Interprocedural control-flow analysis

Nate Nystrom
CS 711
6 Sep 05
Call graphs

- Statically compute a precise call graph
  - Maps call sites to functions called
- Challenge:
  - Methods
  - Higher-order functions
- Can use precise call graph for:
  - optimization
    - reduce dispatch overhead
    - convert calls to lambdas to direct jumps
    - reduce code size
  - program understanding
Various techniques

- **Unique Name** [Calder and Grunwald, POPL’94]
- **Class Hierarchy Analysis** [Dean, Grove, Chambers, ECOOP’95]
  [Fernandez, PLDI’95]
- **Optimistic Reachability Analysis**
  - **Rapid Type Analysis** [Bacon and Sweeney, OOPSLA’96]
- **Propagation-based analysis**
  - **0-CFA** [Shivers, PLDI’88]
  - **k-CFA** [Shivers ‘91]
- **Unification-based analysis** [Steensgaard, POPL’96]
- **Interprocedural Class Analysis** [DeFouw, Grove, Chambers, POPL’98]
**Unique Name**

- Does not build call graph, but does resolve virtual calls
- If only one method named m in entire program
  - Replace all virtual calls to a method named m with a non-virtual call
- Do at link time on object files
- Can resolve (1) only
- For C++ benchmarks, resolves 15% of virtual calls
- Can’t handle same method name in different classes

```cpp
class A {
    int foo() { return 1; }
}
class B extends A {
    int foo() { return 2; }
    int bar(int i) { return i+1; }
}
void main() {
    B p = new B();
    int r1 = p.bar(1);  // 1: B.bar
    int r2 = p.foo();   // 2: B.foo
    A q = p;
    int r3 = q.foo();   // 3: B.foo
}
```
Class Hierarchy Analysis

- Use static type of receiver and the class hierarchy to narrow set of possible targets
- Whole program analysis
- Flow insensitive
- $O(N)$
- Can resolve (1) and (2)
- For C++ benchmarks, resolves 51% of virtual calls

```java
class A {
    int foo() { return 1; }
}
class B extends A {
    int foo() { return 2; }
    int bar(int i) { return i+1; }
}
void main() {
    B p = new B();
    int r1 = p.bar(1); // 1: B.bar
    int r2 = p.foo();   // 2: B.foo
    A q = p;
    int r3 = q.foo();   // 3: B.foo
}
```
Rapid Type Analysis

- Do CHA to build call graph
- If no object of class \( C \) allocated in the program,
  - Remove edges to methods of \( C \)
- \( O(N) \)
- Slightly more expensive than CHA
- Can resolve (1), (2), and (3)
- For C++ benchmarks, resolves 71% of virtual calls

```cpp
class A {
    int foo() { return 1; }
}
class B extends A {
    int foo() { return 2; }
    int bar(int i) { return i+1; }
}
void main() {
    B p = new B();
    int r1 = p.bar(1); // 1: B.bar
    int r2 = p.foo();  // 2: B.foo
    A q = p;
    int r3 = q.foo();  // 3: B.foo
}
```
Disjoint polymorphism

- Multiple related object types used independently
  - e.g., Square and Circle objects are never mixed together in, say, a Collection of Shapes
- Pathological case:
  - Derived1 and Derived2 are disjoint
  - No Base objects allocated
  - All calls are through Base pointers

```java
class Base {
    void m() { assert(false); }
    void p() { assert(false); }
}

class Derived1 extends Base {
    void m() { ... }
}

class Derived2 extends Base {
    void p() { ... }
}
```
Unification-based analysis

- Partitions variables in program and maps each partition to a set of classes
- Initialize with each variable in own partition
- If classes can flow between variables, unify the classes for those variables
  
  ```java
target = source;
T1 m(T2 target) { ... }
m(source);
```

- Resolves (4), but not (5)
- $O(Na(N,N))$

```java
class A {
    int foo() { return 1; }
}
class B extends A {
    int foo() { return 2; }
}
void main() {
    A p = new B();
    int r1 = p.foo(); // 4: B.foo
    A q = new A();
    q = new B();
    int r2 = q.foo(); // 5: B.foo
}
```
Interprocedural class analysis

• Framework integrates
  • propagation-based analysis (0-CFA)
  • unification-based analysis
  • optimistic reachability analysis (RTA)
• Computes set of classes for each program variable
• Builds call graph as side effect
Flow graph representation

- Node for each variable, method, new, call
- Algorithm computes set of classes for each node
- Edge between two nodes if classes can flow between them

```java
target = source;
T1 m(T2 target) { ... }
m(source);
```

source ➔ target
Basic algorithm (0-CFA)

- Construct nodes and edges for top-level variables, statements, and expressions (e.g., main)
- Propagate classes through flow graph starting with main and top-level new expressions
- When call encountered, add edge to target and construct flow graph for target method (if not already done)
- If method not reachable, it will be pruned (as in RTA)
Edge filters

- Edges may have a filter set
  - encode constraints ensured by type declarations or by dynamic dispatch
- Don’t propagate class if filter does not include that class
- Makes algorithm more precise than 0-CFA

```java
class B { m() { ... this ... } }
class C ext B { m() { ... this ... } }
B o = new C();
o.m()
```

```plaintext
B.m()  filter: {B}
   \( \text{this}_B \)

C.m()  filter: {C}
   \( \text{this}_C \)
o
```
Call merging

- Analysis parameterized by \textit{MergeCalls}
- When \textit{MergeCalls} = false:

\[
\begin{align*}
o1.m(a1) & \rightarrow this_B \\
a1 & \rightarrow x1 \\
o2 & \rightarrow this_C \\
a2 & \rightarrow x2 \\
o2.m(a2) & \rightarrow this_B \\
B.m(x1) & \{ ... \} \\
C.m(x2) & \{ ... \}
\end{align*}
\]

- When \textit{MergeCalls} = true:

\[
\begin{align*}
o1.m(a1) & \rightarrow \text{m0} \\
a1 & \rightarrow x1 \\
o2 & \rightarrow \text{m1} \\
a2 & \rightarrow x2 \\
o2.m(a2) & \rightarrow \text{m1} \\
B.m(x1) & \{ ... \} \\
C.m(x2) & \{ ... \}
\end{align*}
\]
Node merging

- Can speedup analysis by merging nodes into *supernodes*
- Nodes merged with successors

```cpp
target = source;
T1 m(T2 target) { ... }
m(source);
```

- Always merging is equivalent to unification-based analysis
Merging parameters

- Analysis parameterized by $P$ and $\text{MergeWithGlobal}$
- When $P = k$, merge node with its successors if node visited more than $k$ times
- When $P = 0$, always merge
- When $P = N$, never merge
- When $\text{MergeWithGlobal} = \text{true}$, use only one global supernode
## Instantiations

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$P$</th>
<th>$MergeWithGlobal$</th>
<th>$MergeCalls$</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-CFA</td>
<td>$N$</td>
<td>N/A</td>
<td>false</td>
<td>$O(N^3)$</td>
</tr>
<tr>
<td>linear-edge 0-CFA</td>
<td>$N$</td>
<td>N/A</td>
<td>true</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>bounded 0-CFA</td>
<td>$O(1)$</td>
<td>false</td>
<td>false</td>
<td>$O(N^2\alpha(N,N))$</td>
</tr>
<tr>
<td>bounded linear-edge 0-CFA</td>
<td>$O(1)$</td>
<td>false</td>
<td>true</td>
<td>$O(N\alpha(N,N))$</td>
</tr>
<tr>
<td>simply bounded 0-CFA</td>
<td>$O(1)$</td>
<td>true</td>
<td>false</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>simply bounded linear-edge 0-CFA</td>
<td>$O(1)$</td>
<td>true</td>
<td>true</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>equivalence class analysis</td>
<td>0</td>
<td>false</td>
<td>true</td>
<td>$O(N\alpha(N,N))$</td>
</tr>
<tr>
<td>RTA</td>
<td>0</td>
<td>true</td>
<td>true</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
Analysis time

- Analysis time increases slightly with $P$
  - Mostly flat when $P$ small, finite
- $MergeWithGlobal = true$ (simply bounded)
  - saves ~10% on Cecil
  - negligible improvement for Java
    - but all the benchmarks are Java compilers
  - 250% for one case when $P = N$
- $MergeCalls = true$ (linear edge)
  - up to 3x for Cecil, or more
  - only 5-20% savings for Java
    - no multimethods, so less edge filtering?
  - some programs can only be analyzed with linear edge (or small $P$)
Precision

- Larger $P$ more precise (less merging)
  - Run-time speedup 0-10% for $P = 0$, 10-350% for $P = N$
- $MergeCalls = true$ (linear edge)
  - About as precise as quadratic edge
  - Less so for Java, but no difference in speedup
- $MergeWithGlobal = true$ (simply bounded)
  - Slightly less precision
  - but on some Cecil benchmarks, improved precision of $MergeWithGlobal = false$ caused 2.5x speedup
  - precision lost on hot virtual calls?
Questions

• All of these analyses are whole-program
  • Can they be modularized?
• Integrating alias analysis, or more precise points to analysis
• Extend class analysis to incorporate context as in $k$-CFA