Verifying Programs that Manipulate Pointers

Anders Møller

University of Aarhus

BRICS

http://www.brics.dk/~amoeller/talks/infinity.pdf

Analyzing the Heap

Heap: *an untidy pile or mass of things*

[Cambridge Dictionary]

- This is how most program analyses view the heap
- because pointers are notoriously difficult to reason about
- but they are an important part of most programming languages…

Example: Reversing a Linked List

```
struct Node {
  struct Node *n;
  int data;
}

Node *reverse(Node *x) {
  Node *y, *t;
  y = NULL;
  while (x != NULL) {
    t = y;
    y = x;
    x = x->n;
    y->n = t;
  }
  return y;
}
```

Assume that the input is an acyclic list, argue that

- there are no null pointer dereferences
- no elements are lost
- the output is an acyclic list
- the output is the reverse of the input
- no other parts of the heap are modified

Why is this difficult?

In addition to the usual problems with software verification,

- destructive updating through pointers (x->f = y) can make **complex structures**
- the heap has **unbounded size**
- **data-structure invariants** typically only hold at the beginning and end of operations

Three Approaches

- **Separation Logic** (Reynolds & O'Hearn)
  - an extension of Hoare logic
- **Parametric Shape Analysis** (Sagiv, Reps & Wilhelm)
  - data-flow analysis using three-valued logic
- **Pointer Assertion Logic** (Møller & Schwartzbach)
  - using monadic second-order logic on trees

Hoare Logic

program statement

Hoare triple: $\{P\}\ S\ \{Q\}$

precondition postcondition

- ***partial correctness***: if S is executed in a store initially satisfying P and it terminates, then the final store satisfies Q
- the ***assertion language*** is typically predicate logic with transitive closure or inductively defined predicates

Hoare Logic and Pointers

Consider the standard axiom for assignment:

$$\frac{}{\{Q[E/X]\}\ \ X=E;\ \ \{Q\}}$$

Example: $\{y+7>42\}$ x=y+7; $\{x>42\}$

This is unsound in presence of pointers!

Example: ~~{y->a=42} x->a=7; {y->a=42}~~

aliasing?

A possible solution?

Idea: view the heap as an array for each field

$$\overline{\{Q[F[X \mapsto E]/F]\}\ \texttt{X->F=E;}\ \{Q\}}$$

This works, but forces ***global reasoning***

- every heap assignment seems to affect every heap assertion

Reversing a Linked List, cont.

Consider a possible **loop invariant**:

$$\exists\alpha,\beta\text{: LIST}[\alpha](x) \wedge \text{LIST}[\beta](y) \wedge \alpha_0^R = \alpha^R \cdot \beta$$

Unfortunately, it is **not enough**:

- we must explicitly **forbid sharing** between the x and y lists
- we must explicitly state that
  every other part of the heap is unaffected!
  - this makes modular specifications impossible

Separation Logic

New assertions:

- **emp** (empty heap)
- $E \mapsto F_1{:}E_1,..,F_n{:}\ E_n$ (one-cell heap)
- $P * Q$ (separating conjunction)
- $P -\!\!* \ Q$ (separating implication)

Axioms

Axiom for assignment *to* the heap ("mutation"):

$$\overline{\{X \mapsto \rho,\ F{:}\_,\sigma\}\quad X\text{->}F = E; \qquad \{X \mapsto \rho, F{:}E,\sigma\}}$$

Axiom for assignment *from* the heap ("lookup"):

$$\overline{\{Y \mapsto \rho, F{:}V,\sigma\}\quad X = Y\text{->}F; \qquad \{X{=}V \wedge Y \mapsto \rho, F{:}V,\sigma\}}$$

- variants for backwards reasoning also exist
- allocation/disposal, pointer arithmetic, etc., also works

The Frame Rule, Local Reasoning

$$\frac{\{P\}\ \ S\ \ \{Q\}}{\{P * R\}\ \ S\ \ \{Q * R\}}$$

- all heap cells that are not mentioned in the specification are guaranteed to remain unchanged!

- many other new inference rules...

- now the loop invariant works immediately for the list reversal example:

$$\exists \alpha,\beta\colon \big(\mathrm{LIST}[\alpha](x) * \mathrm{LIST}[\beta](y)\big) \wedge \alpha_0^R = \alpha^R \cdot \beta$$

Parametric Shape Analysis

Automatic inference of "shape invariants"
using **data-flow analysis** with **three-valued logic**

Representing Concrete Stores using Logic

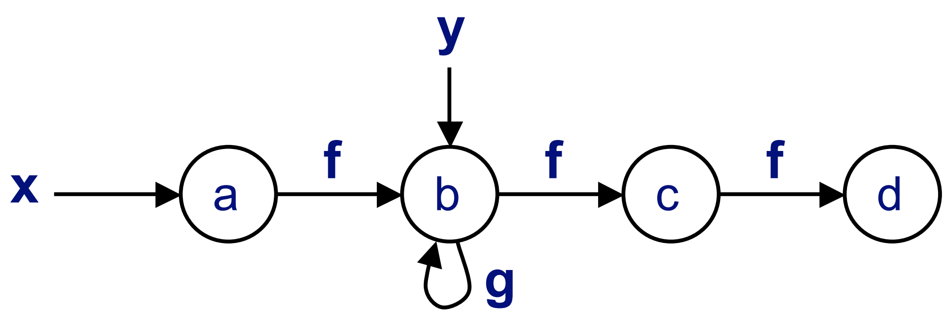
A **logical structure** consists of

- a **universe** of elements
- a family of basic **predicates**

- let the universe represent the set of **heap cells**
- each **variable** x is described by a *unary* predicate x(p)
- each **field** f is described by a *binary* predicate f(p,q)

Example of a Logical Structure

The store



is described by the logical structure whose universe is {a,b,c,d} and the basic predicates are interpreted by:

	a	b	c	d
x	1	0	0	0
y	0	1	0	0

f	a	b	c	d
a	0	1	0	0
b	0	0	1	0
c	0	0	0	1
d	0	0	0	0

g	a	b	c	d
a	0	0	0	0
b	0	1	0	0
c	0	0	0	0
d	0	0	0	0

Queries

Queries are expressed in
first-order logic with transitive closure

- Are x and y pointer aliases?

  $\exists p: x(p) \wedge y(p)$

- Does x point to a cell with a self cycle?

  $\exists p: x(p) \wedge n(p,p)$

- Do x and y refer to disjoint structures?

  $\neg\exists p,q,r: x(p) \wedge y(q) \wedge n^*(p,r) \wedge n^*(q,r)$

Operational Semantics by Predicate Transformation

Statements that modify the store can be described using *predicate transformation*:

- x = NULL — $x'(p) = 0$
- x = y — $x'(p) = y(p)$
- x = y->f — $x'(p) = \exists q{:}\ y(q) \wedge f(q,p)$
- y->f = x — $f'(p,q) = \big(y(p) \wedge x(q)\big) \vee \big(\neg y(p) \wedge f(p,q)\big)$

Program Analysis

How can this be used as a basis for program analysis?

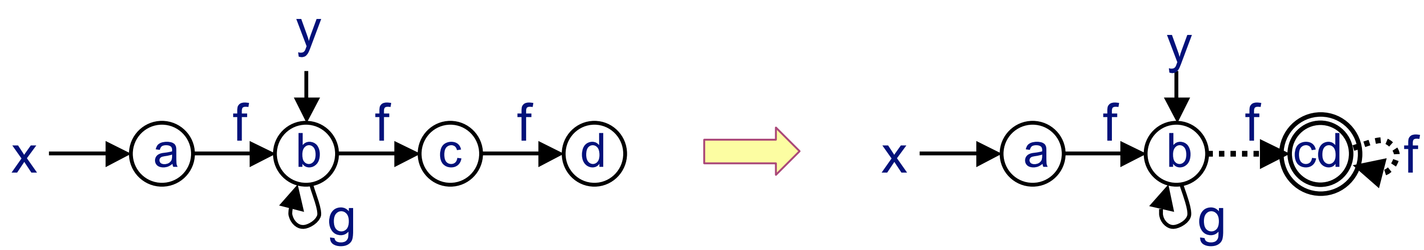
- Idea: use the **standard data-flow analysis framework** with the **lattice** being **sets of logical structures**

- however, the concrete structures have unbounded size (the lattice needs to have finite height)…

- we need some **abstraction**!

Canonical Abstraction

Collapse elements that cannot be distinguished by unary predicates



- the lattice of sets of these abstract structures is finite!

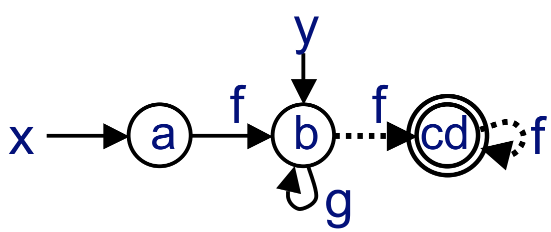
Kleene's 3-Valued Logic

- 0 = false/never
- 1 = true/always
- **$^1/_2$ = don't know**

	0	1	$^1/_2$
$\neg$	1	0	$^1/_2$

$\wedge$	0	1	$^1/_2$
0	0	0	0
1	0	1	$^1/_2$
$^1/_2$	0	$^1/_2$	$^1/_2$

Abstract Structures are 3-Valued Logic Structures



	a	b	cd
x	1	0	0
y	0	1	0

f	a	b	cd
a	0	1	0
b	0	0	$^1/_2$
cd	0	0	$^1/_2$

g	a	b	cd
a	0	0	0
b	0	1	0
cd	0	0	0

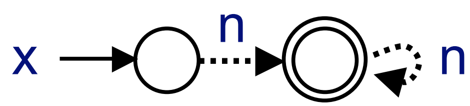
- a 3-valued structure S represents a set of 2-valued structures $\{T_1, T_2, \ldots\}$

- evaluating a query formula Φ on S is a conservative approximation of evaluating it on any T_i

The Analysis - Abstract Interpretation

- The **lattice** consists of **sets of 3-valued structures**
- Concrete **operational semantics** is defined using predicate transformation on **2**-valued structures
- Abstract **transfer functions** can be defined using predicate transformation on **3**-valued structures

Instrumentation Predicates

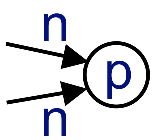


- How do we distinguish **cyclic** lists from **acyclic** lists?
- or recognize **disjoint** lists?
- or handle **doubly-linked** lists?
- or …?

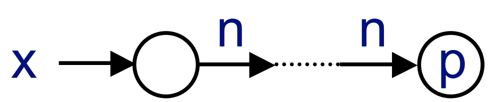
- Allow extra "**instrumentation predicates**" to be defined (in terms of the core predicates)
- *Canonical abstraction* considers all unary predicates, including instrumentation predicates!

Examples of Instrumentation Predicates

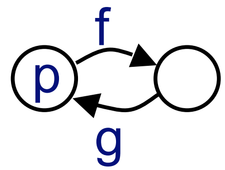
- $is_n(p) = \exists q,r\colon n(q,p) \wedge n(r,p) \wedge q \neq r$
  - do two or more n fields point to p?



- $r_{x,n}(p) = \exists q\colon x(q) \wedge n^*(q,p)$
  - is p reachable from x through n fields?



- $c_{f,g}(p) = \forall q,r\colon f(p,q) \wedge g(q,r) \Rightarrow p=r$
  - does an f dereference from p followed by a g dereference yield p?



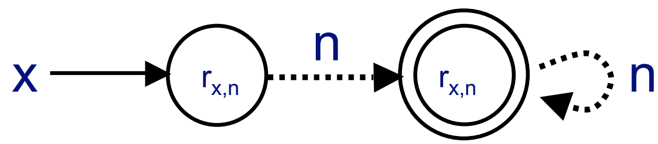
Using Instrumentation Predicates

- The **choice** of instrumentation predicates depends on the application
- Adding instrumentation predicates improves **precision** but decreases **worst-case performance**

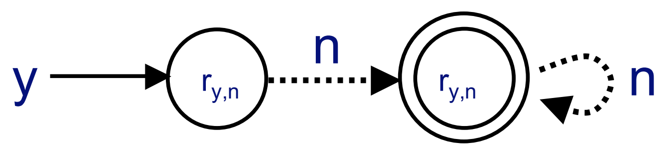
- **Predicate transformation** for instrumentation predicates must also be defined - and proven **sound** relative to the core predicates

Analyzing the List Reversal Program

- Instrumentation predicates:
  - $r_{x,n}(p)$, $r_{y,n}(p)$, $r_{t,n}(p)$ ("*reachability along* n *from* x/y/t")
  - $is_n(p)$ ("*pointed to by more than one* n *field*")
  - $c_n(p)$ ("*cycle along* n *fields*")
- Input structure:



- Running the TVLA tool:
  - fixed point found after 4 iterations
  - output structure:



  - i.e., all cells are moved from x to y without introducing loops!

Pointer Assertion Logic

- Describe stores using **graph types** and **monadic second-order logic on trees** (M2L-Tree)
- Split the programs into **Hoare triples without loops and procedure calls** using explicit invariants
- **Encode** each triple as an M2L-Tree formula and run the MONA decision procedure

- relation to Separation Logic: everything is decidable here
- relation to Parametric Shape Analysis: no abstraction, explicit invariants

List Reversal, revisited

```
struct Node { struct Node *n; int data; }

pred ROOTS(pointer x,y:Node, set R:Node) =
 allpos p of Node:  p in R <=> x<n*>p | y<n*>p

Node *reverse(Node *x)
  set R:Node [ROOTS(x,NULL,R)]
{
  Node *y, *t;
  y = NULL;
  while (x != NULL) [ROOTS(x,y,R)] {
    t = y;
    y = x;
    x = x->n;
    y->n = t;
  }
  return y;
} [ROOTS(NULL,return,R)]
```

In 0.5 secs., the PALE tool verifies that
- all nodes are moved
- no cycles are introduced
- there can be no memory errors

M2L-Tree and Graph Types

- M2L-Tree:
  - monadic second-order logic
  - on tree structures
- Graph types:
  - tree structures specified by recursive types
  - with extra pointers specified by M2L-Tree formulas

Pointer Assertion Logic: M2L-Tree + Graph types

- allows many common heap structures to be expressed (doubly-linked lists, trees with parent pointers, etc.)
- reducible to M2L-Tree and decidable

Encoding Hoare triples in M2L-Tree

Use ***transductions*** to encode loop-free code:

- ***Store predicates*** model the store at each program point

- ***Predicate transformation*** models the semantics of statements
  Example: **`x = y.next;`** $\rightarrow$ $x'(p) = \exists q.\ y(q) \wedge next(q,p)$

- ***Verification condition*** is constructed by expressing the pre- and post-condition using store predicates from end points

- Each triple is then verified separately (by the MONA tool)
- If invalid, a concrete counterexample store is generated!

Summary / Conclusion

Reasoning about programs that use pointers is challenging...

- **Separation Logic**
  - Hoare logic with *separating conjunction*
  - supports *local reasoning* and often permits succinct proofs
- **Parametric Shape Analysis**
  - *data-flow analysis* where stores are represented abstractly using *three-valued logical structures*
  - allows precision/performance to be controlled by the choice of *instrumentation predicates*
- **Pointer Assertion Logic**
  - *decidable* application of Hoare logic
  - requires *invariants*, but high expressiveness and modularity

- Still at research level, not really practically useful (yet?)

References

- *Separation Logic: A Logic for Shared Mutable Data Structures*
  J.C. Reynolds, LICS 2002

- *Parametric Shape Analysis via 3-Valued Logic*
  M. Sagiv, T. Reps, and R. Wilhelm, TOPLAS 20(1)

- *The Pointer Assertion Logic Engine*
  A. Møller and M.I. Schwartzbach, PLDI 2001