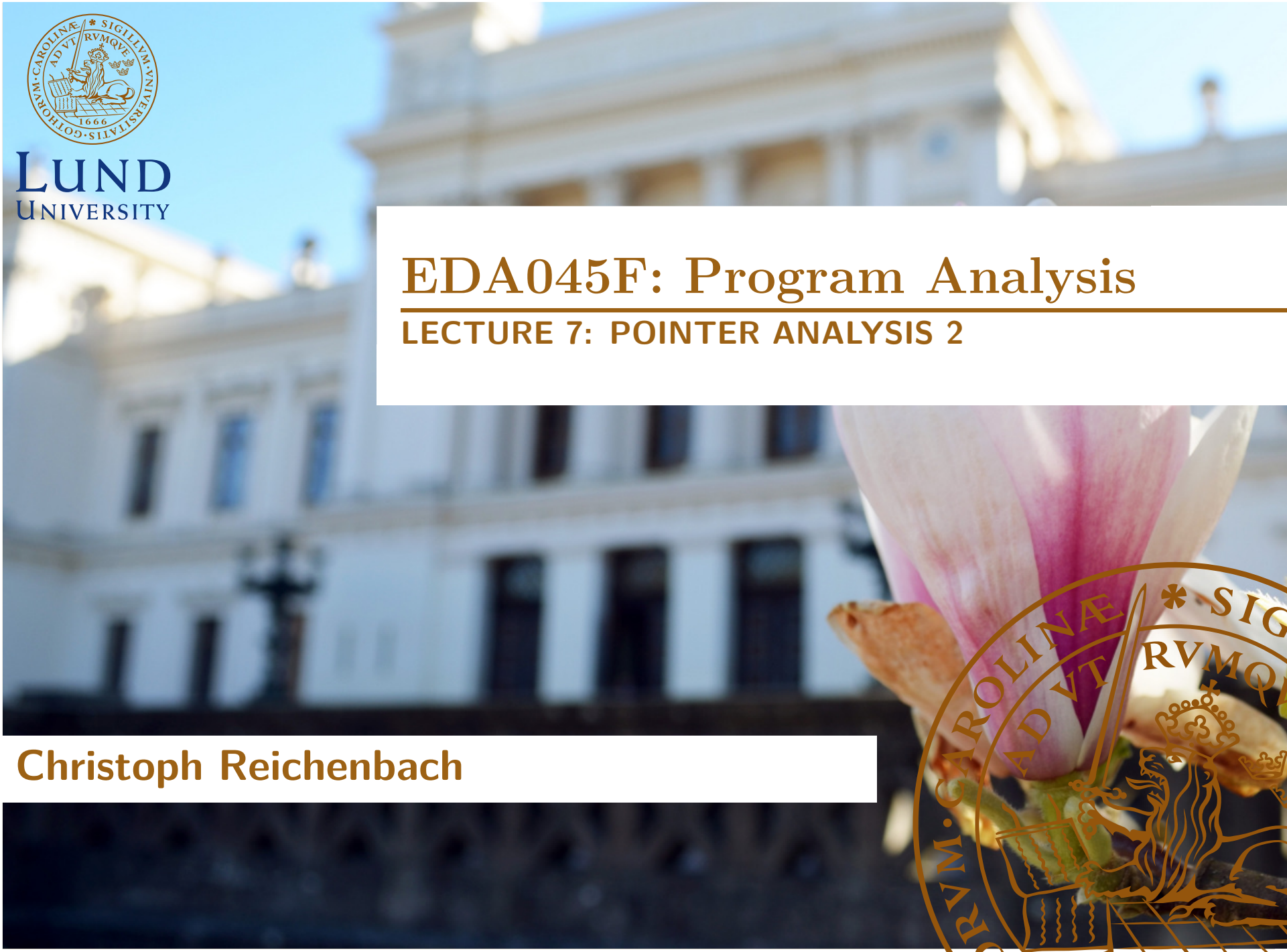


LUND UNIVERSITY

EDA045F: Program Analysis

LECTURE 7: POINTER ANALYSIS 2

Christoph Reichenbach

In the last lecture...

- Datalog
- Soufflé
- Doop

Corrections and Announcements

- Steensgard & Andersson (correct after all!)
- Aggregation vs. Stratification
  - No *need* to stratify aggregation (Soufflé doesn't)
  - Can introduce nontermination
- Final Exam
  - 11 January, 13:00–18:00, here in E:2116

Points-to-Analysis Sensitivities

- Points-to analysis is nondistributive
  - No easy route to precise interprocedural analysis
  - No known effective procedure summary representation
- We still want non-distributive analyses to be precise
  - Example: out-of-bounds checking in method-of-interest `copy()` needs size of array (assumption: we need array allocation site)
  - Approach: repeat analysis on same code for multiple *contexts*
    - no bounds violation in copy at $\mathcal{C}_0$
    - bounds violation in copy at $\mathcal{C}_1 \Leftarrow \mathcal{A}_3$
    - bounds violation in copy at $\mathcal{C}_2 \Leftarrow \mathcal{C}_2$

```
array0 = { }           // A0
array3 = { 0, 1, 2 } // A1
c0 = new Copier(array3) // A2
c1 = new Copier(array0) // A3
c0.copy(array3) // C0
c1.copy(array3) // C1
c0.copy(array0) // C2
```

```
class Copier {
  Copier(int[] s) {
    this.src = s
  }
  copy(int[] dest) {
    dest[0] = this.src[0]
} }
```

k-call-site Sensitivity

- Call-site sensitivity (Doop terminology; traditionally called *context-sensitivity*) analyses method once per call site
- Can determine that $\mathcal{C}_0$ is safe, $\mathcal{C}_1$ is unsafe
  - Analyses `get0` twice: Two different contexts $\mathcal{C}_0$ and $\mathcal{C}_1$
- Simple call-site sensitivity *cannot* distinguish $\mathcal{C}_2$ and $\mathcal{C}_3$
  - Will only analyse `get0` once, for context $\mathcal{C}_4$
- 2-call-site sensitivity: extend context to *caller's caller*
  - Contexts: $\langle \mathcal{C}_2, \mathcal{C}_4 \rangle$ and $\langle \mathcal{C}_3, \mathcal{C}_4 \rangle$
- Need 3-call-site sensitivity etc. for deeper calls

```
array0 = {}
v = get0({ 0, 1 }) // C0
v = get0(array0)   // C1
v = f({ 0, 1 })    // C2
v = f(array0)      // C3
v = g({ 0, 1 })    // C5
```

```
int get0(int[] array) {
  return array[0] }
int f(int[] array) {
  return get0(array) // C4 }
int g(int[] array) {
  return f(array) // C6 }
```

Object Sensitivity

- Object-sensitivity uses as context the method *receiver objects*
- Usually represented by allocation sites:

```
obj0 = new Obj() // A0
obj1 = new Obj() // A1
obj0.m(0) // C0
obj0.m(1) // C1
obj1.m(2) // C2
```

- Method-of-interest `m()` is analysed twice: for $\mathcal{A}_0$ and $\mathcal{A}_1$
- Call sites $\mathcal{C}_0$ and $\mathcal{C}_1$ use same context
- k-Object sensitivity is slighly more complex:
  - Variant 1: *Plain k-Object Sensitivity*
    (common definition in literature and used in Soot)
  - Variant 2: *Full k-Object Sensitivity*
    (original definition and used in Doop)

Plain Object Sensitivity

- 1-object sensitivity cannot distinguish the different invocations $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ to `a.invoke()` at $\mathcal{C}_0$
- *Plain 2-object sentitivity*:
  - Adds the *receiver object of the method that invoked the method-of-interest* to the context
  - Contexts: $\langle \mathcal{A}_3, \mathcal{A}_0 \rangle$ and $\langle \mathcal{A}_4, \mathcal{A}_0 \rangle$
  - Treats $\mathcal{C}_1$ and $\mathcal{C}_2$ the same
  - Distinguishes them from $\mathcal{C}_3$

```
o1 = new Owner() // A1
o2 = new Owner() // A2
c3 = new Caller() // A3
c4 = new Caller() // A4
c3.call(o1.a) // C1
c3.call(o2.a) // C2
c4.call(o1.a) // C3
```

```
class Owner {
  Owner() {
    this.a = new A() // A0
} }
class Caller {
  call(A a) {
    a.invoke() // C0
} }
```

Full Object Sensitivity

- *Full 2-object sentitivity*:
  - Adds the *receiver object of the method that allocated each object*
  - Distinguishes objects allocated at the same site if the objects that executed the allocation differ
  - Contexts: $\langle \mathcal{A}_1, \mathcal{A}_0 \rangle$ and $\langle \mathcal{A}_2, \mathcal{A}_0 \rangle$
  - Treats $\mathcal{C}_1$ and $\mathcal{C}_3$ the same
  - Distinguishes them from $\mathcal{C}_2$

```
o1 = new Owner() // A1
o2 = new Owner() // A2
c3 = new Caller() // A3
c4 = new Caller() // A4
c3.call(o1.a) // C1
c3.call(o2.a) // C2
c4.call(o1.a) // C3
```

```
class Owner {
  Owner() {
    this.a = new A() // A0
} }
class Caller {
  call(A a) {
    a.invoke() // C0
} }
```

Type-Sensitivity

```
class Main {
  main() {
    k1 = new K()   // A4
    k2 = new K()   // A5
    a = k1.make()  // C0
    b = new B()    // A6

    k1.a.invoke()  // C1
    k2.a.invoke()  // C2
    a.invoke()     // C3
    b.a.invoke()   // C4
```

```
class A {
  invoke() {
    // method of interest
} }

class K {
  K() {
    this.a = new A() // A1
  }
  A make() {
    return new A() // A2
} }

class B {
  B() {
    this.a = new A() // A3
} }
```

- Full 2-object sensitivity:
  - $\mathcal{C}_1$: $\langle \mathcal{A}_4, \mathcal{A}_1 \rangle$
  - $\mathcal{C}_2$: $\langle \mathcal{A}_5, \mathcal{A}_1 \rangle$
  - $\mathcal{C}_3$: $\langle \mathcal{A}_4, \mathcal{A}_2 \rangle$
  - $\mathcal{C}_4$: $\langle \mathcal{A}_6, \mathcal{A}_3 \rangle$
- 1-type sensty.+1-object sensty.:
  - $\mathcal{C}_1$: $\langle \textbf{Main}, \mathcal{A}_0 \rangle$
  - $\mathcal{C}_2$: $\langle \textbf{Main}, \mathcal{A}_0 \rangle$
  - $\mathcal{C}_3$: $\langle \textbf{Main}, \mathcal{A}_0 \rangle$
  - $\mathcal{C}_4$: $\langle \textbf{Main}, \mathcal{A}_0 \rangle$

Type sensitivity:

- Based on full type sensitivity
- Merge call sites in same class definition
- Represent by class (less precise)

Heap Analysis Precision

```
class Matrix {
  mult(...)  {...}
}
```

Contexts for analysing mult?

```
class Scene {
  Scene() {
    this.house = new VisibleObj(...) // A3
    this.tree = new VisibleObj(...)  // A4
  }
  drawAll(Camera c) {
    this.house.draw(c) // C2
    this.tree.draw(c)  // C3
} }
```

```
class Camera {
  Camera() {
    this.v = new Matrix() // A7
} }
```

```
main() {
  scene1 = new Scene()    // A0
  scene2 = new Scene()    // A1
  camera = new Camera()   // A2
  scene1.drawAll(camera) // C0
  scene2.drawAll(camera) // C1
}
```

```
class VisibleObj {
  VisibleObj() {
    this.pos = new Matrix()     // A5
    this.colour = new Matrix() // A6
  }
  draw(Camera c) {
    p = this.pos.mult(c.v)  // C4
    // draw!
} }
```

1-CS	callsite sensitive
2-CS	2-callsite sensitive
1-OS	object sensitive
2-P-OS	plain object-sensitive (with call-site allocator)
2-F-OS	full object-sensitive (with alloc-site allocator)
1-TS+1-OS	object sensitive with type sensitivity for alloc-site allocator

Heap Analysis Precision

```
class Matrix {
  mult(...)  {...}
}
```

Contexts for analysing mult?

```
class Scene {
  Scene() {
    this.house = new VisibleObj(...) // A3
    this.tree = new VisibleObj(...)  // A4
  }
  drawAll(Camera c) {
    this.house.draw(c) // C2
    this.tree.draw(c)  // C3
} }
```

```
class Camera {
  Camera() {
    this.v = new Matrix() // A7
} }
```

```
main() {
  scene1 = new Scene()    // A0
  scene2 = new Scene()    // A1
  camera = new Camera()   // A2
  scene1.drawAll(camera) // C0
  scene2.drawAll(camera) // C1
}
```

```
class VisibleObj {
  VisibleObj() {
    this.pos = new Matrix()     // A5
    this.colour = new Matrix() // A6
  }
  draw(Camera c) {
    p = this.pos.mult(c.v)  // C4
    // draw!
} }
```

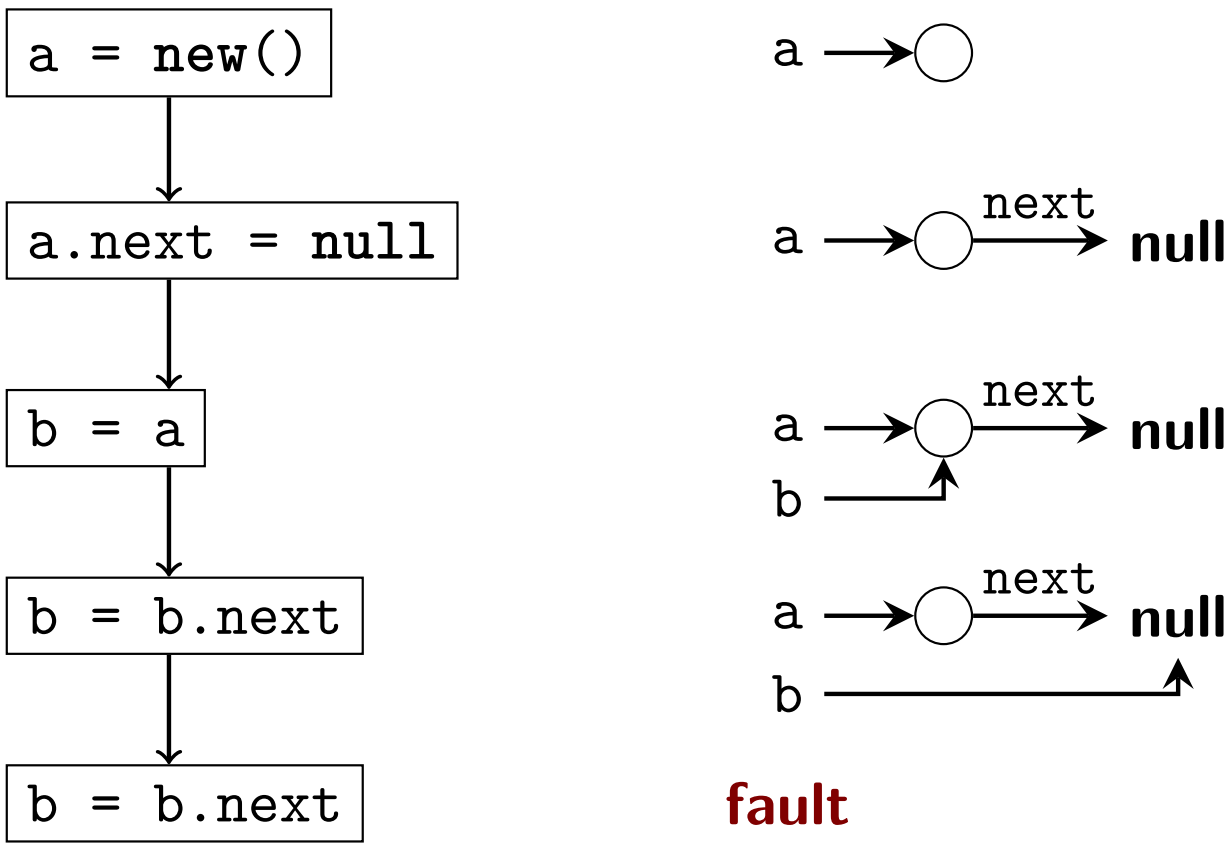
1-CS	$\mathcal{C}_4$
2-CS	$\langle \mathcal{C}_2, \mathcal{C}_4 \rangle$, $\langle \mathcal{C}_3, \mathcal{C}_4 \rangle$
1-OS	$\mathcal{A}_5$
2-P-OS	$\langle \mathcal{A}_0, \mathcal{A}_5 \rangle$, $\langle \mathcal{A}_1, \mathcal{A}_5 \rangle$
2-F-OS	$\langle \mathcal{A}_3, \mathcal{A}_5 \rangle$, $\langle \mathcal{A}_4, \mathcal{A}_5 \rangle$
1-TS+1-OS	$\langle$ **Scene**, $\mathcal{A}_5 \rangle$

Summary

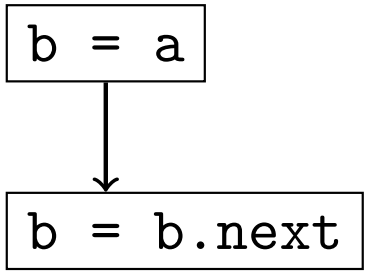
- Analysis sensitivities allow us to analyse methods more precisely
  - Multiple analyses of same method in different *contexts*
  - Context provides additional information (args, globals, heap)
  - With procedure summaries (cf. IFDS / IDE): no repeat analysis necessary, but only for distributive frameworks
- **Call site sensitivity** (traditionally called *context sensitivity*) uses call sites as context
- **Object sensitivity** uses abstract receiver objects of method calls (typically identified by call site) as context (requires pointer analysis)
- k-**call site sensitivity** for $k > 1$ uses call sites, parent call sites, grandparent call sites etc. as context
- **Plain** k-**object sensitivity** for $k > 1$ uses abstract receiver objects of the ancestor method call(s) that led to method-of-interest as context
- **Full** k-**object sensitivity** uses abstract receiver objects of the ancestor method call(s) that led to allocation of receiver object as context
- **Type sensitivity** abstracts over full k-object sensitivity by merging call sites from
- These approaches can be layered and combined
- Worst case analysis cost exponential over k

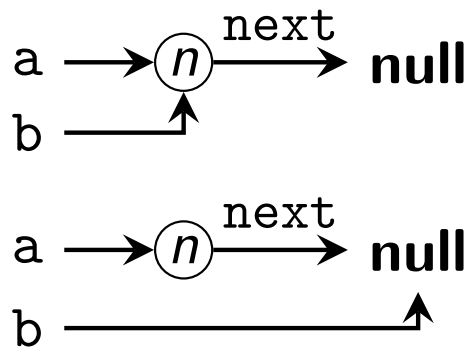
Flow-Sensitive Points-To Analysis

- Points-to analysis in practice: flow-insensitive (Steensgard, Andersen)
- Flow-sensitive analysis possible on small modules



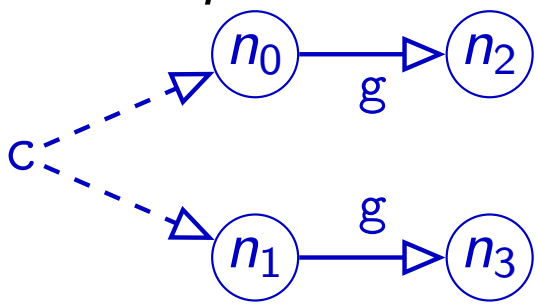
Strong and Weak Updates





- Flow-sensitive program analysis enables *strong updates*:
  - Remove information that is overwritten by update

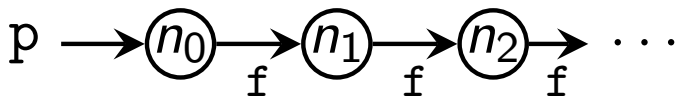
    b $\not\rightarrow$ n after update
- *Weak updates* still needed when ambiguous:



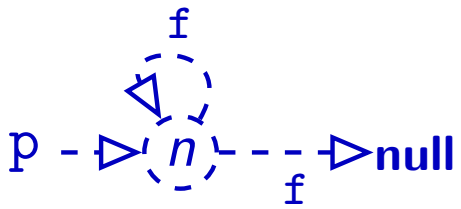
  - Consider `c.g = null`
  - Cannot be strong update (unclear which fact to delete)
- Flow-insensitive points-to analyses only use weak updates

Abstraction and Focus

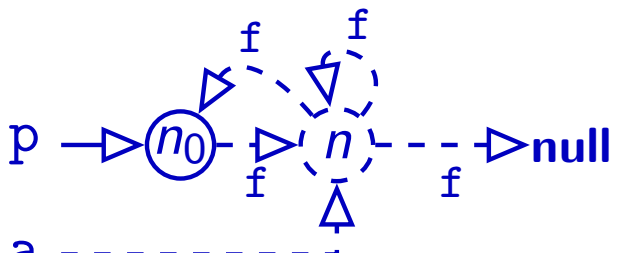
```
p = mklist()
```

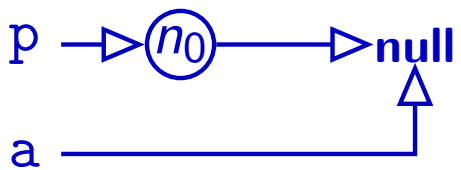


Abstraction




```
a = p.f
```



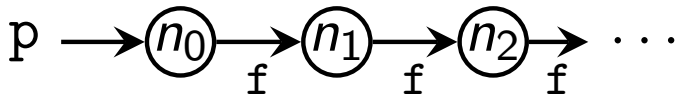


Focus

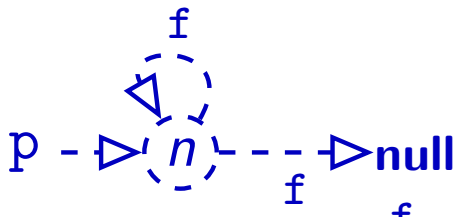
Two possible concretisations

Abstraction and Focus

```
p = mklist()
```

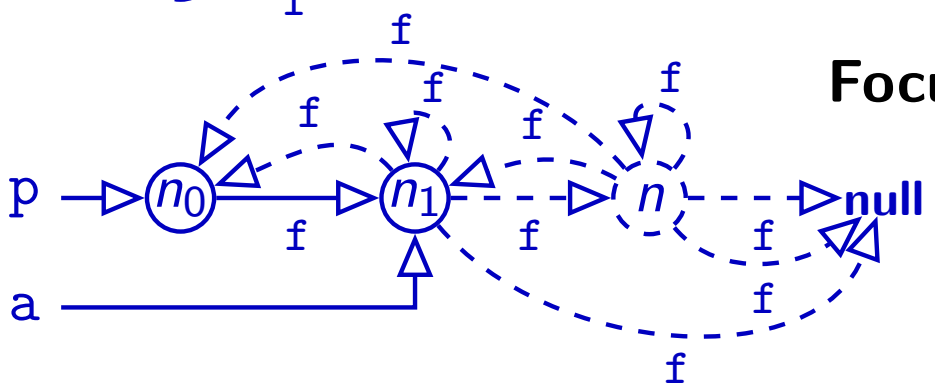


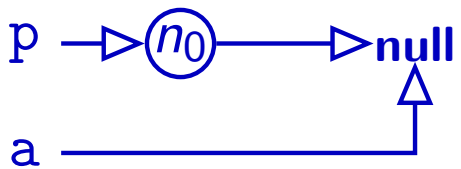
Abstraction




```
a = p.f
```

Focus





Shape Analysis

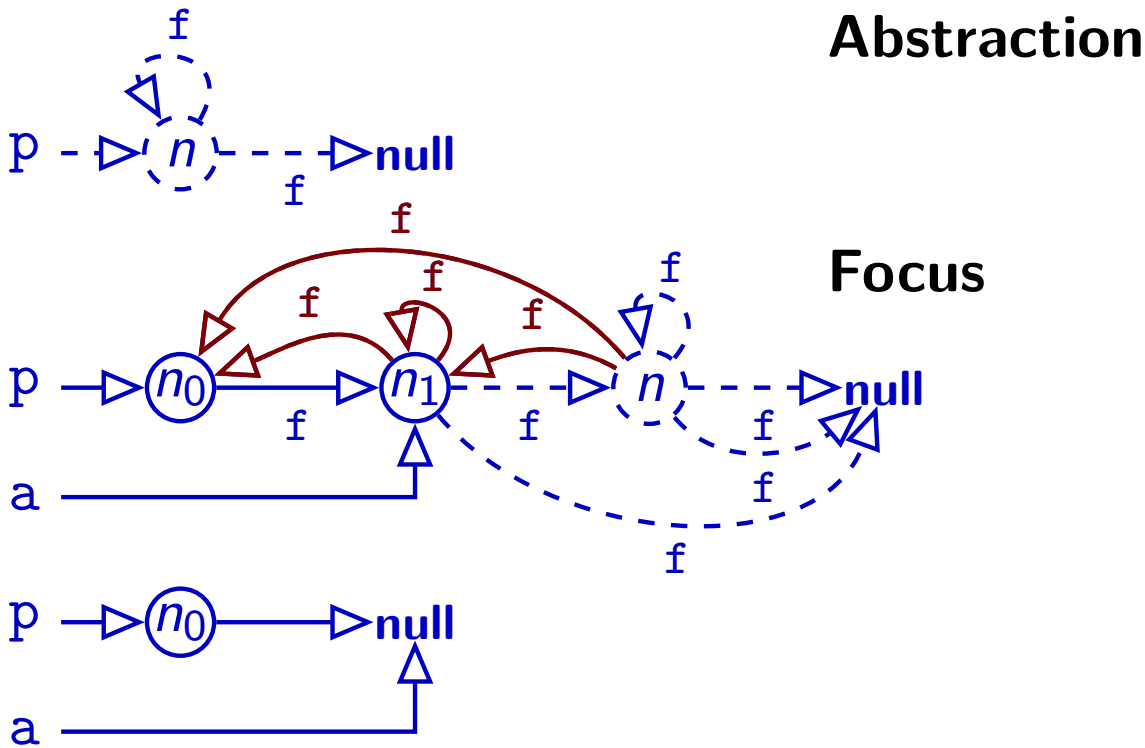
- *Shape analysis* describes more abstract properties of heap nodes
- Examples:
  - *may-be-null*(n): n may be **null**
  - *may-share*(n_1, n_2): $n_1 \rightarrow n \leftarrow n_2$ is possible (for some n)
  - *must-share*(n_1, n_2): $n_1 \rightarrow n \leftarrow n_2$ is certain (for some n)
  - *reachable*(n, m): $n \rightarrow^\star m$
  - *disjoint*(n_1, n_2): n_1 and n_2 point to disjoint structures: When *reachable*(n_1, m) for some m, then $\neg$*reachable*(n_2, m), and if *reachable*(n_2, m), then $\neg$*reachable*(n_1, m).
  - *list*(n, f): Node n is part of a singly linked list along field f:

$$n \xrightarrow{f}{}^\star n' \text{ implies that } n' \not\xrightarrow{f} n \text{ (for any } n')$$

Keeping the Graph in Shape

```
p = mklist()

a = p.f
```



$list(\ell, f)$: $\ell \xrightarrow{f}{}^{\star} \ell'$ implies that $\ell' \not\xrightarrow{f} \ell$ (for any ℓ')

If we know $list(n)$, we also know $list(n_0)$, $list(n_1)$

$list(n)$ allows us to elide impossible edges

Summary

- Flow-sensitive points-to analysis is possible
- **Weak updates** add new points-to relationship options
- **Strong updates** add but also remove points-to relationship options
  - More precise than weak updates
  - Only possible if updated pointer is unambiguous
- **Abstraction** operations introduce summary nodes or merge existing summary nodes with other nodes
- **Focus** operations turn summary nodes/edges into non-summary nodes/edges
  - Must usually consider multiple options to remain sound
- **Shape analysis** analyses graphs for higher-level properties such as graph-disjointness, tree-structuredness, list-structuredness etc.
  - Can improve precision of **Focus** operations

Utility of points-to analysis

- Pointer analysis invaluable for building call graphs
- Also useful for:
  - Optimisation (aliasing information)
  - Finding **memory errors**

Manual Memory Management

```
free(p);
```

- *Manually memory-managed* languages require manual *deallocation* of memory
- Main languages in use, with their deallocators:
  - **C**: `free()`
  - **C++**: `delete`, `delete[]`, (`free()` for compatibility)
- Source of memory errors: using mismatching deallocator

Deallocation Errors

C++

```
A* obj = new A(...);
delete obj;
obj->f(42); // use after deallocation
delete obj; // double deallocation
```

- *double `free`* or *double deallocation*:
  request deallocation of pointer that has already been deallocated
- *use after deallocation* or *use after `free`*:
  Access pointer that has already been deallocated

Stale Pointer Errors

C

```c
int* p = malloc(...);
free(p);
*p = 23; // write to dangling pointer

int* f(int v) {
  int k = v * v;
  return &k;
}
p = f(); // stale stack pointer
```

- *Dangling pointers* are pointers to deallocated memory regions
- Deallocation may happen:
  - *explicitly* on the heap (`free`, ...)
  - *implicitly* on the stack (**return**)

Uninitialised Memory errors

C++

```
int f(int v) {
  int random;
  return v + random;
}
```

- In C/C++: Contents of variable `random` undefined
- Analogous with `malloc` / `realloc`: Memory contents not guaranteed to be zeroed

Out-Of-Bounds Errors

C

```c
int f(int v) {
  int a = 0;
  int k[4] = { 0, 0, 0, 0 };
  int b = 0;
  k[v] = 1;
  ...
}
```

- If `v=-1` or `v=4`:
  - May overwrite `a` or `b`
- Other values may overwrite return address
  - *Buffer overrun*: common security vulnerability

Pointer Type Errors

C

```
class A {...};

std::string s = "foo";
void* sp = &s;
A* a = static_cast<A*>(sp);
a->method();
```

- 'Type conversion' in C/C++ often not typesafe
- Can re-interpret memory contents as type
  - May be sensible (loading memory image from disk)
  - May be nonsensical (cf. example)

Memory Leak

C

```c
int f(int fd) {
  void* v = malloc(1024);
  int len = read(fd, v, 1024);
  return v[(len < 1)?  0 :  len-1];
}
```

Java

```java
class A {
  private static A g = null;
  private A n;
  public A() {
    this.n = g;
    g = this;
} }
```

- Memory goes through stages:
  1. Allocated, initialised
  2. Utilised
  3. Not utilised
  4. Deallocated
- Memory leak: stage 3 ‘too long’

Summary

Memory management:	Unsafe manual	Safe manual	Automatic
Null pointer dereference	fault	fault	fault
Deallocation error	corruption	fault	—
Stale pointer error	corruption	fault	—
Uninitialised memory error	corruption	fault	fault
Out-of-bounds error	corruption	fault	fault
Pointer type error	corruption	fault	fault

- Memory corruption:
  - Hard-to-find errors (may not manifest, manifest much later)
  - Sometimes exploitable for (remote) attacks
- **Memory leak**: only *partly* mitigated by automatic memory management

Review

- k-call-site-sensitive (points-to) analysis
- Plain and full k-object-sensitive (points-to) analysis
- Type-sensitive (points-to) analysis
- Shape analysis (briefly)
- Memory errors in C/C++

To be continued...

- Dynamic Program Analysis
- Program Instrumentation