natural proofs for verification of dynamic heaps

P. MADHUSUDAN

UNIV. OF ILLINOIS AT URBANA-CHAMPAIGN

(VISITING MSR, BANGALORE)

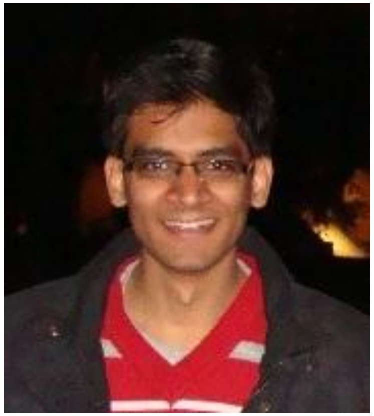
WORKSHOP ON MAKING FORMAL VERIFICATION SCALABLE AND USEABLE
CMI, CHENNAI, INDIA

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 SYS TEMS SOFTWARE

Operating Systems
Browsers
VMs
App platforms

Systems Software

Angry birds
Excel
Mail clients …

Applications

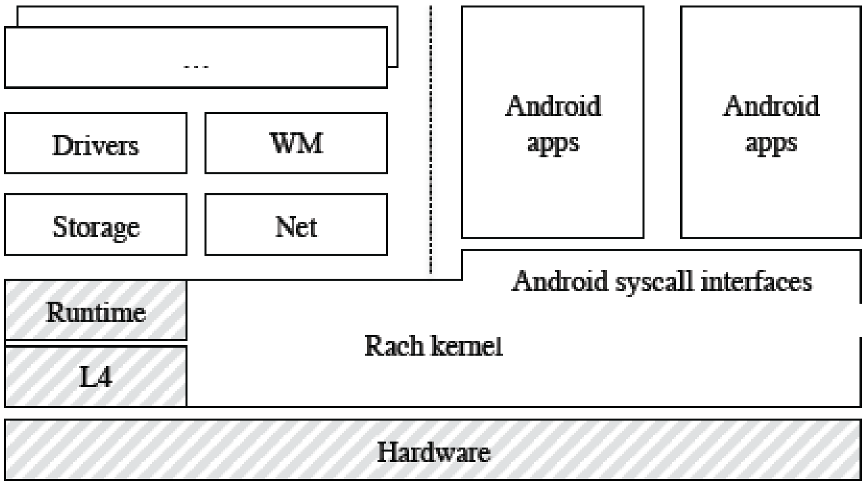
Several success stories:

- 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 se
- Memory isolation between apps
- A page fault handler serves a file- backed page for a process, the fille 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:

$$\forall x, x', i, i' \ \ (x > i \ \ \wedge x' = x + 2 \ \wedge i' = i + 1) \ \Rightarrow i' > x')$$

  - 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**

FULL FUNCTIONAL VERIFICATION
EX. AVL TREE FIND

```
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>>

 }
```

Key requirements:

- **A natural specification logic**
- **Automated reasoning**

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

$$\boldsymbol{m_p}: \boldsymbol{L} \rightarrow \boldsymbol{L} \cup \{\boldsymbol{nil}\} \qquad \text{for each } \boldsymbol{p} \in \boldsymbol{PF}$$

$$\boldsymbol{m_d}: \boldsymbol{L} \rightarrow \mathbb{Z} \qquad \text{for each } \boldsymbol{d} \in \boldsymbol{DF}$$

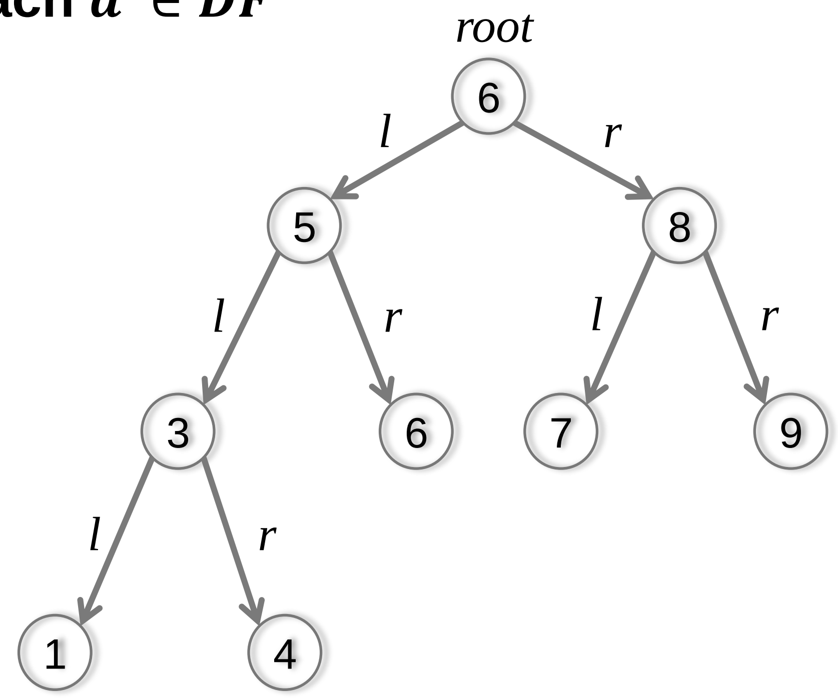
$$\boldsymbol{pv}: \boldsymbol{PV} \rightarrow \boldsymbol{L} \cup \{\boldsymbol{nil}\}$$

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

Example: binary trees

$$\boldsymbol{PF} = \{\, \boldsymbol{l},\ \boldsymbol{r} \,\};\ \ \boldsymbol{DF} = \{\, \boldsymbol{key} \,\};\ \boldsymbol{PV} = \{\boldsymbol{x}\}$$

(all missing arrows go to nil)



DYNAMIC HEAPS: INHERENT UNDECIDABILITY

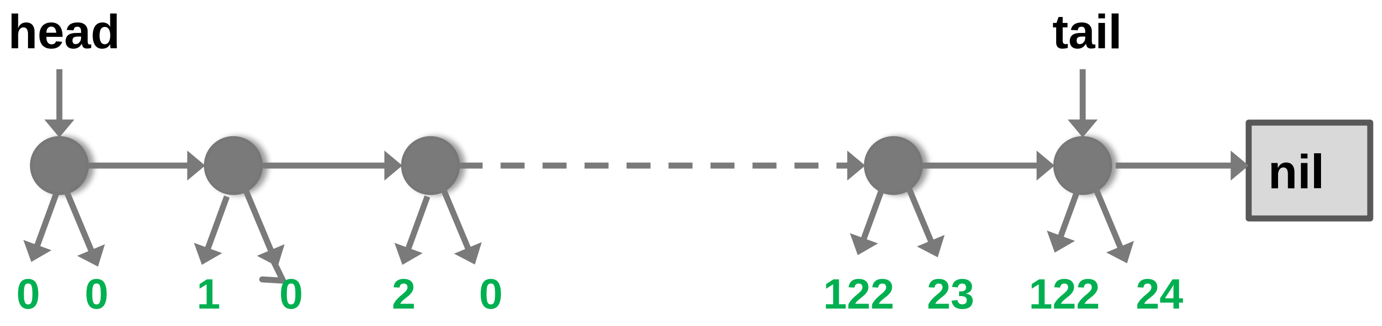
Unbdd # nodes => need universal quantification to describe properties

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

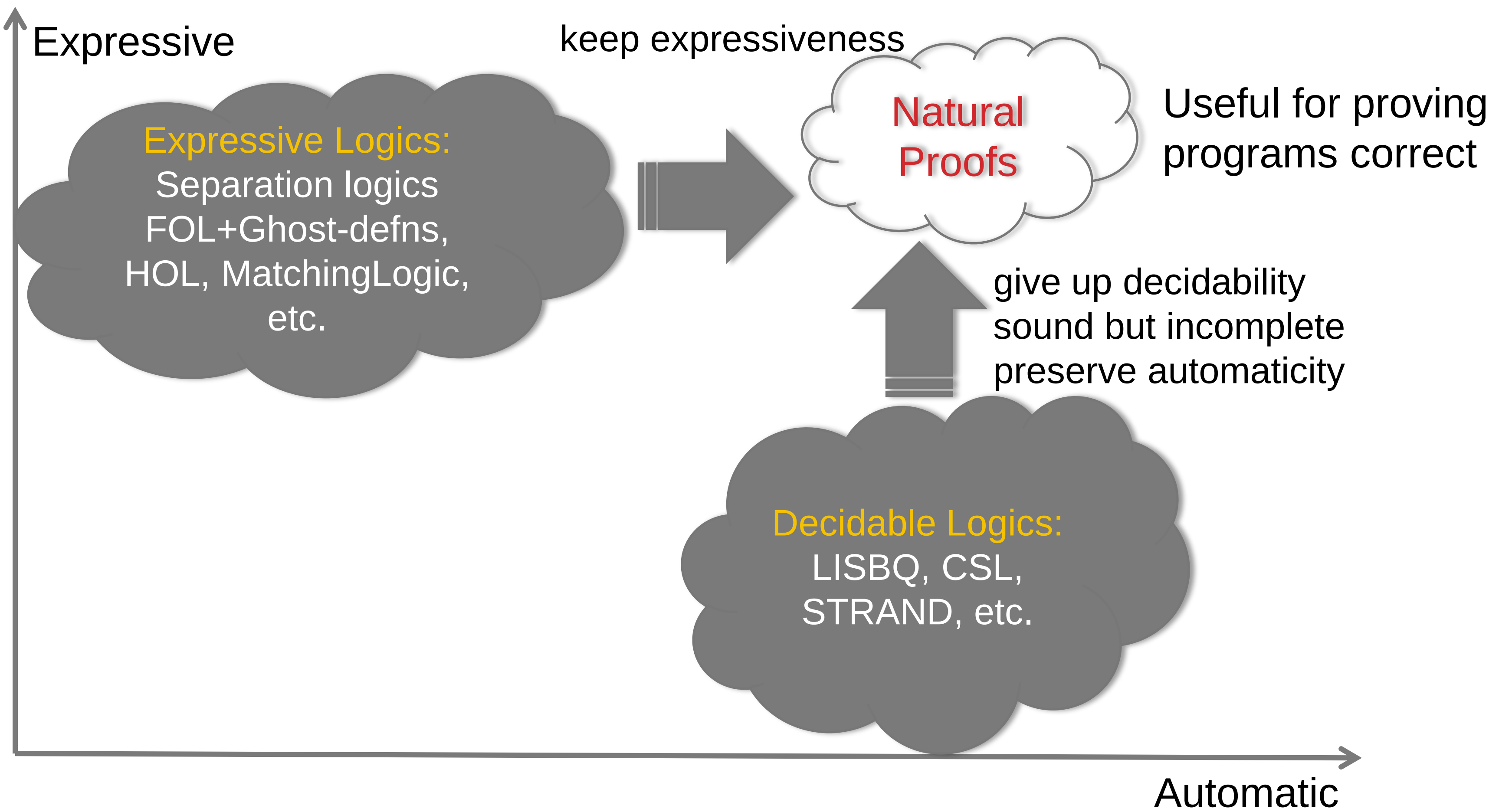
But this immediately gives undecidability of satisfiability/validity:

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

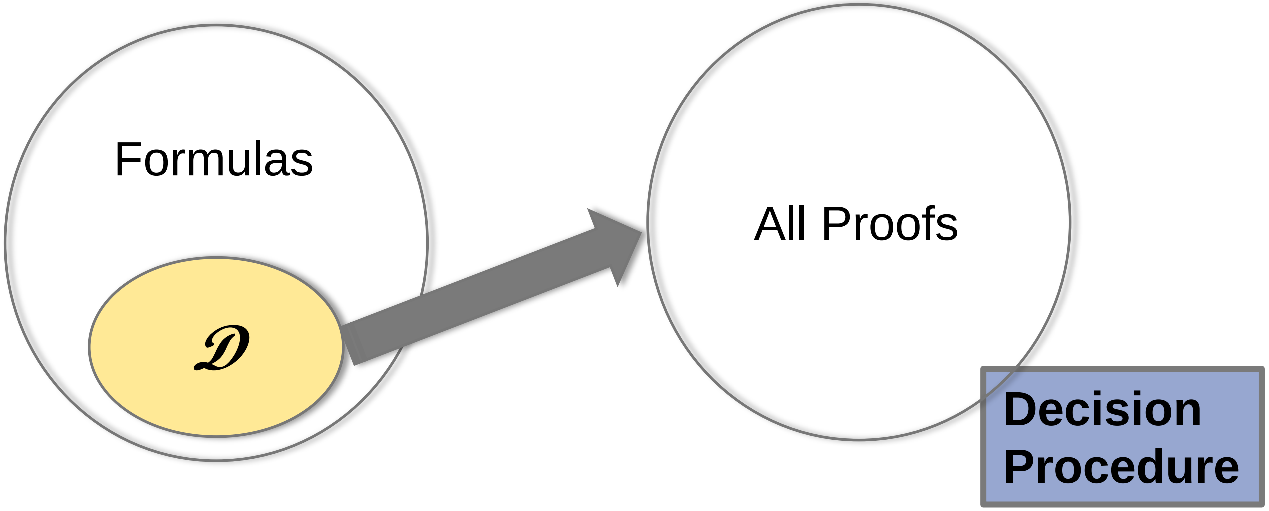
$$\forall\, \boldsymbol{x}, \boldsymbol{y}.\ (\ \boldsymbol{succ}(\boldsymbol{x}, \boldsymbol{y})\ \rightarrow\ \boldsymbol{\varphi}(\boldsymbol{x}.\boldsymbol{c1}, \boldsymbol{x}.\boldsymbol{c2}, \boldsymbol{y}.\boldsymbol{c1}, \boldsymbol{y}.\boldsymbol{c2})\ )$$

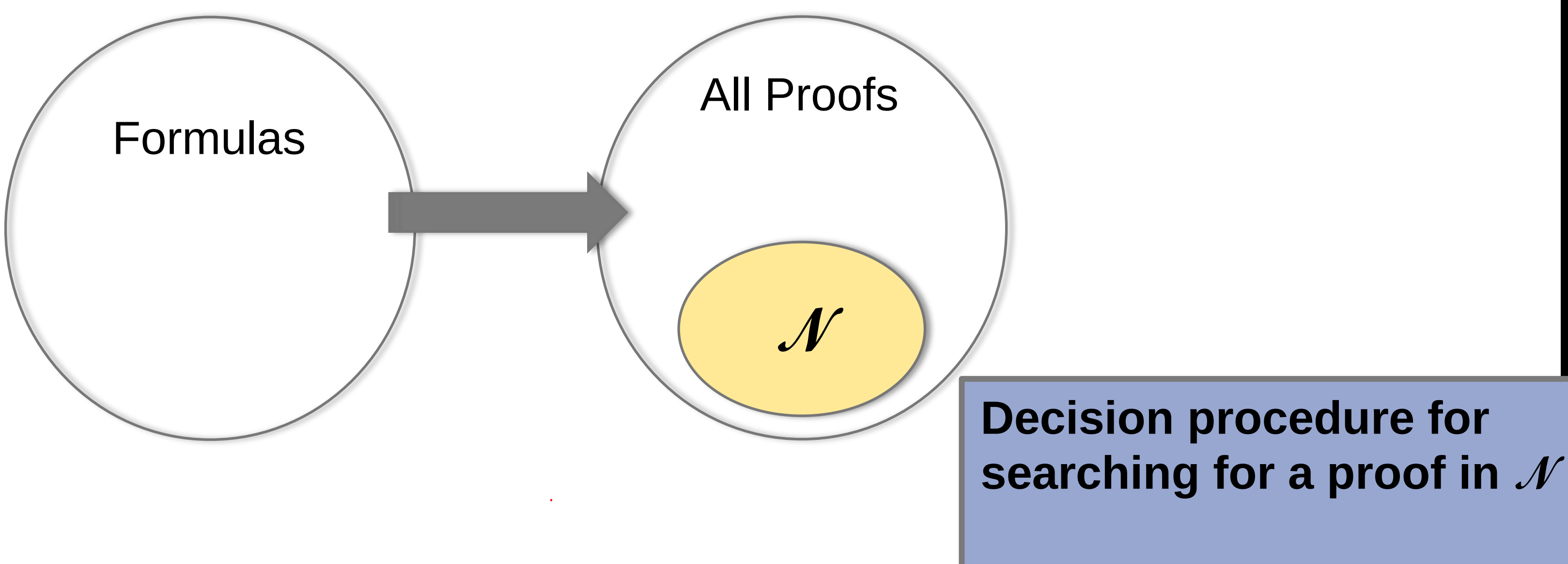


FUNCTIONAL VERIFICATION OF HEAP-MANIPULATING PROGRAMS



DECIDABLE LOGICS VS NATURAL PROOFS

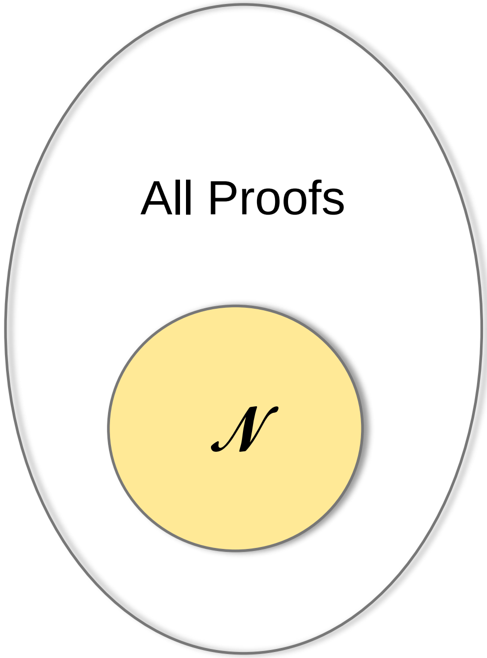




NATURAL PROOFS

- **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 N is efficiently decidable)



Natural proofs

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

$$tree^*(x) \overset{def}{=} (x = \mathrm{nil} \wedge \mathrm{emp}) \vee (x \overset{l,r}{\mapsto} xl, xr) * tree^*(xl) * tree^*(xr)$$

- 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

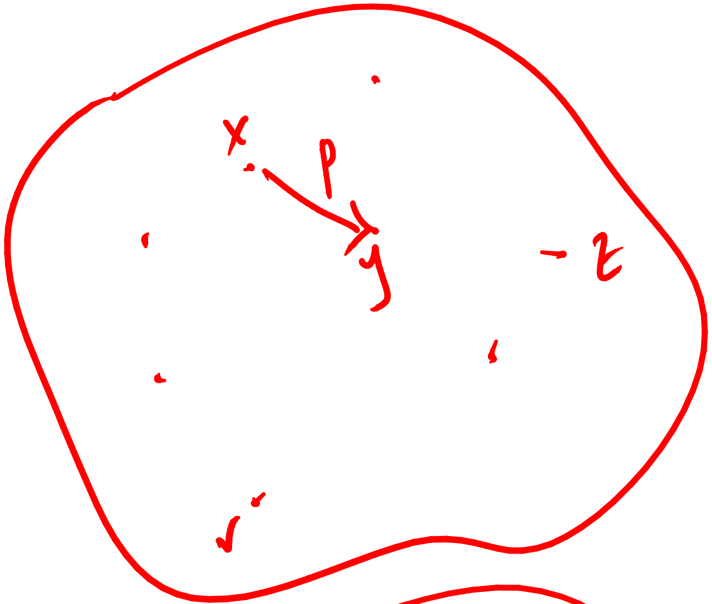
Separation operator: $*$ to combine heaplets

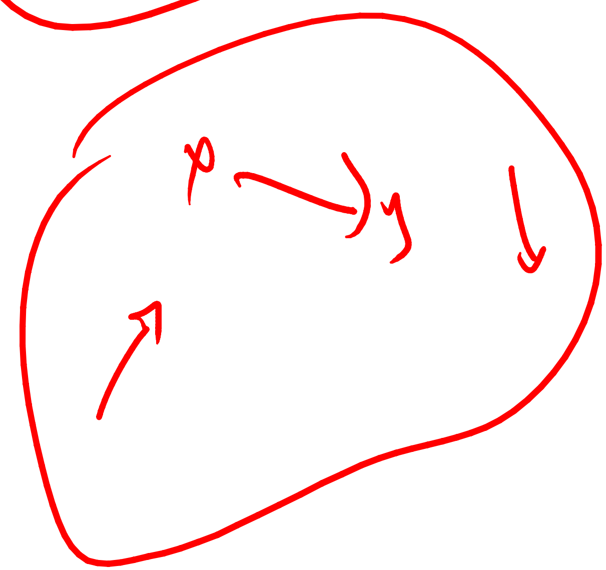
E.g., $x \rightarrow y$ is true on a heaplet where the pointer fields from only the node {x} is defined; not true on larger heaps!

$\mathrm{p} * \mathrm{q}$ is true on a heaplet H if it can be split into two disjoint heaplets, one satisfying p and one satisfying q

SEPARATION LOGIC

$x \overset{p}{\longmapsto} y$



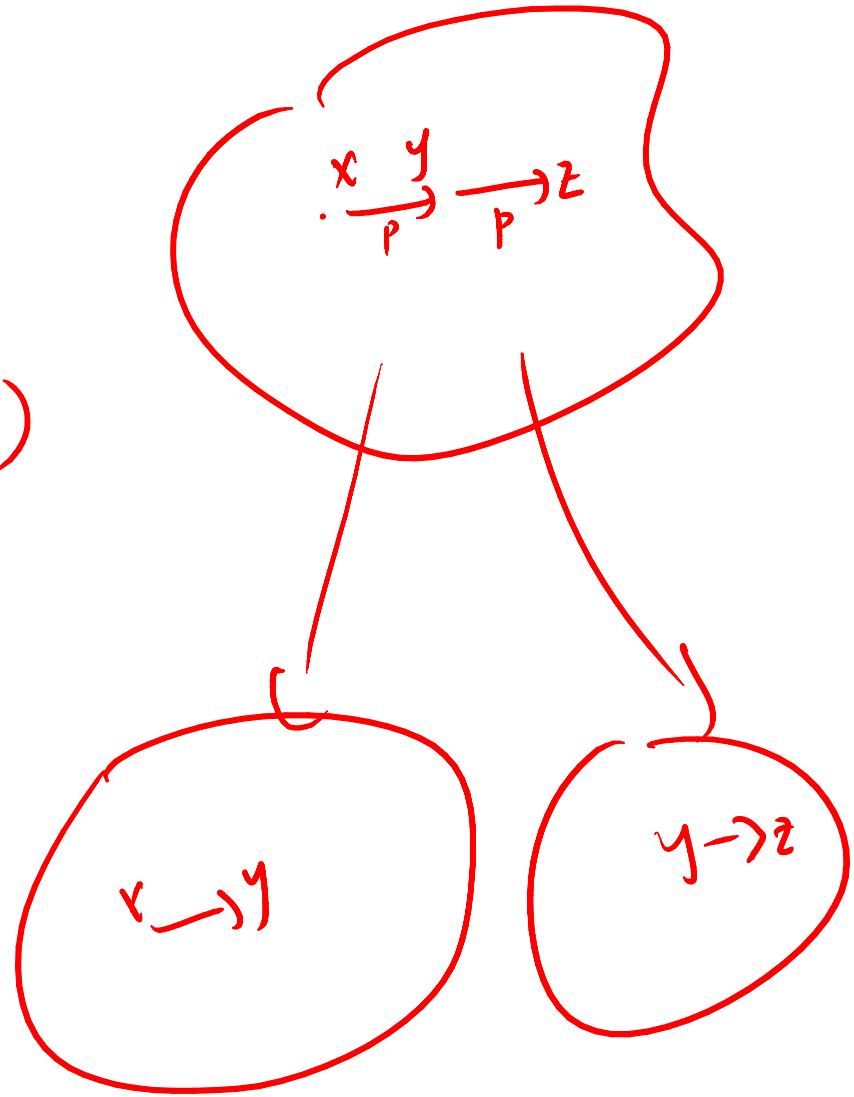


$x \xrightarrow{p} y$ ✗

SEPARATION LOGIC

$$x \xrightarrow{p} y \quad \wedge \quad y \xmapsto{p} z$$

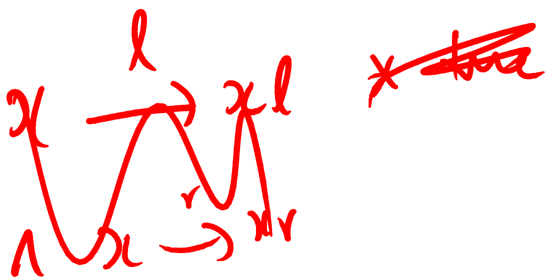
$$\left(x \xrightarrow{p} y\right) \; * \; \left(y \to z\right)$$

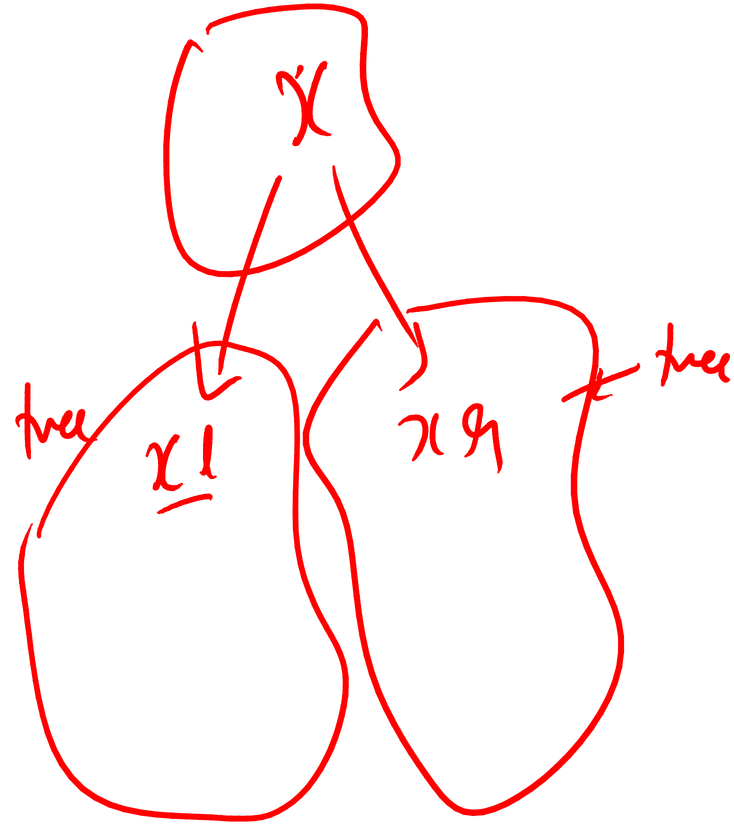


SEPARATION LOGIC

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

$$(x \overset{l,r}{\mapsto} xl, xr) * tree^{*}(xl) * tree^{*}(xr)$$

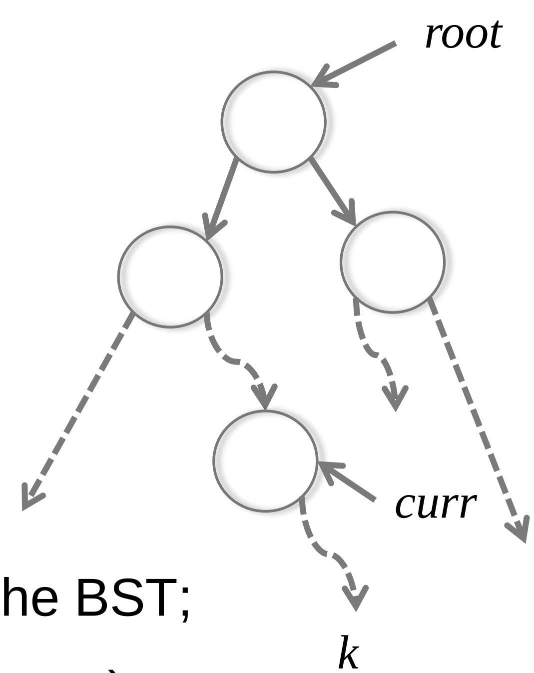




BINARY SEARCH TREE: AN EXAMPLE USING DRYAD

$$bst(x) \overset{def}{=} (x = \text{nil} \wedge \text{emp}) \vee$$
$$(x \overset{l,r,key}{\mapsto} xl, xr, xk) * \big(bst(xl) \wedge keys(xl) < \{xk\}\big) * \big(bst(xr) \wedge \{xk\} < keys(xr)\big)$$

$$keys(x) \overset{def}{=} (x = \text{nil}: \emptyset;$$
$$(x \overset{l,r,key}{\mapsto} xl, xr, xk) * \text{true}: \{xk\} \cup keys(xl) \cup keys(xr);$$
$$\text{default}: \emptyset)$$



$$\psi \equiv bst(root) \wedge (bst(curr) * \text{true})$$
$$\wedge (k \in keys(root) \leftrightarrow k \in keys(curr) * \text{true})$$

(loop invariant for “*bst-search*”:

root points to a BST and *curr* points into the BST;

k is stored in the BST iff *k* is stored under *curr*)

SYNTAX OF DRYAD

$pf \in PF$ $\quad i^* : Loc \rightarrow Int_L$ $\quad p^* : Loc \rightarrow Bool$ $\quad x \in Loc$ Variables

$df \in DF$ $\quad si^* : Loc \rightarrow \mathcal{S}(Int)$ $\quad S \in \mathcal{S}(Int)$ Variables $\quad j \in Int_L$ Variables

$c : Int_L$ Constant $\quad msi^* : Loc \rightarrow \mathcal{MS}(Int)_L$ $\quad MS \in \mathcal{MS}(Int)_L$ Variables $\quad q \in Bool$ Variables

$$
\begin{array}{rlcl}
Loc \text{ Term:} & lt & ::= & x \mid \texttt{nil} \\
Int_L \text{ Term:} & it & ::= & c \mid j \mid i^*_{\vec{pf},\vec{t}}(lt) \mid it + it \mid it - it \\
\mathcal{S}(Loc) \text{ Term:} & slt & ::= & \emptyset_l \mid L \mid \{lt\} \mid sl^*_{\vec{pf},\vec{t}}(lt) \mid slt \cup slt \mid slt \cap slt \mid slt \setminus slt \\
\mathcal{S}(Int) \text{ Term:} & sit & ::= & \emptyset_s \mid S \mid \{it\} \mid si^*_{\vec{pf},\vec{t}}(lt) \mid sit \cup sit \mid sit \cap sit \mid sit \setminus sit \\
\mathcal{MS}(Int)_L \text{ Term:} & msit & ::= & \emptyset_m \mid M \mid \{it\}_m \mid msi^*_{\vec{pf},\vec{t}}(lt) \mid msit \cup msit \mid msit \cap msit \mid msit \setminus msit \\
\\
\text{Positive Formula:} & \varphi & ::= & \texttt{true} \mid q \mid p^*_{\vec{pf},\vec{t}}(lt) \mid \texttt{emp} \mid lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt}, \vec{it}) \mid lt = lt \mid lt \neq lt \mid it \leq it \mid it < it \mid \\
& & & slt \subseteq slt \mid slt \not\subseteq slt \mid sit \subseteq sit \mid sit \not\subseteq sit \mid msit \sqsubseteq msit \mid msit \not\sqsubseteq msit \mid \\
& & & lt \in slt \mid lt \notin slt \mid it \in sit \mid it \notin sit \mid it \in msit \mid it \notin msit \mid sit \leq sit \mid sit < sit \mid \\
& & & msit \leq msit \mid msit < msit \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi * \varphi \\
\text{Formula:} & \psi & ::= & \varphi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg \psi
\end{array}
$$

RECURSIVE DEFINITIONS

Recursive Functions

$Loc \rightarrow Int_L$, $Loc \rightarrow S(Loc)$, $Loc \rightarrow S(Int)$, $Loc \rightarrow MS(Int)_L$

$$f^*_{\overrightarrow{pf},\vec{t}}(x) \stackrel{def}{=} (\varphi^f_1(x,\vec{t},\vec{v}) : t^f_1(x,\vec{v})\ ;\ \ldots\ ;\ \varphi^f_k(x,\vec{t},\vec{v}) : t^f_k(x,\vec{v})\ ;\ \text{default} : t^f_{k+1}(x,\vec{v}))$$

Recursive Predicates

$$p^*_{\overrightarrow{pf},\vec{t}}(x) \stackrel{def}{=} \varphi^p(x,\vec{t},\vec{v})$$

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 $lseg^*_{next,y}(x)$

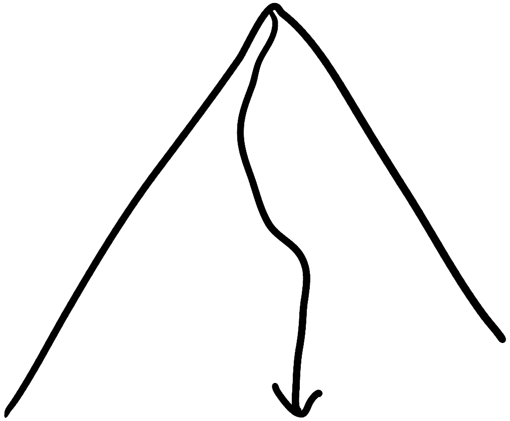
SEMANTICS:RECURSIVE DEFINITIONS

To evaluate a recursive definition $f^{*}_{\overrightarrow{pf},\vec{t}}(lt)$ **over a heaplet *h*:**

- compute the reach set $Reach_{lt}$ with respect to $\overrightarrow{pf}$ and $\vec{t}$
- if $Dom(h) = Reach_{lt}$, then evaluate it as the least fixpoint of the recursive definitions
- otherwise, evaluate to *"undef"*

Example

$$U_{\{l,r\}}(x) \stackrel{def}{=} (x = \mathrm{nil} \wedge emp) \vee \left(x \stackrel{l,r}{\mapsto} y, z * U_{\{l,r\}}(y) \right) \vee \left(x \stackrel{l,r}{\mapsto} y, z * U_{\{l,r\}}(z) \right)$$



DRYAD: $U_{\{l,r\}}(x)$ is satisfied by the entire tree

Conventional SL: $U_{\{l,r\}}(x)$ is satisfied by any path in tree!

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 op t' *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: $\boldsymbol{tree}^*(\boldsymbol{x}) * \boldsymbol{tree}^*(\boldsymbol{y})$

can be translated to

$$tree(x) \wedge tree(y) \wedge$$
$$\mathrm{reach}(x) \cap \mathrm{reach}(y) = \emptyset \wedge$$
$$\mathrm{reach}(x) \cup \mathrm{reach}(y) = G$$

(still quantifier-free)

NATURAL PROOFS FOR DRYAD

We consider programs with modular annotations

- pre/post, loop invariant in DRYAD
- *ret* denotes the returned value

$$
\begin{array}{lcl}
P & :- & P\,;P \mid stmt \\
stmt & :- & u := v \mid u := \mathtt{nil} \mid u := v.pf \mid u.pf := v \\
 & & \mid\; j := u.df \mid u.df := j \mid j := aexpr \\
 & & \mid\; u := \mathtt{new} \mid \mathtt{free}\; u \mid \mathtt{assume}\; bexpr \\
 & & \mid\; u := f(\vec{v}, \vec{z}) \mid j := g(\vec{v}, \vec{z}) \\
aexpr & :- & int \mid j \mid aexpr + aexpr \mid aexpr - aexpr \\
bexpr & :- & u = v \mid u = \mathtt{nil} \mid aexpr \le aexpr \\
 & & \mid\; \neg bexpr \mid bexpr \vee bexpr
\end{array}
$$

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, …, 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 ψ_{VC} involves recursive definitions that can be unfolded 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

$$tree_i(u) \leftrightarrow \begin{pmatrix} (u = \mathrm{nil} \wedge \mathrm{reach}_i(u) = \emptyset) \ \vee \\ (tree_i(ul) \wedge tree_i(ur) \wedge \ldots\ ) \end{pmatrix}$$

- Make the recursive definitions consistent across the timestamp:

$$\begin{aligned} \psi'_{\mathrm{VC}} \ \equiv\ & \psi_{\mathrm{VC}} \wedge \textsc{Unfold} \wedge \textsc{Footprint} \\ & \wedge \textsc{SegUnchanged} \wedge \textsc{CallUnchanged} \wedge \textsc{SelfReach} \end{aligned}$$

Theorem

- ψ_{VC} is valid iff ψ'_{VC} is valid.

FORMULA ABSTRACTION

Natural Proofs in two steps:

$$\psi_{VC} \overset{unfold}{\Rightarrow} \psi'_{VC} \overset{\substack{formula \\ abstraction}}{\Rightarrow} \psi^{abs}_{VC}$$

Formula Abstraction

- replace each recursive definition rec with uninterpreted $\widehat{rec}$
- replace the corresponding reach set $Reach^{rec}$ with uninterpreted H^{rec}
- when a proof for ψ^{abs}_{VC} is found, we call it a natural proof for ψ_{VC}

  (sound but incomplete)

ψ^{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)

$S_1 \subseteq S_2$ between integer sets?

translated to $\forall i_1, i_2. \left(i_1 < i_2 \rightarrow (\neg S_2[i_1] \vee \neg S_1[i_2])\right)$

(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

Datastructure	Routines	Time (s) / routine
Singly linked lists	find, insert_front, insert_back, delete_all, copy, append, reverse	1s
Sorted lists	find, insert, merge, delete_all, insert_sort, reverse, find_last, insert*, quicksort	9s
Doubly-linked lists	insert_front, insert_back, delete_all, append, mid_insert, mid_delete, meld	1s
Cyclic lists	Insert_front, insert_back, delete front, delete_back	1s
Max-heap	heapify	9s
Binary search trees	find, find*, insert, delete, remove_root, find_leftmost, remove_root, delete	47s
Treap	find, delete, insert, remove_root	7s
AVL trees	balance, leftmost, insert, remove	5s
Red-black trees	Insert, insert_left,fix, insert_right_fix, delete, delete_left_fix, delete_right_fix, leftmost	16s
Binomial heap	find_min, merge	78s
Tree traversals	Inorder_tree_to_list, inorder_tree_to_list*, preorder, postorder, inorder	10s

Package	Routines	Time (s) /rout
schorr-waite	marking_iter	1s
glib/gslist.c Singly linkedlist(1.1K)	free, prepend, concat, insert_before, remove_all, remove_link, delete_link, copy, reverse, nth, nth_data, find, position, index, last, length	1s
	append, insert_at_pos, remove, insert_slist, merge_slists, merge_sort	7s
glib/glist.c DoublyLinkedlst(0.3K)	free, prepend, reverse, nth, nth_data, position, find, index, last, length	1s
	quicksort_iter	65s
openbsd/queue.h LOC 0.1K	simpleq_init, simpleq_remove, simpleq_insert_head, simpleq_insert_tail, simpleq_insert_after, simpleq_remove_head	6s
secureOS/ cachepage(0.1K)	Lookup_prev, add_cachepage	4s
secureOS/ memRegion (0.1K)	memory_region_init, create_user_space_reg, split_memory_region	4s
linux/mmap.c (0.1K)	find_vma, remove_vma, remove_vma_list	1s

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*

THANK YOU!

Natural proof for
tree ds [POPL'12]

Natural proofs for
sep logic [submitted]

Building reliable
software with
proven properties

SecureOS [ASPLOS13]
[Mai, King, Pek, Xue]

DEDUCTIVE VERIFICATION WITH AUTOMATED THM PROVING

Concurrency;
rely-guarantee

Combining dec
sep logic with
rely/guar

%

Natural proofs for C

NP -> ghost code in VCC

Ongoing work:
[Pek, Qiu]

Strengthen natural proofs;
strengthen pre/post/LI
automatically

Ongoing work:
[Viswanathan, Qiu]

Learning loop
invariants

Ongoing work:
[Garg, Loding,
Neider]

SCRATCH