lt | it \leq it | it < it | \]
\[ s_{lt} \subseteq s_{lt} | s_{lt} \not\subseteq s_{lt} | s_{it} \subseteq s_{it} | s_{it} \not\subseteq s_{it} | m_{sit} \subseteq m_{sit} | m_{sit} \not\subseteq m_{sit} | \]
\[ lt \in s_{lt} | lt \not\in s_{lt} | it \in s_{it} | it \not\in s_{it} | it \in m_{sit} | it \not\in m_{sit} | s_{it} \leq s_{it} | s_{it} < s_{it} | m_{sit} \leq m_{sit} | m_{sit} < m_{sit} | \]
\[ \phi \land \phi | \phi \lor \phi | \phi \neq \phi \]

Formula: \[ \psi ::= \phi | \psi \land \psi | \psi \lor \psi | \neg \psi \]
RECURSIVE DEFINITIONS

Recursive Functions

\[ \text{Loc} \rightarrow \text{Int}_L, \text{Loc} \rightarrow \text{S}(\text{Loc}), \text{Loc} \rightarrow \text{S}(\text{Int}), \text{Loc} \rightarrow \text{MS}(\text{Int})_L \]

\[
\begin{align*}
  f^*_{pf, \vec{t}}(x) & \overset{\text{def}}{=} (\varphi^f_1(x, \vec{r}, \vec{v}) : r^f_1(x, \vec{v}) ; \ldots ; \varphi^f_k(x, \vec{r}, \vec{v}) : r^f_k(x, \vec{v}) ; \text{default} : r^f_{k+1}(x, \vec{v})) \\
  p^*_{pf, \vec{t}}(x) & \overset{\text{def}}{=} \varphi^p(x, \vec{r}, \vec{v})
\end{align*}
\]

Recursive Predicates

Restrictions

- Subtraction, set-difference and negation are disallowed
- existential variables \( \vec{v} \) are bounded by \( x \)

They are defined over a fixed heaplet

the set of reachable locations using \( \overrightarrow{pf} \), but without going through \( \vec{t} \)

Example: list-segment from \( x \) to \( y \)

\[ \text{lseg}^*_{\text{next}, y}(x) \]
SEMANTICS: RECURSIVE DEFINITIONS

To evaluate a recursive definition $f_{pf,t}^*(lt)$ over a heaplet $h$:

- compute the reach set $\text{Reach}_{lt}$ with respect to $\overrightarrow{pf}$ and $\overrightarrow{t}$
- if $\text{Dom}(h) = \text{Reach}_{lt}$, then evaluate it as the least fixpoint of the recursive definitions
- otherwise, evaluate to “$\text{undef}$”

\[
U_{\{l,r\}}(x) = (x = \text{nil} \land \text{emp}) \lor \\
\left\lfloor l,r \right\rfloor (x \mapsto y, z \ast U_{\{l,r\}}(y)) \lor \\
\left\lfloor l,r \right\rfloor (x \mapsto y, z \ast U_{\{l,r\}}(z))
\]

Example

\[
U_{\{l,r\}}(x) \text{ is satisfied by the entire tree}
\]

DRYAD:

Conventional SL:
TRANSLATING DRYAD TO A CLASSICAL LOGIC

The scope (heaplet required) of a formula can be syntactically determined:

- singleton heap: a single location
- recursive definitions: the set of reachable locations according to certain pointer fields (but ending at certain prespecified nodes)
- \( t \text{ op } t' \text{ and } t \sim t' \): the union of the scopes of both sides

The domain of a heaplet can be modeled as a set of locations, and the heaplet semantics can be expressed using free set variables.

Example: \( \text{tree}^*(x) \times \text{tree}^*(y) \)

\[ \begin{align*}
\text{tree}(x) & \quad \text{tree}(y) \\
\text{reach}(x) & \quad \text{reach}(y) = G
\end{align*} \]
We consider programs with modular annotations

• pre/post, loop invariant in DRYAD

• `ret` denotes the returned value

We verify linear blocks of code, called basic blocks (loops/conditionals are replaced with `assume` statements)

Natural Proofs in 4 steps

1. Translate DRYAD to classical logic
   \((R_0, \ldots, R_n\) as the global heap at each timestamp)

2. VC-Generation (compute strongest-post)

3. Unfolding across the Footprint (still precise, explained later)

4. Formula Abstraction
   (recursive definitions uninterpreted, becomes sound but incomplete)
UNFOLDING ACROSS THE FOOTPRINT

The verification condition $\psi_{VC}$ involves recursive definitions that can be unfolded \textit{ad infinitum}.

Our Strategy

- Unfold only across the footprint (locations get dereferenced in the program). For each $u$ in the footprint, for each timestamp $i$, add

  \[ tree_i(u) \leftrightarrow \left( (u = \text{nil} \land \text{reach}_i(u) = \emptyset) \lor \right) \]

  \[ (tree_i(ul) \land tree_i(ur) \land \ldots) \]

- Make the recursive definitions consistent across the timestamp:

  \[ \psi'_{VC} \equiv \psi_{VC} \land \text{Unfold} \land \text{Footprint} \]
  \[ \land \text{SegUnchanged} \land \text{CallUnchanged} \land \text{SelfReach} \]

Theorem

- $\psi_{VC}$ is valid iff $\psi'_{VC}$ is valid.
FORMULA ABSTRACTION

Natural Proofs in two steps:

- unfold \( VC \)
- formula abstraction \( VC \)
- abs \( VC \)

Formula Abstraction

- replace each recursive definition \( \text{rec} \) with uninterpreted \( \text{rec} \)
- replace the corresponding reach set \( \text{Reach}^{\text{rec}} \) with uninterpreted \( H^{\text{rec}} \)
- when a proof for \( \text{abs}_{VC} \) is found, we call it a natural proof for \( VC \)
  (sound but incomplete)

\( \text{abs}_{VC} \) is mostly expressible in the QF theory of arrays, maps, uninterpreted functions and integers (sets/multisets as arrays, heap mutations as array-store operations, set-operations as mapping functions)

\[ S_1 \subseteq S_2 \text{ between integer sets?} \]
\[ \text{translated to } i_1, i_2, (i_1 < i_2 \rightarrow (S_2[i_1] \lor S_1[i_2])) \]
\[ \text{(decidable in Array Property Fragment)} \]
EXPERIMENTAL EVALUATION

A prototype verifier for Dryad with Z3 as the backend solver

Verified about 100 routines manipulating datastructures, automatically.

- 10+ data structures
  - singly-linked list, sorted list, doubly-linked list, cyclic list, max-heap,
    BST, Treap, AVL tree, red-black tree, binomial heap…
- 80+ DRYAD-annotated programs
  - textbook algorithms, GTK library, OpenBSD library, an ongoing
    OS+Browser verification project…
- All these VCs that were generated by the natural proof methodology set forth in this work were proved by Z3

(To the best of our knowledge)

First terminating automatic mechanism that can prove such a wide variety of data-structure algorithms full-functionally correct
<table>
<thead>
<tr>
<th>Datastructure</th>
<th>Routines</th>
<th>Time (s) / routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singly linked lists</td>
<td>find, insert_front, insert_back, delete_all, copy, append, reverse</td>
<td>1s</td>
</tr>
<tr>
<td>Sorted lists</td>
<td>find, insert, merge, delete_all, insert_sort, reverse, find_last, insert*, quicksort</td>
<td>9s</td>
</tr>
<tr>
<td>Doubly-linked lists</td>
<td>insert_front, insert_back, delete_all, append, mid_insert, mid_delete, meld</td>
<td>1s</td>
</tr>
<tr>
<td>Cyclic lists</td>
<td>Insert_front, insert_back, delete_front, delete_back</td>
<td>1s</td>
</tr>
<tr>
<td>Max-heap</td>
<td>heapify</td>
<td>9s</td>
</tr>
<tr>
<td>Binary search trees</td>
<td>find, find*, insert, delete, remove_root, find_leftmost, remove_root, delete</td>
<td>47s</td>
</tr>
<tr>
<td>Treap</td>
<td>find, delete, insert, remove_root</td>
<td>7s</td>
</tr>
<tr>
<td>AVL trees</td>
<td>balance, leftmost, insert, remove</td>
<td>5s</td>
</tr>
<tr>
<td>Red-black trees</td>
<td>Insert, insert_left, fix, insert_right_fix, delete, delete_left_fix, delete_right_fix, leftmost</td>
<td>16s</td>
</tr>
<tr>
<td>Binomial heap</td>
<td>find_min, merge</td>
<td>78s</td>
</tr>
<tr>
<td>Tree traversals</td>
<td>Inorder_tree_to_list, inorder_tree_to_list*, preorder, postorder, inorder</td>
<td>10s</td>
</tr>
<tr>
<td>Package</td>
<td>Routines</td>
<td>Time (s) / rout</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td><strong>schorr-waiter</strong></td>
<td>marking_iter</td>
<td>1s</td>
</tr>
<tr>
<td><strong>glib/gslist.c</strong></td>
<td>free, prepend, concat, insert_before, remove_all, remove_link, delete_link, copy, reverse, nth, nth_data, find, position, index, last, length</td>
<td>1s</td>
</tr>
<tr>
<td><strong>Singly linkedlist (1.1K)</strong></td>
<td>append, insert_at_pos, remove, insert_slist, merge_slists, merge_sort</td>
<td>7s</td>
</tr>
<tr>
<td><strong>glib/glist.c</strong></td>
<td>free, prepend, reverse, nth, nth_data, position, find, index, last, length</td>
<td>1s</td>
</tr>
<tr>
<td><strong>DoublyLinkedlist (0.3K)</strong></td>
<td>quicksort_iter</td>
<td>65s</td>
</tr>
<tr>
<td><strong>openbsd/queue.h</strong></td>
<td>simpleq_init, simpleq_remove, simpleq_insert_head, simpleq_insert_tail, simpleq_insert_after, simpleq_remove_head</td>
<td>6s</td>
</tr>
<tr>
<td><strong>LOC 0.1K</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>secureOS/cachepage</strong></td>
<td>Lookup_prev, add_cachepage</td>
<td>4s</td>
</tr>
<tr>
<td><strong>memRegion (0.1K)</strong></td>
<td>memory_region_init, create_user_space_reg, split_memory_region</td>
<td>4s</td>
</tr>
<tr>
<td><strong>linux/mmap.c (0.1K)</strong></td>
<td>find_vma, remove_vma, remove_vma_list</td>
<td>1s</td>
</tr>
</tbody>
</table>
CONCLUSIONS

• Deductive verification with automated theorem proving
  Tipping point; very effective

• Logics for heaps and automated reasoning for them form a fascinating landscape of research

• Despite being the core software verification problem, field is quite young.

• Natural proofs:
  • Sound-but-incomplete procedures
  • Search for natural proof done by SMT solvers
  • Very expressive logics
  • Embodies natural proof tactics
  • Tractable separation logic
DEDUCTIVE VERIFICATION WITH AUTOMATED THM PROVING

Building reliable software with proven properties
SecureOS [ASPLOS13] [Mai, King, Pek, Xue]

Natural proof for tree ds [POPL’12]
Natural proofs for sep logic [submitted]

Strengthen natural proofs; strengthen pre/post/LI automatically

Concurrency; rely-guarantee

Combining dec sep logic with rely/guar

Natural proofs for C
NP -> ghost code in VCC

Ongoing work: [Pek, Qiu]

Learning loop invariants

Ongoing work: [Viswanathan, Qiu]

Ongoing work: [Garg, Loding, Neider]
SCRATCH