Static Analysis: Pointers & Heap Structures

http://d3s.mff.cuni.cz

Pavel Parízek
Goals
- Determine possible targets objects for each pointer variable
- Find possibly aliased program variables of a reference type (pointers)

Very important for programs that use heap and objects
- Mainstream languages: C, C++, Java, C#, Scala
- Aspects: virtual methods (call graphs), aliasing
Example program

```java
1:  void main() {
2:      Customer c1 = loadCustomerData(1);
3:      Customer c2 = loadCustomerData(2);
4:      if (c2 == null) c2 = new Customer();
5:      Region r = new Region("Praha");
6:      c1.reg = r;
7:      c2.reg = r;
8:      c1.reg = new Region("Brno");
9:      List<Order> orders = c2.reg.getNewOrders();
10:     orders.process();
11:  }
12:  Customer loadCustomerData(int id) {
13:      Customer c = new Customer(id);
14:      return c;
15:  }
```
Terminology

• Abstract heap object
  - Allocation site (o := new C)
  - Set of dynamic heap objects

• Points-to set
  - Set $pt(p)$ of abstract heap objects that the pointer variable $p$ may point to during program execution

• Aliased variables
  $$pt(p) \cap pt(r) \neq \emptyset$$
Points-to analysis

- Determines the points-to set $pt(p)$ for each pointer variable $p$ in a given program.

- Characteristics
  - Safe over-approximation
  - $x := y \Rightarrow pt(y) \subseteq pt(x)$

- Algorithms
  - Basic: exhaustive subset-based flow-insensitive context-insensitive (Andersen)
  - Advanced: flow-sensitive, context-sensitive (few kinds), demand-driven, strong updates, ...
  - Trade-offs: scalability versus precision
Example: computing points-to sets

Q1: Find the points-to set for the variable c2.

Q2: Find all the aliased variables and fields.
• **May-alias**
  - Two variables may possibly refer to the same heap object at some point during execution

• **Must-alias**
  - Two variables must always refer to the same heap object at a specific program point
Modeling updates

• Weak update (may-alias)
  - Given operation on $p$ may or may not be actually performed on any element of the set $pt(p)$

• Strong update (must-alias)
  - Operation performed on $p$ and other variables provably aliased with $p$ at a given point
Computing must-alias information

- Allocation sites
  - Fixed partitioning of the heap
  - Fixed name for a heap object

- Access path
  - Variable name followed by a possibly empty sequence of field names (dereferences)
  - Example: p, p.f.g, q.f

- Set of access paths
  - Dynamically changing name for abstract heap object
Tracking access paths

- Abstract heap object \( o \)
  - Tuple \(<o, \text{set of access paths}>\)

- Processing statements
  - Current tuple (old): \(<o, AP_{\text{old}}>\)
  - Object allocation: \( v = \text{new } C \)
    - New tuple: \(<o, \{v}\>\)
  - Assignment: \( v = e \)
    - New tuple: \(<o, AP_{\text{old}} \cup \{ v.ap \mid e.ap \in AP_{\text{old}} \}>\)
  - Assignment: \( v.f = e \)
    - New tuple: \(<o, AP_{\text{old}} \cup \{ v.f.ap \mid e.ap \in AP_{\text{old}} \}>\)
  - Assignment: \( v = \text{null} \)
    - New tuple: \(<o, AP_{\text{old}} \setminus \{ v.ap \mid ap \in AP_{\text{old}} \}>\)
Applications

- Client analyses
  - Call graph construction
  - Escape analysis
    - Scope: method, thread

- Verification
  - Null pointer dereference
  - Static data race detection
  - Resource leaks detection
Null pointer dereference (NPA)

- Option 1: use classic data-flow analysis
- Option 2: use results of pointer analysis
NPA: data-flow analysis

- Analysis domain: list of pointer variables
- Facts: variables with possible null value
- Transfer functions: assignment (null, ...)
- Merge operator: set union (over-approx)

Processing results

- For each dereferencing statement check whether the results say that a given pointer may be null
- Statements: field access, method call, array access
NPA: using pointer analysis

• Input
  ▪ Results of the may point-to analysis
  ▪ Specific dereference operation on \( v \)

• Empty points-to set \( pt(v) \)
  ➔ possible null value
Call graph construction

- Goal: for each call site, find the set of possibly invoked methods

- Statement: \( r = v.m(a_1, \ldots, a_N) \)

Approaches
- Class Hierarchy Analysis (CHA)
  - static type (class) of \( v \) and all possible subtypes
- Using results of pointer analysis
  - dynamic types of abstract heap objects in \( pt(v) \)
Escape analysis

- **Method scope**
  - Goal: identify objects written to heap \((v.f = o)\)
  - Purpose: local objects may be safely reclaimed

- **Thread scope**
  - Goal: identify possibly shared heap objects
    - shared object = reachable from multiple threads
  - Purpose: eliminating thread choices (POR)
  - Algorithm: escaping roots, transitive reachability
Static analysis in program verification

- Constructing abstraction
- Intermediate representation
- Program slicing
  - Find and remove statements irrelevant for the given property
Method summaries

- **Purpose**: scalable inter-procedural analysis

- **Approach**
  - Use available method summary for $M$
  - Ignore edges: call - entry, return - exit

- **Example**: side effects analysis
  - Field accesses on shared heap objects
  - Parameters escaped inside to the heap
Pointer analysis in WALA

- Heap graph

- Nodes
  - PointerKey: local variables, fields
  - InstanceKey: allocation sites

- Edges
  - points-to relation: PointerKey $\rightarrow$ InstanceKey
Examples

- Source code

- Collecting points-to sets
- Thread escape analysis
- Identify aliased variables
Advanced topics

- Shape analysis
- Separation logic
Shape analysis

• Goal
  ▪ Determine possible structure (shape) of the heap
  ▪ Find nodes to which the local variables may point

• Information
  ▪ Sharing between heap structures
  ▪ Cycles between nodes (pointers)
  ▪ Unreachable heap nodes (objects)

• Applications: garbage collection, detecting errors
Shape analysis: how it works

- **Representation (domain)**
  - Possible shapes of heap data structures for each program point

- **Abstraction (summarization)**
  - Summary heap nodes and edges
  - Loss of precision (length, depth)
Separation logic

• Goal
  - Reasoning about low-level programs that use mutable heap data structures

• Extends Hoare logic (triples \{P\} \textit{S} \{Q\})

• Logic operator \(\ast\) ("separating conjunction")
  - \(P \ast Q\) is true \(\Rightarrow\) disjoint heap structures

• Supports local reasoning (modularity)
Tools

- TVLA
  - http://www.cs.tau.ac.il/~tvla/

- Predator

- SLAyer
  - https://github.com/Microsoft/SLAyer

- jStar
  - https://github.com/seplogic/jstar

- Infer
Further reading

