Construction of Call Graphs

- Java overview slides on web page
Call Graphs

• Widely-used representation of calling relationships
  – Key component of interprocedural control-flow analysis
  – First step toward interprocedural dataflow analysis

```java
class A {
    public static void main(...) {
        X x = new X(); // c1
        if (…) x = new Y(); // c2
        x.m(); // c3
    }
}

class X {
    void m() {...}
}

class Y extends X {
    void m() {...}
}
```
Map of what is coming next

• Call graph construction for C

• Call graph construction for object-oriented languages (focus on Java)
  – Class Hierarchy Analysis
  – Rapid Type Analysis

• If you are not familiar with Java: brief overview of relevant Java features is available on the web page
Call Graph Construction for C

Problem: function pointers


```c
typedef int (*PFB)();
struct parse_table {
    char *name;
    PFB func;
};
int func1() { ... }
int func2() { ... }
struct parse_table table[] = {
    {"name1", &func1},
    {"name2", &func2} 
};
PFB find_p_func(char *s) {
    for (i=0; i<num_func; i++)
        if (strcmp(table[i].name, s)==0)
            return table[i].func;
    return NULL;
}
int main(int argc, char *argv[]) {
    ...
    PFB parse_func=find_p_func(argv[1]);
    if (parse_func)
        (*parse_func)();
    else { ... }
}
Another Example

```c
struct _chunk { ... };
struct obstack {
    struct _chunk *chunk;
    struct _chunk *(*chunkfun) ();
    void (*freefun) (); }

void chunk_fun(struct obstack *h, void *f) {
    h->chunkfun = (struct _chunk *(*)(*()) f; }

void free_fun(struct obstack *h, void *f) {
    h->freefun = (void (*)(*) f; }

int main() {
    struct obstack h;
    chunk_fun(&h,&xmalloc);
    free_fun(&h,&xfree); ... }
```

• What do we do with these function pointers?
  • Simple answer: any function whose address is taken could possibly be called
    – Can try to restrict only to functions that “match” the types at the call site; be careful ... e.g., void *xmalloc(size_t)
Precise Resolution of Function Pointers

• Need interprocedural points-to analysis
  – Need a call graph! (flow of pointer values through parameter passing and procedure return values)

• Simple solution
  – Conservative call graph based on address-taken
  – Do points-to analysis
  – Re-compute the call graph using points-to information

• Or, call graph construction during points-to analysis
  – Start without any knowledge of f.p. calls
  – When a f.p. “shows up” in Pt(fp) at a call (*fp)(...), resolve it and update the points-to solution
  – Theoretically more precise; hard to design/implement
Call Graph Construction for C (cont’d)

• Problems come not only from function pointers ...
• **Library calls**: typically, the pre-compiled libraries are not analyzed
  – Standard libraries
  – Third-party libraries
• A library call can trigger a callback to the program
  – E.g. in stdlib.h: void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))
• **setjmp** and **longjmp**
  setjmp(jmp_buf env): stores the registers in env, including the stack pointer and the program counter
  longjmp(env): restores the registers; execution continues after the setjump program point
Methods Calls (Invocations in Java)

• **x.m(a,b):** method invocation at compile time
  – A target method is associated with the call
  – “compile-time target”, “static target”
  – Based on the declared type of variable x

```java
class A { void m(int p, int q) {...} ... }
class B extends A { void m(int r, int s) {...} ... }
A x;
x = new B();
x.m(1,2);
```

x has declared type A: compile-time target is A.m
javac encodes this in the bytecode (foo.class)

```
virtualinvoke x,<A: void m(int,int)>
```
Methods Calls (Invocations in Java)

- `virtualinvoke x, <A: void m(int,int)>` inside the JVM
  - Look at the class Z of the object that x refers to at that particular moment
  - Search Z for a method with signature `m(int,int)` and return type `void`
  - If Z doesn’t have it, go to Z’s superclass, and so on upwards, until a match is found
  - Invoke the method on the object that is pointed-to by x

Run-time (dynamic) target: “lowest” method that matches the signature and the return type of the static target (“lowest” w.r.t. inheritance chain from Z to java.lang.Object)

This process is called virtual dispatch or method lookup
Call Graphs for Software Understanding

• Tools for software understanding
  – “smart” development environments (e.g., Eclipse), maintenance tools, visualization tools, etc.
Call Graphs for Optimizations

• Resolution of virtual calls
  – e.g. “virtualinvoke” in Java bytecode

    class A { void m() { ... } }
    class B extends A { void m() { ... } }
    A a; . . . . a.m();

• If the call has only one outgoing edge in the call graph, the virtual dispatch at run time will always produce the same target
  – So, before the program is even executed, we can replace the virtual call with a “normal” call
  – Or, alternatively, after the program is loaded in the JVM, do run-time analysis and optimizations
Resolution of Virtual Calls

• Probably the oldest optimization problem for object-oriented languages
  – Smalltalk, C++, Java, many research languages
  – Goal 1: remove run-time virtual dispatch
  – Goal 2: inlining – insert the body of the called method in the caller (big performance win)

A.class
B.class
...

Bytecode-to-bytecode Optimizer

A.class
B.class
...

JVM

Virtual call resolution + inlining + other optimizations that need the call graph

• Do this at compile time or at run time
The World of Call Graph Construction [Grove & Chambers 2001]

the **perfect** call graph: cannot be computed

**X** is less precise than **Y**: $G_X$ is a superset of $G_Y$
Class Hierarchy Analysis (CHA)

- The simplest method for call graph construction
  - At the bottom of the previous slide
- Start from main, and perform reachabilityability
  - The only tricky part: virtual calls
- Helper function used in CHA: **dispatch**
  - Simulates the effects of the run-time virtual dispatch (a.k.a. method lookup)
- Note: even CHA gets tricky in the presence of dynamic class loading, reflection, native methods, etc.
  - “Assumption Hierarchy for a CHA Call Graph Construction Algorithm”, Jason Sawin and Atanas Rountev, *IEEE Int. Working Conference on Source Code Analysis and Manipulation, 2011*
dispatch
dispatch(call_site s, receiver_class rc)
  sig = signature_of_static_target(s)
  ret = return_type_of_static_target(s)
  c = rc;
  while (c != null)
    if class c contains a method m with signature sig and return type ret
      return m
    c = superclass(c)
  print “ERROR: this should be unreachable”
One Possible Implementation of CHA

Queue worklist
CallGraph Graph
worklist.addAtTail(main);
Graph.addNode(main)
while (worklist.notEmpty())
    m = worklist.getFromHead();
    process_method_body(m);
process_method_body(method m)
for each call site s inside m
  if s is a static call or a constructor call or a call through super
    add_edge(s)
  if s is a virtual call v.n(...)
    rcv_class = type_of(v);
    for each non-abstract class c that is a subclass of rcv_class or rcv_class itself
      x = dispatch(s,c)
      add_edge(s,x)
add_edge

call site s)

// for static calls, constructor calls, and calls through super
m = target(s);
if m is not in Graph
    Graph.addNode(m);
    worklist.addAtTail(m);
    Graph.addEdge(s,m)

add_edge(call_site s, run_time_target x)

// same here
Example

class A {
    void m() { }
    void n() { }
    static void main(…) {
        B b = new B();
        b.m(); // c1
        A a = b;
        a.m(); // c2
    }
}

class B extends A {
    void m() {
        A x = new A();
        x.n(); // c3
    }
}

class C extends B {
    void m() { }
    void n() { }
}

the “real” call graph
```java
class A {
    void m() { }
    void n() { }
    static void main(...) {
        B b = new B();
        b.m(); // c1
        A a = b;
        a.m(); // c2
    }
}

class B extends A {
    void m() {
        A x = new A();
        x.n(); // c3
    }
}

class C extends B {
    void m() { }
    void n() { }
}
```

workist: add and then remove A.main

c1: dispatch for rcv_type B -> target B.m
c1: dispatch for rcv_type C -> target C.m
Example

• State after processing c1
  – worklist = {B.m,C.m}
  – Graph.Nodes = {A.main, B.m, C.m}
  – Graph.Edges = { (c1,B.m), (c1,C.m) }

• Edge \((c1,C.m)\) is spurious (infeasible)
  – There is no execution of the program in which \(c1\) invokes C.m

• More precise analyses produce fewer spurious edges
  – Typically are more expensive (time/memory)
Example

```java
class A {
    void m() { }
    void n() { }
    static void main(...) {
        B b = new B();
        b.m(); // c1
        A a = b;
        a.m(); // c2
    }
}

class B extends A {
    void m() {
        A x = new A();
        x.n(); // c3
    }
}

class C extends B {
    void m() { }
    void n() { }
}
```

\[c2\]: call through `a`, which is of type `A`
\[c2\]: dispatch for rcv_type `A` -> target `A.m`
\[c2\]: dispatch for rcv_type `B` -> target `B.m`
\[c2\]: dispatch for rcv_type `C` -> target `C.m`
Example

- State after processing c2
  - worklist = \{B.m,C.m,A.m\}
  - Graph.Nodes = \{A.main, B.m, C.m, A.m\}
  - Graph.Edges = \{(c1,B.m),(c1,C.m),
    (c2,A.m),(c2,B.m),(c2,C.m) \}$

- Edges \((c2,A.m)\) and \((c2,C.m)\) are spurious

- After we are done with A.main, take the next method at the head of the queue
  - in this case B.m
class A {
    void m() { }
    void n() { }
    static void main(...) {
        B b = new B();
        b.m(); // c1
        A a = b;
        a.m(); // c2
    }
}

class B extends A {
    void m() {
        A x = new A();
        x.n(); // c3
    }
}

class C extends B {
    void m() { }
    void n() { }
}

c3: call through x, which is of type A

c3: dispatch for rcv_type A -> target A.n

c3: dispatch for rcv_type B -> target A.n

c3: dispatch for rcv_type C -> target C.n
Example

- State after processing c3
  - worklist = {C.m, A.m, A.n, C.n}
  - Graph.Nodes = {A.main, B.m, C.m, A.m, C.n}
  - Graph.Edges = {(c1,B.m), (c1,C.m), (c2,A.m), (c2,B.m), (c2,C.m), (c3,A.n), (c3,C.n) }

- Edge (c3,C.n) is spurious

- The rest of the methods in the queue have empty bodies, so the rest of the algorithm doesn’t create any new edges/nodes
Resulting Call Graph

The "real" call graph
Rapid Type Analysis

• An analysis that is the next step after CHA
  – Guaranteed to produce a call graph that is a subset of the call graph produced by CHA
  – Still quite imprecise. There are many analyses that are better than RTA

• “type analysis”
  – Idea: given a reference/pointer variable, try to figure out what types of objects this variable may refer/point to
Rapid Type Analysis

• Basic insight: some classes are *never instantiated* in reachable methods
  – i.e. there is never a new X() expression

• Main reason: programs that are built on top of libraries
  – Large parts of the library code are unused

• When we try to figure out the possible run-time targets of a virtual call, we can safely ignore classes that are not instantiated
One Possible Implementation of RTA

Queue worklist

CallGraph Graph

worklist.addAtTail(main);

Set instantiated_classes

Map pending_call_sites

Graph.addNode(main)

while (worklist.notEmpty())

    m = worklist.getFromHead();

    process_method_body(m);
process_method_body(method m)

for each expression \texttt{new X} inside m
  if (X \notin \texttt{instantiated\_classes})
    add X to \texttt{instantiated\_classes}
  resolve\_pending(X)

for each call site s inside m
  if s is a \texttt{static call} or a \texttt{constructor call} or a \texttt{call through super}
    add\_edge(s)
  if s is a \texttt{virtual call v.n(...)}
    rcv\_class = type\_of(v);
    for each non-abstract class c that is a subclass of rcv\_class or rcv\_class itself
      process\_rcv\_class(c,s)
process_rcv_class

process_rcv_class(class c, call_site s)
  x = dispatch(s,c)
  if c ∈ instantiated_classes
      add_edge(s,x)
  else // c is not currently instantiated,
      // but in the future it may be, so
      // we have to remember this edge
      remember (s,x) in pending(c)
resolve_pending(class c)
// class c became instantiated, and
// we need to add all pending edges
for each (s,x) in pending(c)
    add_edge(s,x)

Called by process_method_body:
for each expression new X
    if (X \notin instantiated_classes)
        add X to instantiated_classes
        resolve_pending(X)
class A {
    void m() { }
    void n() { }
    static void main(...) {
        B b = new B();
        b.m(); // c1
        A a = b;
        a.m(); // c2
    }
}

class B extends A {
    void m() {
        A x = new A();
        x.n(); // c3
    }
}

class C extends B {
    void m() { }
    void n() { }
}

Example

the “real” call graph
class A {
    void m() { }
    void n() { }
    static void main(...) {
        B b = new B();
        b.m(); // c1
        A a = b;
        a.m(); // c2
    }
}

class B extends A {
    void m() {
        A x = new A();
        x.n(); // c3
    }
}

class C extends B {
    void m() {
    }
    void n() {
    }
}

Example

worklist: add and then remove A.main
instantiated_classes = {B}
c1: dispatch for rcv_type B -> target B.m
c1: dispatch for rcv_type C -> target C.m
Example

• process_rcv_class(c1,B)
  – Since B is instantiated, add edge (c1,B.m)

• process_rcv_class(c1,C)
  – Since C is not instantiated, we do not add edge (c1,C.m) to the call graph
    • Remember (c1,C.m) in pending(C)

• State after processing c1
  – worklist = {B.m}
  – Graph.Nodes = {A.main, B.m}
  – Graph.Edges = {(c1,B.m)}
class A {
    void m() { }
    void n() { }
    static void main(...) {
        B b = new B();
        b.m(); // c1
        A a = b;
        a.m(); // c2
    }
}

class B extends A {
    void m() {
        A x = new A();
        x.n(); // c3
    }
}

class C extends B {
    void m() {
    }
    void n() {
    }
}

c2: call through a, which is of type A

c2: dispatch for rcv_type A -> target A.m

c2: dispatch for rcv_type B -> target B.m

c2: dispatch for rcv_type C -> target C.m
Example

• (c2,A): add (c2,A.m) to pending(A)
• (c2,B): add (c2,B.m) to Graph
• (c2,C): add (c2,C.m) to pending(C)
• State after processing c2
  – worklist = {B.m}
  – Graph.Nodes = {A.main, B.m}
  – Graph.Edges = {(c1,B.m), (c2,B.m)}
  – pending(A) = {(c2,A.m)}
  – pending(C) = {(c1,C.m),(c2,C.m)}
Example

```java
class A {
    void m() { }
    void n() { }
    static void main(...) {
        B b = new B();
        b.m(); // c1
        A a = b;
        a.m(); // c2
    }
}

class B extends A {
    void m() {
        A x = new A();
        x.n(); // c3
    }
}

class C extends B {
    void m() { }
    void n() { }
}
```

`instantiated_classes = {B,A}`

triggers a call to resolve_pending(A), with `pending(A) = { (c2,A.m) }`
Example

• resolve_pending(A)
  – Graph.Nodes = {A.main, B.m, A.m}
  – Graph.Edges = {(c1,B.m), (c2,B.m), (c2,A.m)}
  – worklist = {A.m}

• At call site c3: x.n()
  – x is of type A => A, B, or C possible
  – A and B are instantiated, there is no B.n; so, edge (c3,A.n) is added to the graph

• A.m and A.n have empty bodies, and the graph is completed
RTA vs. CHA

A.main → B.m → A.n

A.main → B.m → C.m → A.n

A.m

B.m

C.m

A.n

C.n

C.n

A.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n

A.m

B.m

C.m

A.n

C.n
RTA vs. CHA

• The key advantage: RTA was able to determine that C is never instantiated in reachable methods
  – This means that C.m and C.n can never be targets

• Of course, this is just one possible source of imprecision
  – Analyses that are “more aggressive” than RTA focus on some of these sources
Some Existing Analyses

\[ G_{\text{CFA}} \]

\[ G_{\text{KTA}} \]

\[ G_{\text{0-Bounded}} \]

\[ G_{\text{1-Bounded Linear Edge}} \]

\[ G_{\text{0-Bounded Linear Edge}} \]

\[ G_{\text{k-Bounded Linear Edge}} \]

\[ G_{\text{\infty-Bounded Linear Edge}} \]

\[ G_{\text{VTA}} \]

\[ G_{\text{FTA}} \]

\[ G_{\text{MTA}} \]

\[ G_{\text{XTA}} \]

\[ G_{\text{CIA}} \]
More Existing Analyses
Class Analysis

- **Class analysis**: given a reference variable $x$, what are the classes of the objects that $x$ may refer to?
  - a.k.a. “type analysis” (e.g., RTA)
  - After a class analysis, it is trivial to construct the call graph
    - As a separate post-processing phase

- Most class analyses construct the call graph on the fly during the analysis
  - For object-oriented languages, “call graph construction”, “class analysis”, and “type analysis” are often used as synonyms

- Points-to analysis can be thought of as a particular form of class/type analysis
  - Next: “classic” points-to analysis, closely related to 0-CFA type analysis (see two slides earlier)