Dimensions in Pointer Analysis

CS6013: Modern Compilers - Theory and Practice

Manas Thakur

PACE Lab, IIT Madras

April 8th, 2016

Pointer Analysis

- Establishes which pointers (or heap references) can point to which objects (or storage locations).
- Applications: Alias analysis, shape analysis, escape analysis, etc.

Pointer Analysis

- Establishes which pointers (or heap references) can point to which objects (or storage locations).
- Applications: Alias analysis, shape analysis, escape analysis, etc.

```
class A {...}
class B extends A {...}
class C {
   public void foo() {
      A a1, a2, a3;
      a1 = new A();      //l1
      if(*)
         a2 = new A();   //l2
      else
         a2 = new B();   //l3
      a3 = a1;
   }
}
```

Pointer Analysis

- Establishes which pointers (or heap references) can point to which objects (or storage locations).
- Applications: Alias analysis, shape analysis, escape analysis, etc.

```
class A {...}
class B extends A {...}
class C {
   public void foo() {
      A a1, a2, a3;
      a1 = new A();      //l1
      if(*)
         a2 = new A();   //l2
      else
         a2 = new B();   //l3
      a3 = a1;
   }
}
```

Points-to sets:

- a1 $\rightarrow$ {l1}
- a2 $\rightarrow$ {l2, l3}
- a3 $\rightarrow$ {l1}

Overview

1 Pointer Analysis

2 Analysis Dimensions
- Flow-sensitivity
- Field-sensitivity
- Interprocedural analysis
- Context-sensitivity

3 Application

4 Conclusion

Flow-sensitivity

Flow-sensitive: Maintain information at each point of the program.

```
class C {
   public void foo() {
      A a1, a2, a3;
1:    a1 = new A();      //l1
2:    if(*)
3:       a2 = new A();   //l2
4:    else
5:       a2 = new A();   //l3
6:    a1 = a2;
7:    a3 = a2;
8:    ...
   }
}
```

Flow-sensitivity

Flow-sensitive: Maintain information at each point of the program.

```
class C {
   public void foo() {
      A a1, a2, a3;
      a1 = new A();      //l1
      if(*)
         a2 = new A();   //l2
      else
         a2 = new A();   //l3
      a1 = a2;
      a3 = a2;
      ...
   }
}
```

Flow-insensitive points-to sets:

- a1: {l1, l2, l3}
- a3: {l2, l3}

Flow-sensitivity

Flow-sensitive: Maintain information at each point of the program.

```
class C {
   public void foo() {
      A a1, a2, a3;
1:    a1 = new A();      //l1
2:    if(*)
3:       a2 = new A();   //l2
4:    else
5:       a2 = new A();   //l3
6:    a1 = a2;
7:    a3 = a2;
8:    ...
   }
}
```

Flow-insensitive points-to sets:

- a1: {l1, l2, l3}
- a3: {l2, l3}

Flow-sensitive points-to sets for:

- a1:
  - {} till line no. 1
  - {l1} from line nos. 1 to 6
  - {l2, l3} afterwards
- a3:
  - {} till line no. 7
  - {l2, l3} afterwards

Field-sensitivity

Field-sensitive: Maintain information separately for fields of an object.

```
class A {A f1; A f2;}
class C {
   public void foo() {
      A a1;
      a1 = new A();     //l1
      a1.f1 = new A(); //l2
      a1.f2 = new A(); //l3
   }
}
```

Field-sensitivity

Field-sensitive: Maintain information separately for fields of an object.

```
class A {A f1; A f2;}
class C {
   public void foo() {
      A a1;
      a1 = new A();     //l1
      a1.f1 = new A(); //l2
      a1.f2 = new A(); //l3
   }
}
```

Field-insensitive points-to sets:

- a1 → {l1, l2, l3}

Field-sensitivity

Field-sensitive: Maintain information separately for fields of an object.

```
class A {A f1; A f2;}
class C {
   public void foo() {
      A a1;
      a1 = new A();     //l1
      a1.f1 = new A();  //l2
      a1.f2 = new A();  //l3
   }
}
```

Field-insensitive points-to sets:

- a1 → {l1, l2, l3}

Field-sensitive points-to sets:

- a1 → {l1}
- a1.f1 → {l2}
- a1.f2 → {l3}

Intraprocedural vs Interprocedural analyses

Say, method `f0` calls method `f1`.

Intraprocedural analysis:

- Information computed for `f0` ignores the points-to results of `f1`.
- Conservative assumptions are made at call sites.

Intraprocedural vs Interprocedural analyses

Say, method `f0` calls method `f1`.

Intraprocedural analysis:

- Information computed for `f0` ignores the points-to results of `f1`.
- Conservative assumptions are made at call sites.

Interprocedural analysis:

- Information computed for `f0` considers the points-to results of `f1`.
- Requires a call-graph.

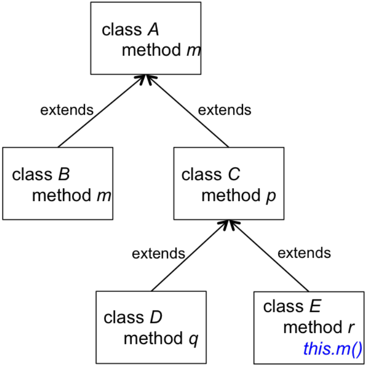
Call-Graph Construction

- A call-graph is needed to determine the possible callees at a call-site.
  - Offline as a pre-analysis.
  - On-the-fly using points-to results.

PACE

Call-Graph Construction

- A call-graph is needed to determine the possible callees at a call-site.
  - Offline as a pre-analysis.
  - On-the-fly using points-to results.
- Class Hierarchy Analysis (CHA)



CHA helps in determining that only one implementation of `m` can be called.

Context-sensitivity

Context-sensitive: Maintain different results for different contexts from which a method is called.

Context-sensitivity

Context-sensitive: Maintain different results for different contexts from which a method is called.

What is a context?

Context-sensitivity

Context-sensitive: Maintain different results for different contexts from which a method is called.

What is a context?

- Call-site-sensitivity
- Object-sensitivity

1-Call-site-sensitive

```
class A { fb()...}
class B extends A { fb()...}

class C {
   A a1;

   public void foo() {
      a1 = new A();    // l1
c1:   bar(a1);
      a1 = new B();    // l2
c2:   bar(a1);
   }

   public void bar(A p1) {
      p1.fb();
   }
}
```

1-Call-site-sensitive

```
class A { fb()...}
class B extends A { fb()...}

class C {
   A a1;

   public void foo() {
      a1 = new A();    // l1
c1:   bar(a1);
      a1 = new B();    // l2
c2:   bar(a1);
   }

   public void bar(A p1) {
      p1.fb();
   }
}
```

Context-insensitive:

- `a1` → {`l1, l2`}
- Both A's and B's `fb` can be called.

1-Call-site-sensitive

```
class A { fb()...}
class B extends A { fb()...}

class C {
   A a1;

   public void foo() {
      a1 = new A();   // l1
c1:   bar(a1);
      a1 = new B();   // l2
c2:   bar(a1);
   }

   public void bar(A p1) {
      p1.fb();
   }
}
```

Context-insensitive:

- a1 → {l1, l2}
- Both A's and B's fb can be called.

1-Call-site-sensitive:

- a1 → {l1}
  A's fb will be called.
- a1 → {l2}
  B's fb will be called.

2-Call-site-sensitive

```
class C {
   public void foo1() {
      bar();
   }

   public void foo2() {
      bar();
   }

   public void bar() {
      fb();
   }
}
```

2-Call-site-sensitive

```
class C {
   public void foo1() {
      bar();
   }

   public void foo2() {
      bar();
   }

   public void bar() {
      fb();
   }
}
```

1-Call-site-sensitive:

- 1 context for `fb`

2-Call-site-sensitive

```
class C {
   public void foo1() {
      bar();
   }

   public void foo2() {
      bar();
   }

   public void bar() {
      fb();
   }
}
```

1-Call-site-sensitive:

- 1 context for `fb`

2-Call-site-sensitive:

- 2 contexts for `fb`

1-Object-sensitive

Distinguish contexts based on the allocation site of the receiver.

1-Object-sensitive

Distinguish contexts based on the allocation site of the receiver.

```
main() {
    o1 = new A();
    o2 = new A();
    o1.bar();
    o2.foo();
    o2.bar();
    o2.foo();
}

foo() {
    ...
}

bar() {
    ...
}
```

1 context for `foo`; 2 contexts for `bar`

2-Object-sensitive

Distinguish contexts based on:

- Allocation site of receiver
- Allocation site of allocator of receiver

2-Object-sensitive

Distinguish contexts based on:

- Allocation site of receiver
- Allocation site of allocator of receiver

```
main() {
    o1 = new A();
    o1.foo();
    o1 = new A();
    o1.foo();
}

foo() {
    o2 = new A();
    o2.bar();
}

bar() {...}
```

1-Object-sensitive:

- 1 context for bar

2-Object-sensitive:

- 2 contexts for bar

Which context-sensitivity is better?

```
main() {
    o1 = new A();
    o1.foo();
    o1.foo();
}

foo() {
    o2 = new A();
    o2.bar();
}

bar() {...}
```

2-Object-sensitive:

- 1 context for bar

Which context-sensitivity is better?

```
main() {
    o1 = new A();
    o1.foo();
    o1.foo();
}

foo() {
    o2 = new A();
    o2.bar();
}

bar() {...}
```

2-Object-sensitive:

- 1 context for bar

2-Call-site-sensitive:

- 2 contexts for bar
- No change in precision

Which context-sensitivity is better?

```
main() {
    o1 = new A();
    o1.foo();
    o1.foo();
}

foo() {
    o2 = new A();
    o2.bar();
}

bar() {...}
```

2-Object-sensitive:

- 1 context for bar

2-Call-site-sensitive:

- 2 contexts for bar
- No change in precision

There is no thumb-rule for choosing the type of context-sensitivity; it depends on the application and the desired precision.

PACE

Overview

Escape Analysis

Definition

An object is said to *escape* from a method/thread if it can be accessed in another method/thread.

Escape Analysis

Definition

An object is said to *escape* from a method/thread if it can be accessed in another method/thread.

In Java, an object may escape the allocating method when:

- Passed as an argument to another method.
- Returned from the method.
- Accessible by a static (global) variable. (thread-escape)

Escape Analysis

Definition

An object is said to *escape* from a method/thread if it can be accessed in another method/thread.

In Java, an object may escape the allocating method when:

- Passed as an argument to another method.
- Returned from the method.
- Accessible by a static (global) variable. (thread-escape)

Escape analysis helps in:

- Stack allocation
- Synchronization elimination

Example

```
class A {...}
class C {
   static A global;
   public void foo() {
      A a1, a2, a3;
      a1 = new A();   //l1
      a2 = new A();   //l2
      a3 = new A();   //l3
      global = a2;
      a1.m2();
   }
}
```

Example

```
class A {...}
class C {
    static A global;
    public void foo() {
        A a1, a2, a3;
        a1 = new A();    //l1
        a2 = new A();    //l2
        a3 = new A();    //l3
        global = a2;
        a1.m2();
    }
}
```

Points-to sets :

- a1 $\rightarrow$ {l1}
- a2 $\rightarrow$ {l2}
- a3 $\rightarrow$ {l3}
- global $\rightarrow$ {l2}

Example

```
class A {...}
class C {
   static A global;
   public void foo() {
      A a1, a2, a3;
      a1 = new A();   //l1
      a2 = new A();   //l2
      a3 = new A();   //l3
      global = a2;
      a1.m2();
   }
}
```

Points-to sets :

- a1 → {l1}
- a2 → {l2}
- a3 → {l3}
- global → {l2}

Escape analysis results :

- l1 escapes foo()
- l2 escapes foo() as well as the thread
- l3 does not escape

Exercise: Flow-sensitivity

```
class C {
   static A global;

   public void foo() {
p1:    A a1 = new A();   //l1
p2:    A a2 = new A();   //l2
p3:    global = a1;
       ...
   }

}
```

Exercise: Flow-sensitivity

```
class C {
   static A global;

   public void foo() {
p1:    A a1 = new A();  //l1
p2:    A a2 = new A();  //l2
p3:    global = a1;
       ...
   }

}
```

Flow-insensitive:

- `l1` escapes the thread.

Flow-sensitive:

- `l1` escapes the thread after the point p3.

Exercise: Field-sensitivity

```
class C {
   static A global;

   public void foo() {
      A a1 = new A();   //l1
      a1.f = new A();   //l2
      A a2 = new A();   //l3
      global = a1.f;
   }

}
```

Exercise: Field-sensitivity

```
class C {
   static A global;

   public void foo() {
      A a1 = new A();   //l1
      a1.f = new A();   //l2
      A a2 = new A();   //l3
      global = a1.f;
   }

}
```

Field-insensitive:

- l1 and l2 escape the thread.

Field-sensitive:

- l2 escapes the thread.

Exercise: Context-sensitivity

```
class A {
   A f;
   public void bar() {
      A b3 = new A();  //l4
      this.f = b3;
   }
}

class C {
   static A global;
   public void foo() {
      A a1 = new A();  //l1
      a1.bar();        //c1
      global = a1;
      a1.bar();        //c2
   }
}
```

Exercise: Context-sensitivity

```
class A {
   A f;
   public void bar() {
      A b3 = new A();  //l4
      this.f = b3;
   }
}

class C {
   static A global;
   public void foo() {
      A a1 = new A();  //l1
      a1.bar();        //c1
      global = a1;
      a1.bar();        //c2
   }
}
```

Context-insensitive (bar):

- l4 escapes the thread.

Context-sensitive (bar):

- l4 does not escape the thread from the call at c1.
- l4 escapes the thread from the call at c2.

Conclusion

- There are various dimensions along which the precision of a pointer analysis can be improved.
- Usually there is a tradeoff between the precision and the efficiency of an analysis.
- The dimensions that we discussed can be applied to improve the precision of other program analyses as well.

Conclusion

- There are various dimensions along which the precision of a pointer analysis can be improved.
- Usually there is a tradeoff between the precision and the efficiency of an analysis.
- The dimensions that we discussed can be applied to improve the precision of other program analyses as well.

Thank You.

Pointers for the enthusiast

- Vivien F. and Rinard M., *Incrementalized Pointer and Escape Analysis*, PLDI 2001.
- Hardekopf B. and Lin C., *The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code*, PLDI 2007.
- Whaley J. and Lam Monica S., *Cloning-based Context-sensitive Pointer Alias Analysis using Binary Decision Diagrams*, PLDI 2004.

- Slides: https://manasthakur.github.io/docs/cs6013-dpa.pdf