Interprocedural Analysis

Aleksandra Biresev

s6albire@cs.uni-bonn.de

Interprocedural Analysis

- An interprocedural analysis operates across an entire program, flowing information from call sites to their callees and vice versa

```
class Superfun {
    String m(String param1) {
        return "This is super fun!“+param1;
    }
}
class Fun extends Superfun {
    String m(String param1) {
        return "This is fun!“+param1;
    }
}
void main() {
     String arg1=“ Really!”;
     String s;
     Superfun  sf = new Superfun();
     Fun f = new Fun();
     sf = f
     s = sf.m(arg1);
}
```

Call site of m(String)

Interprocedural Analysis needs a Call Graph

Sample Program

```
class Superfun {
  String m(){
        return "This is super fun!";
    }
 }
class Fun extends Superfun {
  String m(){
        return "This is fun!";
    }
 }
void main() {
     Superfun  sf = new Superfun();
     Fun  f = new Fun();
     sf=f;
     … = sf.m();
 }
```

Call Graph

- Representation of program's calling structure
- Set of nodes and edges such that:
  - There is one node for each procedure in the program
  - There is one node for each call site
  - If call site c may call procedure p, then there is an edge from the node for c to the node for p

Interprocedural Analysis needs a Call Graph

Sample Program

```
class Superfun {
  String m() {
        return "This is super fun!";
    }
 }
class Fun extends Superfun {
  String m() {
        return "This is fun!";
    }
 }
void main() {
    Superfun  sf = new Superfun();
    Fun  f = new Fun();
    sf=f;
    … = sf.m();
 }
```

Statically bound calls

- Call target of each invocation can be determined statically
- Each call site has an edge to exactly one procedure in the call graph
- Examples:
  - All calls in C, Pascal, …
  - Calls of "static" methods in Java

Interprocedural Analysis needs a Call Graph

Sample Program

```
class Superfun {
  String m(){
        return "This is super fun!";
    }
 }
class Fun extends Superfun {
  String m(){
        return "This is fun!";
    }
 }
void main() {
     Superfun  sf = new Superfun();
     Fun  f = new Fun();
     sf=f;
     … = sf.m();
 }
```

receiver expression

Dynamically bound calls

- Normal case in Java
- We need to know the dynamic type of the message receiver before we can determine which method is invoked
- Dynamic type = class form which the receiver was instantiated
- How to approximate this information statically?

Interprocedural Analysis needs a Call Graph

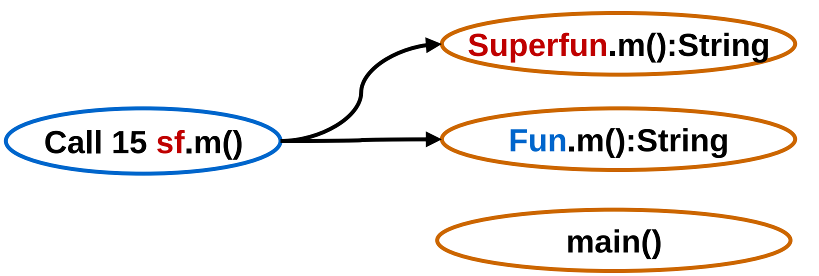
Sample Program

```
1.   class Superfun {
2.     String m() {
3.          return "This is super fun!";
4.     }
5.  }
6.   class Fun extends Superfun {
7.     String m() {
8.          return "This is fun!";
9.     }
10. }
11. void main() {
12.      Superfun  sf = new Superfun();
13.      Fun  f = new Fun();
14.      sf=f;
15.      … = sf.m();
16. }
```

receiver expression

Call Graph based on static type information

- Regard all methods in
  - static type of receiver
  - … and in each subtype
- Static Type
  - Type declared in the program
- Example
  - sf has static type Superfun
- In our example we get



Why Interprocedural Analysis?

Using interprocedural analysis a number of important compiler problems can be solved:

- Resolution of Virtual Method Invocations (later in this talk)
  - More precise Call Graph construction via PTA
- Pointer Alias Analysis
  - Could two variables eventually point to the same object?
- Parallelization
  - Will two array references never point to the same array?
- Detection of Software Errors and Vulnerabilities (SQL Injection)
  - Can user input become executable code?

➢ The basis of any interprocedural analysis is „Points-to Analysis“

Outline

- ✓ Interprocedural Analysis
- ✓ Call Graphs
- ➢ Variants of Interprocedural Analysis
- ➢ Approaches to Context-Sensitive Interprocedural Analysis
- ➢ Points-To-Analysis (PTA)
- ➢ Logic programming as an implementation framework

Variants of Interprocedural Analysis

Context-insensitive

Context-sensitive

Cloning-based

Inline-based

Summary-based

Interprocedural Analysis ▶ Context-insensitivity

```
void p() {
    a = 2;
    b = id(a);
}
void q() {
    c = 3;
    d = id(c);
}
void r() {
    e = 3;
    f = id(e);
}
int id(int x) {
    return x;
}
```

- We do not care about who called the procedure that we currently analyse
  - E.g. for id(int)
- Simple but imprecise
  - All inputs (values of a, c, e) are merged
  - Set of potential values for x = {2, 3}
  - Imprecision propagates to call sites
  - We can only discover {2, 3} as potential values for b, d and f
- Not so bad when program units are small (few assignments to any variable)
  - Example: Java code often consists of many small methods

Interprocedural Analysis ▶ Context-sensitivity

```
void p() {
    a = 2;
    b = id(a);
}
void q() {
    c = 3;
    d = id(c);
}
void r() {
    e = 3;
    f = id(e);
}
int id(int x) {
    return x;
}
```

- We do care about who called the procedure that we currently analyse
  - E.g. id(a) or id(c) or id(e)
- More precise
  - Inputs are propagated individually for each call
  - Results are returned only to the related call
  - We can discover b=2, d=3 or f=3.

Context-sensitivity ▸ Calling contexts

```
1.    void p() {
2.        a = 2;
3.        b = id(a);
4.    }
5.    void q() {
6.        c = 3;
7.        d = id(c);
8.    }
9.    void r() {
10.       e = 3;
11.       f = id(e);
12.  }
13.  int id(int x) {
14.      return add_id(x);
15.  }
16.  int add_id(int x){
17.      return x;
18.  }
```

- Context
  - Summary of the sequence of calls that are currently on the run-time stack
- Call string
  - Sequence of call sites for the calls on the stack
- Call site
  - Identified by its line number
- Example
  - Call sequence id(a) → add_id(x) summarized as call string (3,14)
  - Call strings on this slide: (3, 14), (7,14), (11,14)
  - Call strings on previous slide: (3), (7), (11)

Approaches to context-sensitive analysis ▶ Cloning-Based

```
void p(){
    a=2;
    b=id1(a);
}
void q(){
    c=3;
    d=id2(c);
}
void r(){
    e=3;
    f=id3(e);
}
```

Not cloned (not called)
Call clones of id(int)

Clones of id(int)
Call clones of add_id(int)

```
int id1(int x)  {
    return add_id1(x);
}
int id2(int x){
    return add_id2(x);
}
int id3(int x){
    return add_id3(x);
}
```

Clones of add_id(int)

```
int add_id1(int x){
    return x;
}
int add_id2(int x){
    return x;
}
int add_id3(int x){
    return x;
}
```

- Each procedure is cloned once for each relevant context
- Then context-insensitive analysis of the code resulting from cloning / inlining
- Problem
  - Exponentially many contexts in the worst case!
  - Exponentially many clones

Approaches to context-sensitive analysis ▶ Inlining-Based

```
void p(){
    a=2;
    b=a;      <-------
}
void q(){
    c=3;
    d=c;      <-------
}
void r(){
    e=3;
    f=e;      <-------
}
```

Inlined body of id(a) after expanding id(int) by inlining add_int()

Inlined body of id(c) after expanding id(int) by inlining add_int()

Inlined body of id(e) after expanding id(int) by inlining add_int()

```
int id(int x) {
    return x; <-----
}
```

Inlined body of add_id(x)

```
int add_id(int y){
    return y;
}
```

- Rather than physically cloning, recursively inline body of called procedure at the call
  - Simplifications possible
- In reality, we do not need to clone the code, neither to inline
  - See talk of Saad

Approaches to context-sensitive analysis ▶ Summary-Based

```
 void p() {
     a = 2;
     b = id(a);
 }
 void q() {
     c = 3;
     d = id(c);
 }
 void r() {
     e = 3;
     f = id(e);
}
int id(int x) {
    return add_id(x);
}
int add_id(int x){
    return x;
}
```

a) Original

Σ

```
 void p() {
     a=2;
     b=id_a2();
 }
 void q() {
     c=3;
     d=id_a3();
 }
 void r() {
     e=3;
     f=id_a3();
}
int id_a2() {
    return 2;
}
int id2_a3() {
    return 3;
}
```

b) Summaries for 2 different parameter values of id()

- Each procedure is represented by a concise description ("summary") that encapsulates some observable behavior of the procedure
- The primary purpose of the summary is to avoid reanalyzing a procedure's body at every call site
- The analysis consists of two parts:
  - A top-down phase that propagates caller information (parameter values) to compute results of the callees
  - A bottom-up phase that computes a “transfer function” to summarize the effect of a procedure

Points-to Analysis (PTA)

Also known as „Pointer Analysis“

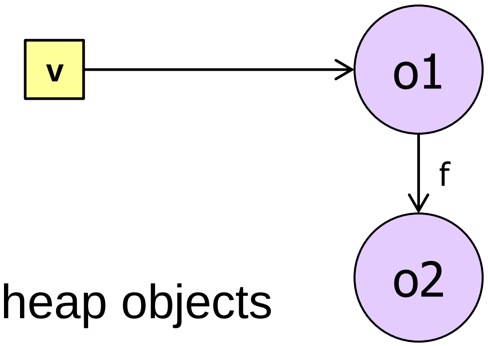
Pointer Analysis / Points-to Analysis (PTA)

- Special form of data flow analysis
  - Not interested in primitive values but only in object references / pointers

- Question: „To which objects can a variable refer?“

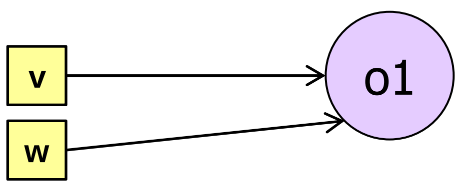
- PTA is at the heart of any interprocedural analysis
  - Because it improves the precision of the call graph

Pointer Analysis ▶ Objects flow to Variables



- Stack variables
  - point to heap objects
- Heap objects
  - may have fields that are references to other heap objects

- A heap object is named by the statement that creates it
  - We assume each statement is on a separate line and name the objects by the line number of their creation statement
  - Example
    1. T v = new T; // o1 = obj created on line 1
    2. w = v ; // now w also points to o1



- Note many run-time objects may have the same name
  - The above code might be executed multiply (many calls to it or loop)

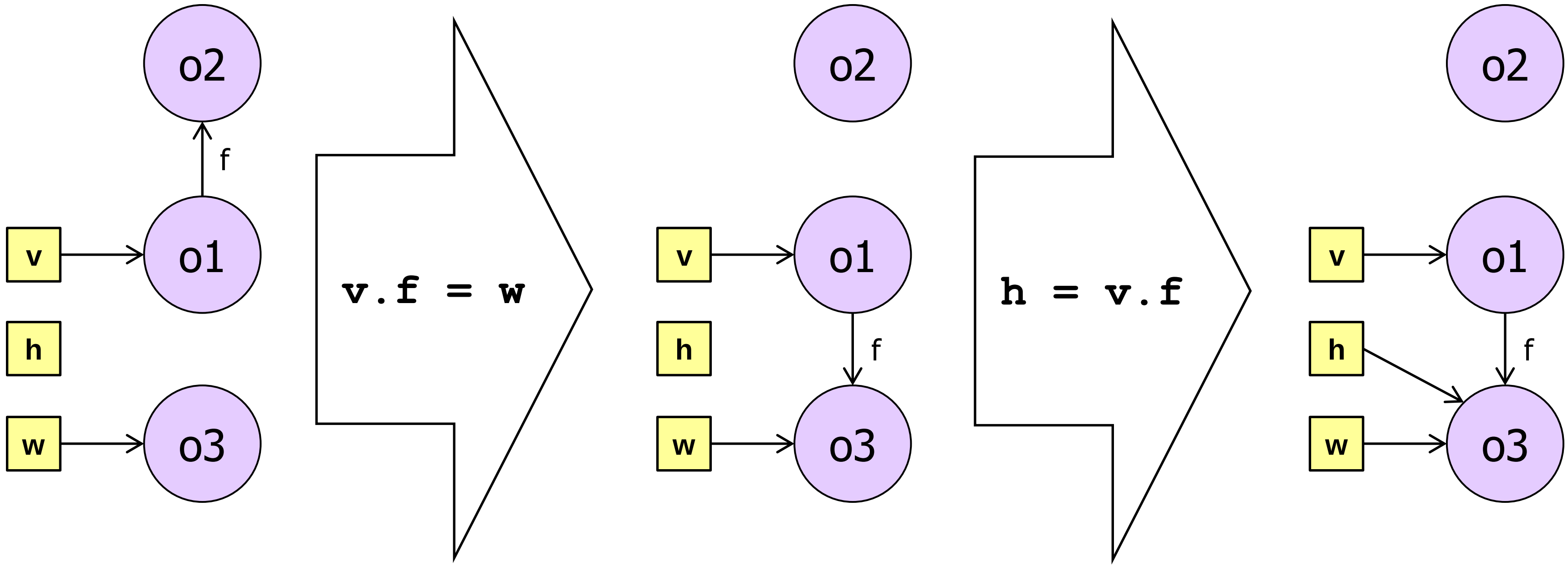
Pointer Analysis ▶ Objects flow through Fields

Putfield

- **`v.f = w`** makes the f field of the object pointed to by v point to what w points to

Getfield

- **`h = v.f`** makes h point to the object pointed to by the f field of the object pointed to by v



universität bonn ROOTS

Call Graph Construction

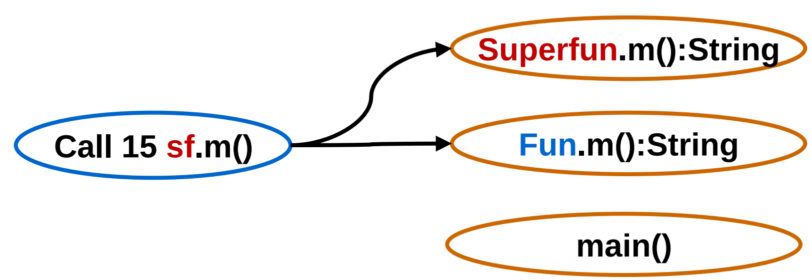
Sample Program

```
1.  class Superfun {
2.    String m() {
3.          return "This is super fun!";
4.     }
5.  }
6.  class Fun extends Superfun {
7.    String m() {
8.          return "This is fun!";
9.     }
10. }
11. void main() {
12.      Superfun  sf = new Superfun();
13.      Fun  f = new Fun();
14.      sf=f;
15.      … = sf.m();
16. }
```

receiver expression

Call Graph based on static type information

- Regard all methods in
  - static type of receiver
  - … and in each subtype
- In our example we get



universitätbonn ROOTS

Call Graph Construction: Precise

Sample Program

```
1.   class Superfun {
2.     String m(){
3.          return "This is super fun!";
4.     }
5.  }
6.   class Fun extends Superfun {
7.     String m(){
8.          return "This is fun!";
9.     }
10. }
11.  void main(){
12.     Superfun  sf = new Superfun();
13.     Fun  f = new Fun();
14.     sf=f;
15.     … = sf.m();
16. }
```

receiver expression

Call Graph based on Points-To-Analysis (PTA)

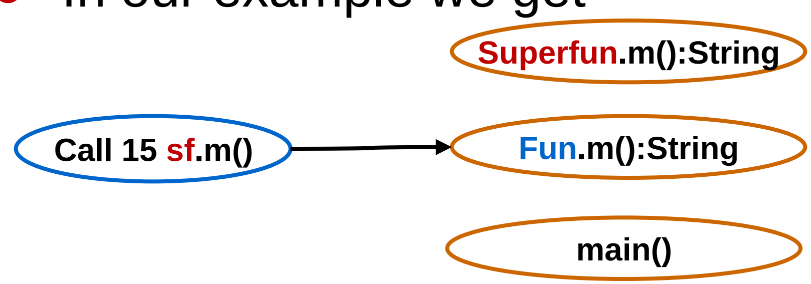
- Statically determine objects refered to by the receiver

After line 12.: sf — points to → Obj 12 : Superfun

After line 13.: f — points to → Obj 13 : Fun

After line 14.: sf — points to → Obj 13 : Fun

- Only regard methods in the classes of these objects!
- In our example we get



universitätbonn ROOTS

Logic-based Points-to Analysis

Expressing analyses as Prolog rules

Logical Representation

- Logic allows integration of different aspects of a flow problem
  - Example: **`reach(d,x,i) = "definition d of variable x can reach point i."`**

- Example: Assignment
  - For the assignment statement "v=w" in the analysed code, there is a fact / tuple "assign(v,w)" in the relation "assign(To,From)"
  - JTransformer: "assignT(*Id*, *B*, *M*, v, w)" stands for "The assignment v=w occurs in the block *B* of method *M* and has the internal identity *Id*".

- Notational convention
  - Instead of using relations for the various statement forms, we shall simply use the quoted statement itself to stand for the fact representing the statement → Abstraction from particular fact representation
  - Example: "v=w" instead of "assign(v,w)" or "assignT(*Id*, *B*, *M*, v, w)"
  - In "O: V = new T" O represents the label (line number) of the statement

Example: Iterative algorithm- round 1

pts(a, o2)

pts(b, o7)

Program code:

```
T p(T x) {
   T a = new T;
   a.f = x;
   return a;
}
void main() {
   T b = new T;
   b = p(b);
   b = b.f;
}
```

Derivation rules:

```
1. pts(V,O)    :- "O: V = new T".
2. pts(V,O)    :- "V=W", pts(W,O).
3. pts(V,O)    :- "V=W.F", alias(W,X) "X.F=V2", pts(V2,O).
4. alias(W,X) :- pts(W,O), pts(X,O).
```

Example: Iterative algorithm- round 2

pts(a, o2)

pts(b, o7)

---

pts(x, o7)

Program code:

```
T p(T x) {
   T a = new T;
   a.f = x;
   return a;
 }
 void main() {
   T b = new T;
   b = p(b);
   b = b.f;
}
```

Derivation rules:

```
1. pts(V,O)    :- "O: V = new T".
2. pts(V,O)    :- "V=W", pts(W,O).
3. pts(V,O)    :- "V=W.F", alias(W,X) "X.F=V2", pts(V2,O).
4. alias(W,X) :- pts(W,O), pts(X,O).
```


universitätbonn ROOTS

Example: Iterative algorithm- round 2

pts(a, o2)
pts(b, o7)
---
pts(x, o7)
pts(b,o2)

Program code:

```
1.  T p(T x) {
2.    T a = new T;
3.    a.f = x;
4.    return a;
5.  }
6.  void main() {
7.    T b = new T;
8.    b = p(b);
9.    b = b.f;
10.}
```

Derivation rules:

```
1. pts(V,O)    :- "O: V = new T".
2. pts(V,O)    :- "V=W", pts(W,O).
3. pts(V,O)    :- "V=W.F", alias(W,X) "X.F=V2", pts(V2,O).
4. alias(W,X)  :- pts(W,O), pts(X,O).
```

Example: Iterative algorithm- round 3

pts(a, o2)

pts(b, o7)

---

pts(x, o7)

pts(b,o2)

---

pts(x, o2)

Program code:

```
T p(T x) {
   T a = new T;
   a.f = x;
   return a;
}
void main() {
   T b = new T;
   b = p(b);
   b = b.f;
}
```

Derivation rules:

```
1. pts(V,O)    :- "O: V = new T".
2. pts(V,O)    :- "V=W", pts(W,O).
3. pts(V,O)    :- "V=W.F", alias(W,X) "X.F=V2", pts(V2,O).
4. alias(W,X)  :- pts(W,O), pts(X,O).
```

Example: Iterative algorithm- round 3

pts(a, o2)

pts(b, o7)

---

pts(x, o7)

pts(b,o2)

---

pts(x, o2)

alias(a, b)

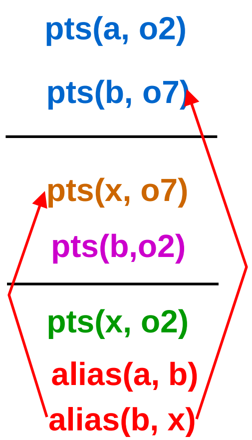
Program code:

```
1.   T p(T x)  {
2.      T a = new T;
3.      a.f = x;
4.      return a;
5.   }
6.   void main() {
7.      T b = new T;
8.      b = p(b);
9.      b = b.f;
10. }
```

Derivation rules:

```
1. pts(V,O)    :- "O: V = new T".
2. pts(V,O)    :- "V=W", pts(W,O).
3. pts(V,O)    :- "V=W.F", alias(W,X) "X.F=V2", pts(V2,O).
4. alias(W,X)  :- pts(W,O), pts(X,O).
```

Example: Iterative algorithm- round 3



Program code:

```
1.  T p(T x) {
2.    T a = new T;
3.    a.f = x;
4.    return a;
5.  }
6.  void main() {
7.    T b = new T;
8.    b = p(b);
9.    b = b.f;
10.}
```

Derivation rules:

```
1. pts(V,O)    :- "O: V = new T".
2. pts(V,O)    :- "V=W", pts(W,O).
3. pts(V,O)    :- "V=W.F", alias(W,X) "X.F=V2", pts(V2,O).
4. alias(W,X) :- pts(W,O), pts(X,O).
```

Example: Iterative algorithm- round 4

pts(a, o2)

pts(b, o7)

---

pts(x, o7)

pts(b,o2)

---

pts(x, o2)

alias(b, a)

alias(x, b)

---

alias(a, x)

Program code:

```
1.  T p(T x) {
2.     T a = new T;
3.     a.f = x;
4.     return a;
5.  }
6.  void main() {
7.     T b = new T;
8.     b = p(b);
9.     b = b.f;
10. }
```

Derivation rules:

```
1. pts(V,O)    :- "O: V = new T".
2. pts(V,O)    :- "V=W", pts(W,O).
3. pts(V,O)    :- "V=W.F", alias(W,X) "X.F=V2", pts(V2,O).
4. alias(W,X) :- pts(W,O), pts(X,O).
```

Example: Iterative algorithm- round 4

pts(a, o2)

pts(b, o7)

---

pts(x, o7)

pts(b,o2)

---

pts(x, o2)

alias(b, a)

alias(x, b)

---

alias(a, x)

pts(b,o7)

Program code:

```
T p(T x) {
   T a = new T;
   a.f = x;
   return a;
}
void main() {
   T b = new T;
   b = p(b);
   b = b.f;
}
```

Derivation rules:

```
1. pts(V,O)    :- "O: V = new T".
2. pts(V,O)    :- "V=W", pts(W,O).
3. pts(V,O)    :- "V=W.F", alias(W,X) "X.F=V2", pts(V2,O).
4. alias(W,X) :- pts(W,O), pts(X,O).
```


universitätbonn ROOTS

Example: Iterative algorithm- round 4

pts(a, o2)

pts(b, o7) ←

---

pts(x, o7)

pts(b,o2)

---

pts(x, o2)

alias(b, a)

alias(x, b)

---

alias(a, x)

~~pts(b,o7)~~ ← **already derived**

Program code:

```
T p(T x) {
   T a = new T;
   a.f = x;
   return a;
}
void main() {
   T b = new T;
   b = p(b);
   b = b.f;
}
```

Derivation rules:

```
1. pts(V,O)    :- "O: V = new T".
2. pts(V,O)    :- "V=W", pts(W,O).
3. pts(V,O)    :- "V=W.F", alias(W,X) "X.F=V2", pts(V2,O).
4. alias(W,X)  :- pts(W,O), pts(X,O).
```


universitätbonn ROOTS

Example: Iterative algorithm- round 4

pts(a, o2)

pts(b, o7)

---

pts(x, o7)

pts(b,o2)

---

pts(x, o2)

alias(b, a)

alias(x, b)

---

alias(a, x)

Program code:

```
T p(T x) {
   T a = new T;
   a.f = x;
   return a;
}
void main() {
   T b = new T;
   b = p(b);
   b = b.f;
}
```

Derivation rules:

```
1. pts(V,O)    :- "O: V = new T".
2. pts(V,O)    :- "V=W", pts(W,O).
3. pts(V,O)    :- "V=W.F", alias(W,X) "X.F=V2", pts(V2,O).
4. alias(W,X) :- pts(W,O), pts(X,O).
```


universitätbonn ROOTS

Adding Context to Prolog Rules

- To add contexts to Prolog rules we can define the following predicates:
  - **`pts(V,C,H)`**
    - ⇨ An additional argument, representing the context, must be given to the predicate pts
  - **`alias(V1,C,V2)`**
    - ⇨ An additional argument, representing the context, must be given to the predicate alias
  - **`formal(M,I,V)`**
    - ⇨ Says that V is the i-th formal parameter declared in method M
  - **`csinvokes(S,C,M,D)`**
    - ⇨ Is true if the call site S in context C calls the D context of method M
  - **`actual(S,I,V)`**
    - ⇨ V is the i-th actual parameter used in call site S

Example: Prolog program for context-sensitive points-to analysis

```
1. pts(V,C,H)    :- "H: T V = new T()",
                    csinvokes(H, C, _, _).
2. pts(V,C,H)    :- "V=W", pts(W,C,H).

3. pts(V,C,O)    :- "V=W.F",
                    alias(W,C,X),
                    "X.F=V2",
                    pts(V2,C,O).
4. pts(V,D,H)    :- csinvokes(S,C,M,D),
                    formal(M,I,V),
                    actual(S,I,W),
                    pts(W,C,H).
5. alias(W,C,X) :- pts(W,C,O), pts(X,C,O).
```

- In rule 1. predicate **`csinvokes (H,C,_,_)`** is added in order to add information that call site H is placed in context C
- Rule 4. says that if the call site S in context C calls method M of context D, then the formal parameters in method M of context D can point to the objects pointed to by the corresponding actual parameters in context C

Conclusions

- An Interprocedural analysis operates across an entire program, flowing information from call sites to their callees and vice versa
- Interprocedural Analysis needs a Call Graph
- Variants of Interprocedural Analysis:
  - Context-insensitive
  - Context-sensitive:
    - Cloning-based
    - Inline-based
    - Summary-based
- Key to any Interprocedural Analysis is a Points-to Analysis
  - Because it improves the precision of the call graph
- Expressing analyses as Prolog rules

References

- “Compilers Principles, Tehniques, & Tools” Second Edition, by Alfred Aho, Ravi Sethi, Monica Lam and Jeffrey Ullman,

  ISBN 0-321-48681–1, Publisher Greg Tobin
- “Everything Else About Data Flow Analysis” by Jeffrey Ullman,

  infolab.stanford.edu/~**ullman**/dragon/w06/lectures/**data**log.ppt
- “Why Use Datalog to Analyze Programs?” by Monica Lam, Stanford University, http://www.springerlink.com/content/y474887q446g5052/

Next Talks

- Eda: „Field-based and Field-Sensitive Analysis"
  - Properly modelling the flow of objects through fields
- Saad: „Context-Sensitive Analysis"
  - Context sensitive analysis as an extension of field-sensitive analysis
- Obaid: „On-the-fly call graph"
  - Dilemma: PTA needs a CG and a precise CG needs PTA ☹
- Mohammad: „Shape analysis"
  - More precise analyses using techniques we ignored so far the sake of efficiency