Pointer Analysis – Part I

CS 6340

Pointer Analysis

- Answers which pointers can point to which memory locations at run-time
- Central to *many* program optimization & verification problems
- Problem is undecidable
  - No exact (i.e. both sound & complete) solution
- But many conservative (i.e. sound) approximate solutions exist
  - Determine which pointers *may* point to which locations
  - All incomplete but differ in precision (i.e. false-positive rate)
- Continues to be active area of research
  - hundreds of papers and tens of theses

Example Java Program

```
static void main() {
    String[] a = new String[] { “a1”, “a2” };
    String[] b = new String[] { “b1”, “b2” };
    List<String> l;
    l = new List<String>();
    for (int i = 0; i < a.length; i++) {
        String v1 = a[i];
        l.append(v1);
    }
    print(l);
    l = new List<String>();
    for (int i = 0; i < b.length; i++) {
        String v2 = b[i];
        l.append(v2);
    }
    print(l);
}
```

```
class Link<T> {
    T data;
    Link<T> next;
}

class List<T> {
    Link<T> tail;
    void append(T c) {
        Link<T> k = new Link<T>();
        k.data = c;
        Link<T> t = this.tail;
        if (t != null)
            t.next = k;
        this.tail = k;
    }
}
```

0-CFA Pointer Analysis for Java

- Flow sensitivity
  - flow-insensitive: ignores intra-procedural control flow
- Call graph construction
- Heap abstraction
- Aggregate modeling
- Context sensitivity

Flow Insensitivity: Example

```
static void main() {
    String[] a = new String[] { “a1”, “a2” };
    String[] b = new String[] { “b1”, “b2” };
    List<String> l;
    l = new List<String>();
    for (int i = 0; i < a.length; i++) {
        String v1 = a[*];
        l.append(v1);
    }
    l = new List<String>();
    for (int i = 0; i < b.length; i++) {
        String v2 = b[*];
        l.append(v2);
    }
}
```

```
class Link<T> {
    T data;
    Link<T> next;
}

class List<T> {
    T tail;
    void append(T c) {
        Link<T> k = new Link<T>();
        k.data = c;
        Link<T> t = this.tail;
        if (t != null)
            t.next = k;
        this.tail = k;
    }
}
```

Flow Insensitivity: Example

```
static void main() {
    String[] a = new String[] { “a1”, “a2” }
    String[] b = new String[] { “b1”, “b2” }
    List<String> l
    l = new List<String>()
    String v1 = a[*]
    l.append(v1)
    l = new List<String>()
    String v2 = b[*]
    l.append(v2)
}
class List<T> {
    T tail;
    void append(T c) {
        Link<T> k = new Link<T>()
        k.data = c
        Link<T> t = this.tail
        t.next = k
        this.tail = k
    }
}
```

0-CFA Pointer Analysis for Java

- Flow sensitivity
  - flow-insensitive: ignores intra-procedural control flow
- Call graph construction
  - “on-the-fly”: mutually recursively with pointer analysis
- Heap abstraction
- Aggregate modeling
- Context sensitivity

Call Graph (Base Case): Example

```
static void main() {
    String[] a = new String[] { “a1”, “a2” }
    String[] b = new String[] { “b1”, “b2” }
    List<String> l
    l = new List<String>()
    String v1 = a[*]
    l.append(v1)
    l = new List<String>()
    String v2 = b[*]
    l.append(v2)
}
class List<T> {
    T tail;
    void append(T c) {
        Link<T> k = new Link<T>()
        k.data = c
        Link<T> t = this.tail
        t.next = k
        this.tail = k
    }
}
```

Code deemed reachable so far …

0-CFA Pointer Analysis for Java

- Flow sensitivity
  - flow-insensitive: ignores intra-procedural control flow
- Call graph construction
  - "on-the-fly": mutually recursively with pointer analysis
- Heap abstraction
  - object alloc. sites: does not distinguish objects created at same site
- Aggregate modeling
- Context sensitivity

Heap Abstraction: Example

```
static void main() {
  String[] a = new String[] { "a1", "a2" }
  String[] b = new String[] { "b1", "b2" }
  List<String> l
  l = new List<String>()
  String v1 = a[*]
  l.append(v1)
  l = new List<String>()
  String v2 = b[*]
  l.append(v2)
}
class List<T> {
   T tail;
   void append(T c) {
      Link<T> k = new Link<T>()
      k.data = c
      Link<T> t = this.tail
      t.next = k
      this.tail = k
   }
}
```

Heap Abstraction: Example

```
static void main() {
  String[] a = new1 String[] { "a1", "a2" }
  String[] b = new2 String[] { "b1", "b2" }
  List<String> l
  l = new3 List<String>()
  String v1 = a[*]
  l.append(v1)
  l = new4 List<String>()
  String v2 = b[*]
  l.append(v2)
}
class List<T> {
   T tail;
   void append(T c) {
      Link<T> k = new5 Link<T>()
      k.data = c
      Link<T> t = this.tail
      t.next = k
      this.tail = k
   }
}
```

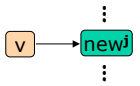
Heap Abstraction: Example

```
static void main() {
  String[] a = new1 String[] { "a1", "a2" }
  String[] b = new2 String[] { "b1", "b2" }
  List<String> l
  l = new3 List<String>()
  String v1 = a[*]
  l.append(v1)
  l = new4 List<String>()
  String v2 = b[*]
  l.append(v2)
}
class List<T> {
   T tail;
   void append(T c) {
      Link<T> k = new5 Link<T>()
      k.data = c
      Link<T> t = this.tail
      t.next = k
      this.tail = k
   }
}
```

Note: Pointer analyses for Java typically do not distinguish between string literals (like "a1", "a2", "b1", "b2" above): they use a single location to abstract them all

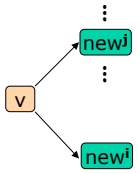
Rule for Object Alloc. Sites

- Before:



v = new^i ...

- After:

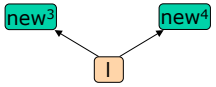


Note: This and each subsequent rule involving assignment is a "weak update" as opposed to a "strong update"

(i.e., it *accumulates* as opposed to *updates* the points-to information for the l.h.s.), a hallmark of flow-insensitivity

Rule for Object Alloc. Sites: Example

```
static void main() {
  String[] a = new1 String[] { "a1", "a2" }
  String[] b = new2 String[] { "b1", "b2" }
  List<String> l
  l = new3 List<String>()
  String v1 = a[*]
  l.append(v1)
  l = new4 List<String>()
  String v2 = b[*]
  l.append(v2)
}
class List<T> {
   T tail;
   void append(T c) {
      Link<T> k = new5 Link<T>()
      k.data = c
      Link<T> t = this.tail
      t.next = k
      this.tail = k
   }
}
```

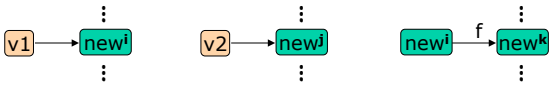


0-CFA Pointer Analysis for Java

- Flow sensitivity
  - flow-insensitive: ignores intra-procedural control flow
- Call graph construction
  - "on-the-fly": mutually recursively with pointer analysis
- Heap abstraction
  - object alloc. sites: does not distinguish objects created at same site
- Aggregate modeling
  - does not distinguish elements of same array
  - field-sensitive for instance fields
- Context sensitivity

Rule for Heap Writes

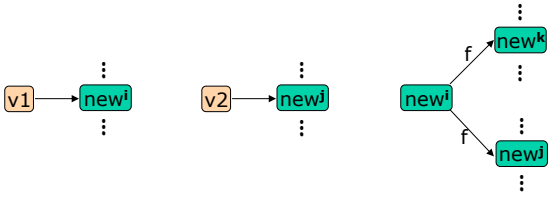
- Before:



v1.f = v2

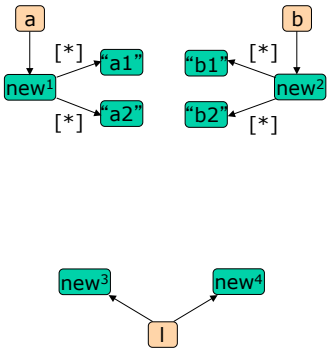
f is instance field or [*] (array element)

- After:



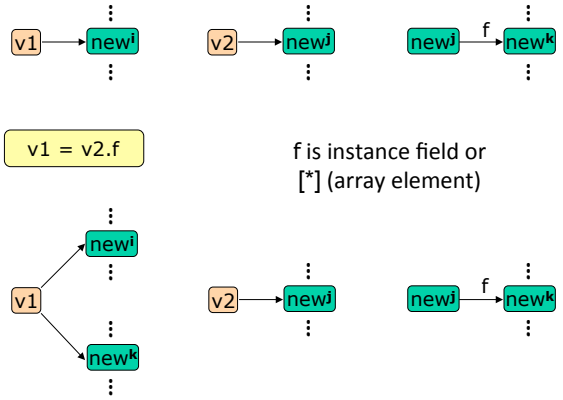
Rule for Heap Writes: Example

```
static void main() {
   String[] a = new1 String[] { "a1", "a2" }
   String[] b = new2 String[] { "b1", "b2" }
   List<String> l
   l = new3 List<String>()
   String v1 = a[*]
   l.append(v1)
   l = new4 List<String>()
   String v2 = b[*]
   l.append(v2)
}
class List<T> {
   T tail;
   void append(T c) {
      Link<T> k = new5 Link<T>()
      k.data = c
      Link<T> t = this.tail
      t.next = k
      this.tail = k
   }
}
```



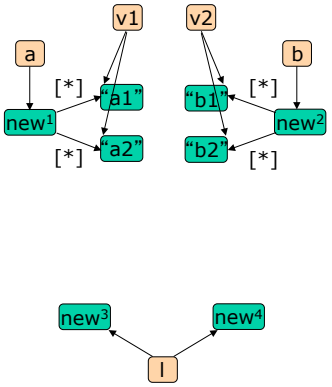
Rule for Heap Reads

- Before:
- After:



Rule for Heap Reads: Example

```
static void main() {
   String[] a = new1 String[] { "a1", "a2" }
   String[] b = new2 String[] { "b1", "b2" }
   List<String> l
   l = new3 List<String>()
   String v1 = a[*]
   l.append(v1)
   l = new4 List<String>()
   String v2 = b[*]
   l.append(v2)
}
class List<T> {
   T tail;
   void append(T c) {
      Link<T> k = new5 Link<T>()
      k.data = c
      Link<T> t = this.tail
      t.next = k
      this.tail = k
   }
}
```

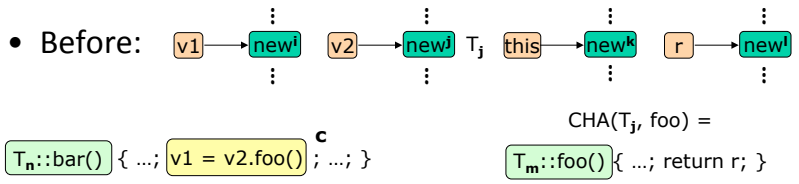


0-CFA Pointer Analysis for Java

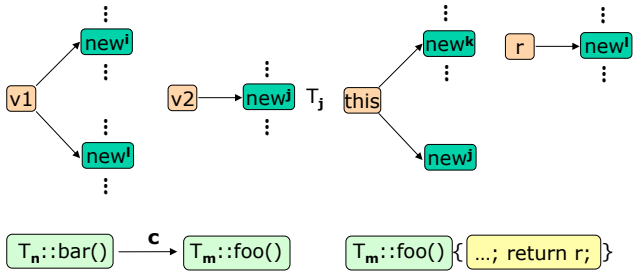
- Flow sensitivity
  - flow-insensitive: ignores intra-procedural control flow
- Call graph construction
  - "on-the-fly": mutually recursively with pointer analysis
- Heap abstraction
  - object alloc. sites: does not distinguish objects created at same site
- Aggregate modeling
  - does not distinguish elements of same array
  - field-sensitive for instance fields
- Context sensitivity
  - context-insensitive: ignores inter-procedural control flow (analyzes each function in a single context)

Rule for Dynamic Dispatching Calls

- Before:

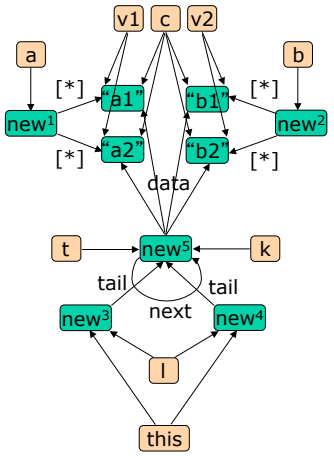


- After:



Call Graph (Inductive Step): Example

```
static void main() {
  String[] a = new1 String[] { "a1", "a2" }
  String[] b = new2 String[] { "b1", "b2" }
  List<String> l
  l = new3 List<String>()
  String v1 = a[*]
  l.append(v1)
  l = new4 List<String>()
  String v2 = b[*]
  l.append(v2)
}
class List<T> {
  T tail;
  void append(T c) {
    Link<T> k = new5 Link<T>()
    k.data = c
    Link<T> t = this.tail
    t.next = k
    this.tail = k
  }
}
```



Classifying Pointer Analyses

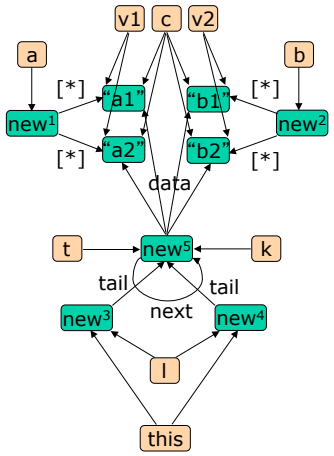
- Heap abstraction
- Alias representation
- Aggregate modeling
- Flow sensitivity
- Context sensitivity
- Call graph construction
- Compositionality
- Adaptivity

Heap Abstraction

- Single node for entire heap
  - Cannot distinguish between heap-directed pointers
  - Popular in stack-directed pointer analyses for C
- Object allocation sites ("0-CFA")
  - Cannot distinguish objects allocated at same site
  - Predominant pointer analysis for Java
- String of call sites ("k-CFA with heap specialization/cloning")
  - Distinguishes objects allocated at same site using finitely many strings of call sites
  - Predominant heap-directed pointer analysis for C
- Strings of object allocation sites in object-oriented languages ("k-object-sensitivity")
  - Distinguishes objects allocated at same site using finitely many strings of object allocation sites

Example

```
static void main() {
  String[] a = new1 String[] { "a1", "a2" }
  String[] b = new2 String[] { "b1", "b2" }
  List<String> l
  l = new3 List<String>()
  String v1 = a[*]
  l.append(v1)
  l = new4 List<String>()
  String v2 = b[*]
  l.append(v2)
}
class List<T> {
  T tail;
  void append(T c) {
    Link<T> k = new5 Link<T>()
    k.data = c
    Link<T> t = this.tail
    t.next = k
    this.tail = k
  }
}
```



Alias Representation

- Points-to Analysis: Computes the set of memory locations that a pointer may point to
  - Points-to graph represented explicitly or symbolically (e.g. using Binary Decision Diagrams)
  - Predominant kind of pointer analysis
- Alias Analysis: Computes pairs of pointers that may point to the same memory location
  - Used primarily by older pointer analyses for C
  - Can be computed using a points-to analysis: *may-alias*(v_1,v_2) if *points-to*(v_1) ∩ points-to(v_2) ≠ ∅

Aggregate Modeling

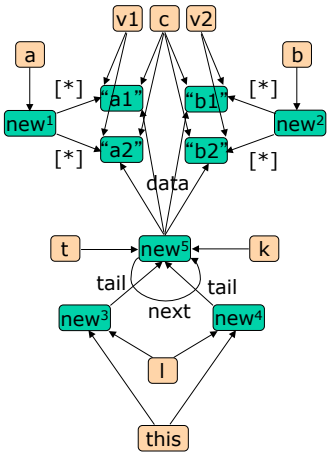
- Arrays
  - Single field ([*]) representing all array elements
  - Cannot distinguish elements of same array
  - But array dependence analysis used in parallelizing compilers is capable of making such distinctions
- Records/Structs
  - Field-insensitive/field-independent: merge *all* fields of *each* abstract record object
  - Field-based: merge *each* field of *all* record objects
  - Field-sensitive: model *each* field of *each* abstract record object (most precise)

Flow Sensitivity

- Flow-insensitive
  - Ignores intra-procedural control-flow (i.e. order of statements within a function)
  - Computes one solution for whole program or per function
  - Usually combined with Static Single Assignment (SSA) transformation to get limited flow sensitivity
  - Two kinds:
    - Steensgaard's or equality-based: almost linear time
    - Anderson's or subset-based: cubic time
- Flow-sensitive
  - Computes one solution per program point
  - Typically performs "strong updates" for assignments
  - More precise but typically less scalable

Example

```
static void main() {
  String[] a = new1 String[] { "a1", "a2" }
  String[] b = new2 String[] { "b1", "b2" }
  List<String> l
  l = new3 List<String>()
  String v1 = a[*]
  l.append(v1)
  l = new4 List<String>()
  String v2 = b[*]
  l.append(v2)
}
class List<T> {
  T tail;
  void append(T c) {
    Link<T> k = new5 Link<T>()
    k.data = c
    Link<T> t = this.tail
    t.next = k
    this.tail = k
  }
}
```

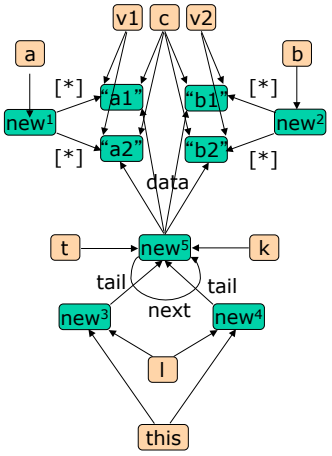


Context Sensitivity

- Context-insensitive
  - Ignores inter-procedural control-flow (does not match calls/returns)
  - Analyzes each function in a single abstract context
- Context-sensitive
  - Two kinds:
    - Cloning-based (k-limited)
      - k-CFA or k-object-sensitive (for object-oriented languages)
    - Summary-based
      - Top-down or bottom-up
  - Analyzes each function in multiple abstract contexts (cloning-based or top-down summary-based) or in a single parametric context (bottom-up summary-based)
  - More precise but typically less scalable

Example

```
static void main() {
  String[] a = new1 String[] { "a1", "a2" }
  String[] b = new2 String[] { "b1", "b2" }
  List<String> l
  l = new3 List<String>()
  String v1 = a[*]
  l.append(v1)
  l = new4 List<String>()
  String v2 = b[*]
  l.append(v2)
}
class List<T> {
  T tail;
  void append(T c) {
    Link<T> k = new5 Link<T>()
    k.data = c
    Link<T> t = this.tail
    t.next = k
    this.tail = k
  }
}
```



Call Graph Construction

- Significant issue for OO languages like C++/Java
- Answers which methods a dynamically-dispatching call site can invoke at run-time
- Problem is undecidable
  - No exact (i.e. both sound and complete) solution
- But many conservative (i.e. sound) approximate solutions exist
  - Find which methods each dynamically-dispatching call site *may* invoke
  - All incomplete but differ in precision (i.e. false-positive rate)

Call Graph Construction (contd.)

- Two variants
  - in advance: prior to pointer analysis
  - on-the-fly: mutually recursively with pointer analysis
- Context insensitive algorithms
  - CHA: Class Hierarchy Analysis
    - Use class hierarchy alone to compute which methods each dynamically-dispatching call site may invoke
  - RTA: Rapid Type Analysis
    - Restrict CHA to classes instantiated in reachable object allocation sites
  - 0-CFA
    - Use pointer analysis to compute object allocation sites "this" argument of dynamically-dispatching call site may point to
- Context-sensitive algorithms
  - k-CFA
  - k-object-sensitive

Compositionality

- Whole-program
  - Cannot analyze open programs (e.g. libraries)
  - Predominant kind of pointer analysis
- Compositional/modular
  - Can analyze program fragments
    - Missing callers (does not need "harness")
    - Missing callees (does not need "stubs")
  - Solution is parameterized to accommodate unknown facts from the missing parts
  - Solution is instantiated to yield less parameterized (or fully instantiated) solution when missing parts are encountered
  - Parameterization harder in presence of dynamic dispatching
    - Existing approaches rely on call graph computed by a whole-program analysis but can be highly imprecise
  - Open problem

Adaptivity

- Non-adaptive
  - Computes exhaustive solution of fixed precision regardless of client
- Demand-driven
  - Computes partial solution, depending upon a query from a client, but of fixed precision
- Client-driven
  - Computes exhaustive solution but can use different precision in different parts of the solution, depending upon client
- Iterative/Refinement-based
  - Starts with an imprecise solution and refines it in successive iterations depending upon client