Inter-procedural CAT

Simone Campanoni
simone.campanoni@northwestern.edu
Procedures/functions

• Abstraction
  • Cornerstone of programming
  • Introduces barriers to analysis

• So far looked at **intra-procedural** analysis
  • Analyzing a single procedure

• **Inter-procedural** analysis uses calling relationships among procedures (Call Graph)
  • Enables more precise analysis information

```c
void bar (void){
  x = 5;
  px = &x;
  foo(px);
  y = x + 5;
}
```

Is x constant?
Inter-procedural analysis

**Goal:** Avoid making overly conservative assumptions about the effects of procedures and the state at call sites

**Terminology**

```c
int a, e;  // Globals
void foo(int *b, int *c){  // Formal parameters
    (*b) = e;
}

bar(){
    int d;  // Local variables
    foo(a, d);  // Actual parameters
}
```
Inter-procedural analysis vs. inter-procedural transformation

**Inter-procedural analysis**

- Gather information across multiple procedures (up to the entire program)
- Can use this information to improve intra-procedural analyses and transformation (e.g., CP)

**Inter-procedural transformation**

- Code transformations that involve multiple procedures e.g., Inlining, procedure cloning, function specialization
Outline

① Sensitivity of analysis

② Single compilation

③ Separate compilations

④ Caller -> callee vs. callee -> caller propagations

⑤ Final remarks
Sensitivity of intra-procedural analysis

• Flow-sensitive vs. flow-insensitive
Flow sensitivity example

Is x constant?

```c
void f (int x){
    A: x = 4;
    ...
    B: x = 5;
}
```

Flow-sensitive analysis
- It can compute one answer for every program point
  - x is 4 after A
  - x is 5 after B
- Requires iterative data-flow analysis or similar technique

Flow-insensitive analysis
- It computes one answer for the entire procedure
  - x is not constant
- Can compute in linear time
- Less accurate (ignores control flows)
Sensitivity of intra-procedural analysis

- Flow-sensitive vs. flow-insensitive
- Path-sensitive vs. path-insensitive
Path sensitivity example

Is x constant?

if (x == 0)

x = 4;

x = 5;

print(x)

Path-sensitive analysis

• Computes one answer for every execution path
  • x is 4 at print(x) if you came from the left path
  • x is 5 at print(x) if you came from the right path

• Subsumes flow-sensitivity

• Very expensive

Path-insensitive analysis

• Computes one answer for all path
  • x is not constant at print(x)
Sensitivity of inter-procedural analysis

- Flow-sensitive vs. flow-insensitive
- Path-sensitive vs. path-insensitive
- Context-sensitive vs. context-insensitive
Is \( x \) constant?

\[
\begin{align*}
& a = \text{id}(4); & b = \text{id}(5); \\
\text{id}(x) & \{ \text{return } x; \}
\end{align*}
\]

Context-sensitive analysis

- It can compute one answer for every call-site
  - \( x \) is 4 in the first call
  - \( x \) is 5 in the second call
- Re-analyzes callee for each caller

Context-insensitive analysis

- It computes one answer for all call-sites:
  - \( x \) is not constant
- Perform one analysis independent of callers
- Suffers from unrealizable paths:
  - Can mistakenly conclude that \( \text{id}(4) \) can return 5 because
    we merge information from all call-sites
Call graph

• First problem: how do we know what procedures are called from where?
  • Especially difficult in higher-order languages, languages where functions are values
  • What about C programs?
  • We’ll ignore this for now

• Let’s assume we have a (static) call graph
  • Indicates which procedures can call which other procedures, and from which program points

```c
void foo (int a, int (*p_to_f)(int v)){
  int l = (*p_to_f)(5);
  a = l + 1;
  return a;
}
```
Call graph example

```java
f() {
    1: g();
    2: g();
    3: h();
}

g() {
    4: h();
}

h() {
    5: f();
    6: i();
}

i() { ... }
```

From now on we assume we have a static call graph.
Generating a call graph with LLVM

• From the command line:
  opt -dot-callgraph program.bc -disable-output
  (see test0)

• From your pass:
  • Explicit iteration
    • LLVM_callgraph/llvm/[0-4]
DEMO
Generating a call graph with LLVM

- From the command line:
  opt -dot-callgraph program.bc -disable-output
  (see test0)

- From your pass:
  - Explicit iteration
    - LLVM_callgraph/llvm/[0-4]
  - CallGraphWrappingPass
    - LLVM_callgraph/llvm/[5-6]
Using CallGraphWrappingPass

• Declaring your pass dependence
  ```c++
  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired< CallGraphWrapperPass >();
  }
  ```

• Fetching the call graph
  ```c++
  bool runOnModule(Module &M) override {
    errs() << "Module " << M.getName() << "\n";
    CallGraph &CG = getAnalysis<CallGraphWrapperPass>().getCallGraph();
  }
  ```
Using CallGraphWrappingPass

• From a Function to a node of the call graph

```cpp
errs() << " Function " << F.getName() << "\n";
CallGraphNode *n = CG[&F];
```

• From node to callees

```cpp
for (auto callee : *n){
    auto calleeNode = callee.second;
    auto callInst = callee.first;
}
```

• From node to Function

```cpp
auto calleeF = calleeNode->getFunction();
errs() << " " << calleeF->getName() << "\n";
```
DEMO
Outline

① Sensitivity of analysis

② Single compilation

③ Separate compilations

④ Caller -> callee vs. callee -> caller propagations

⑤ Final remarks
Intra-procedural dataflow analysis

• How have we been performing reaching definitions so far?

```c
main() {
    A: x = 7;
    B: r = p();
    C: y = 80;
    D: t = p();
    E: print t, r;
}
int p (void) {
    F: m = 1;
    G: return m;
}
```
Intra-procedural dataflow analysis

• How have we been performing reaching definitions so far?

main() {
  A: x = 7;
  B: r = p();
  C: y = 80;
  D: t = p();
  E: print t, r;
}

int p (void) {
  F: m = 1;
  G: return 1;
}
Inter-procedural dataflow analysis
flow-sensitive

• How can we handle procedure calls?
• Obvious idea: make one big CFG (control-flow supergraph)

main() {
A: x = 7;
B: r = p();
C: y = 80;
D: t = p();
E: print t, r;
}

int p (void) {
F: m = 1;
G: return m;
}

• Better accuracy
• Worst analysis time
  • No separate analysis

A
\[\text{IN} = \{A\}\]

B
\[\text{IN} = \{A, F, C\}\]
\[\text{IN} = \{A, F, C\}\]

C
\[\text{IN} = \{A, F, C\}\]
\[\text{IN} = \{A, F, C\}\]

D

E

F
\[\text{IN} = \{A, F\}\]
\[\text{IN} = \{A\}\}\]

G

H
Inter-procedural dataflow analysis
flow-sensitive

• Make one big CFG (control-flow supergraph)

main() {
A: x = 7;
B: r = p(x);
C: y = 80;
D: t = p(y);
E: print t, r;
}

int p (int v) {
F: m = v + 1
G: return m;
}

• Better accuracy
• but not enough
Inter-procedural dataflow analysis flow-sensitive

• Make one big CFG (control-flow supergraph)

```c
main() {
  A: x = 7;
  B: r = p(x);
  C: y = 80;
  D: t = p(y);
  E: print t, r;
}

int p (int v) {
  F: m = v + 1
  G: return m;
}
```

• Better accuracy
• but not enough

• Problem: v is seen from the point of view of all call sites
• How to address this problem?
Inter-procedural dataflow analysis flow/context-sensitive

• Make one big CFG (control-flow supergraph)

main() {
  A: x = 7;
  B: r = p(x);
  C: y = 80;
  D: t = p(y);
  E: print t, r;
}

int p (int v) {
  F: m = v + 1
  G: return m;
}

• Better accuracy
• More memory/analysis time
Inter-procedural dataflow analysis
flow/context-sensitive

main() {
  A: x = 7;
  B: r = p(x);
  C: y = 80;
  D: t = p(y);
  E: print t, r;
}

int p (int v) {
  F: if (v < 10)
      G: m = 1;
  else
      H: m = 2;
  I: return m;
}
• Even an inter-procedural flow- and context-sensitive analysis isn’t able to perform the constant propagation we want
• We need to make our analysis even more complex

• Since this seems hard, let’s try something easier
• Let’s try to follow a simpler solution:
  • We copy the body of the callee inside the caller
  • Function inlining
Inter-procedural code transformation: function inlining

• Function inlining:
  • Use a new copy of a procedure’s CFG at a call site
  • Adjust copied code within the caller
    (e.g., rename variables, map formal parameters to actual parameters)

```c
void myF (void){
    int r0 = myG(3, 4);
    int r1 = r0 + 1;
    return r1;
}
```

```c
int myG (int p0, int p1){
    int v0 = p0 + p1;
    return v0;
}
```

Let’s inline this call
Inter-procedural code transformation: function inlining

• Function inlining:
  • Use a new copy of a procedure’s CFG at a call site
  • Adjust copied code within the caller
    (e.g., rename variables, map formal parameters to actual parameters)

```c
void myF (void){
    int p0 = 3; int p1 = 4;
    int v0 = p0 + p1;
    int r0 = v0;
    int r1 = r0 + 1;
    return r1;
}
```

```c
int myG (int p0, int p1){
    int v0 = p0 + p1;
    int r0 = myG(3, 4);
    return v0;
}
```
Inter-procedural code transformation: function inlining

• Function inlining:
  • Use a new copy of a procedure’s CFG at a call site
  • Adjust copied code within the caller
    (e.g., rename variables, map formal parameters to actual parameters)
• In LLVM:
  • You don’t need to implement this transformation, it already exists 😊
  • InlineResult InlineFunction(CallBase *, InlineFunctionInfo &, ...) 

InlineFunctionInfo IFI;
if (InlineFunction(call, IFI).isSuccess()) {
  #include "llvm/Transforms/Utils/Cloning.h"
  ...
} else { ...

Extra parameters are optional
Function inlining in LLVM and alias analysis

- InlineResult InlineFunction(
  CallBase &, InlineFunctionInfo &,
  AAResults *CalleeAAR = nullptr,
  bool InsertLifetime = true, ...)

void f (){
  ... // pre_g
  call g()
  ... // post_g
}
void g(){
  %1 = alloca(...)
  ... // g_body
  ... // post_g
}

void f (){
  %1 = alloca()
  ... // pre_g
  ... // g_body
  ... // post_g
}

New live range of %1

But we know %1 can only be used (directly or indirectly) within g_body
Inter-procedural code transformation: function inlining

Example of function inlining: *inline the callee of B*

main() {
    A: x = 7;
    B: r = p(x);
    C: y = 80;
    D: t = p(y);
    E: print t, r;
}

int p (int v) {
    F: if (v < 10)
    G:       m = 1;
    else
    H:       m = 2;
    I: return m;
}
Inter-procedural code transformation: function inlining

```c
main() {
    A: x = 7;
    B: r = p(x);
    C: y = 80;
    D: t = p(y);
    E: print t, r;
}

int p (int v) {
    F: if (v < 10)
    G:       m = 1;
    else
    H:       m = 2;
    I: return m;
}
```

Another example of function inlining: `inline the callee of D`
Inter-procedural code transformation: function inlining

```c
main() {
    A: x = 7;
    B: r = p(x);
    C: y = 80;
    D: t = p(y);
    E: print t, r;
}

int p (int v) {
    F: if (v < 10)
        G:       m = 1;
    else
        H:       m = 2;
    I: return m;
}
```
Inter-procedural dataflow analysis
flow/context-sensitive

main() {
A: x = 7;
B: r = p(x);
C: y = 80;
D: t = p(y);
E: print t, r;
}

int p (int v) {
F: if (v < 10)
G: m = 1;
else
H: m = 2;
I: return m;
}

• What did it change?
• Solutions?
Function inlining

• Inlining
  • Use a new copy of a procedure’s CFG at each call site
  • Useful if not used always

• Problems?
  • May be expensive! Exponential increase in size of CFG
  • You can’t always determinate callee at compile time (e.g., in OO languages)
  • Library source is usually unavailable

• What about recursive procedures?
  p(int n) { ... p(n-1); ... }

• More generally, cycles in the call graph
main() {
A: x = 7;
B: r = p(x);
C: y = 80;
D: t = p(y);
E: print t, r;
}

int p (int v) {
F: if (v < 10) 
G:       m = 1;
else
H:       m = 2;
I: return m;
}

• Better accuracy
• Much worst analysis time
Inter-procedural dataflow analysis
flow/context-sensitive

What about programs with a deep hierarchy of many procedures?
Re-analyze callee for all distinct calling paths

• **Pro:** precise
• **Cons:** what’s the analysis time?
• **Solution:** separate compilation

```c
main() {
  A: x = 7;
  B: r = p(x);
  C: y = 80;
  D: t = p(y);
  E: print t, r;
}

int p(int v) {
  F: if (v < 10)
  G:       m = 1;
  H:       m = 2;
  I: return m;
}
```
Outline

① Sensitivity of analysis

② Single compilation

③ Separate compilations

④ Caller -> callee vs. callee -> caller propagations

⑤ Final remarks
Separate compilation

• Each function is analyzed separately

• The result of the analysis of a function is a “summary node”, which reports what you need to know about this function

• When you analyze a function $F$ that invokes $G$, you use the summary node of $G$ to analyze $F$

• Typically: the call graph is used to first analyze callees and then callers
Summary context: example

- Summary context: summarize effect of called procedure for callers

```c
main() {
    A: x = 7;
    B: r = p(x);
    C: y = 80;
    D: t = p(y);
    E: print t, r;
}

int p (int v) {
    F: int v = 42;
    I: return v;
}
```

Higher accuracy compared to intra-procedural

Summary: p returns 42
main() {
    A: x = 7;
    B: r = p(x);
    C: y = 80;
    D: t = p(y);
    E: print t, r;
}

int p (int v) {
    F: if (v < 10)
    G:       m = 1;
    else
    H:       m = 2;
    I: return m;
}
Separate compilation

• Each function is analyzed separately

• The result of the analysis of a function is a “summary node”, which reports what you need to know about this function

• We can decide to increase the amount of information embedded in the summary node
Summary context: example 2

• Summary context: summarize effect of called procedure depending on formal parameters for callers

```
main() {
  A: x = 7;
  B: r = p(x);
  C: y = 80;
  D: t = p(y);
  E: print t, r;
}
```

```
int p (int v) {
  F: if (v < 10)
    G: m = 1;
  else
    H: m = 2;
  I: return m;
}
```

Summary: p returns

Constant 1 if parameter is < 10
Constant 2 otherwise
Designing an inter-procedural analysis

Summary precision
- e.g., constant/not-constant depending on the formal parameters

What to summarize
- e.g., returning values
- e.g., memory locations modified

Number of summary nodes

Sensitivity
- e.g., flow
Context sensitivity

• Simplest solution: 1 copy per procedure

main

IN = {A}

IN = {A, B}

IN = {A, B, C}

A: x = 7;
B: r = p(x);
C: y = 80;
D: t = p(y);
E: print t, r;

main() {

IN = {G, H}

IN = {G}

IN = {} 

F: if (v < 10)
G: m = 1;
else
H: m = 2;
I: return m;
}

int p (int v) {

Summary: p doesn’t return a constant
Context sensitivity

• Simplest solution: 1 copy per procedure

• Do we have a summary node for p?
• Yes. Fetch it

Summary: p doesn’t return a constant
Context sensitivity

- Simplest solution: 1 copy per procedure
- Simple solution: make a small number of copies of contexts (e.g., all callees of a procedure from a caller)

```
main() {
    A: x = 7;
    B: r = p(x);
    C: y = 80;
    D: t = p(y);
    E: print t, r;
}

int p (int v) {
    F: if (v < 10)
    G:       m = 1;
    else
    H:       m = 2;
    I: return m;
}
```

- Do we have a summary node for p(7)?
- No. Compute it
Context sensitivity

• Simplest solution: 1 copy per procedure
• Simple solution: make a small number of copies of contexts (e.g., all callees of a procedure from a caller)

```
main() {
A: x = 7;
B: r = p(x);
C: y = 80;
D: t = p(y);
E: print t, r;
}

int p (int v) {
F: if (v < 10)
G:       m = 1;
H:       m = 2;
I: return m;
}
```
Context sensitivity

• Simplest solution: 1 copy per procedure
• Simple solution: make a small number of copies of contexts (e.g., all callees of a procedure from a caller)

main

IN = {A}

IN = {A, B}

IN = {A, B, C}

• Do we have a summary node for p(80)?
• No. Compute it

main() {
    A: x = 7;
    B: r = p(x);
    C: y = 80;
    D: t = p(y);
    E: print t, r;
}

int p (int v) {
    F: if (v < 10)
    G: m = 1;
else
    H: m = 2;
    I: return m;
}
Context sensitivity

• Simplest solution: 1 copy per procedure
• Simple solution: make a small number of copies of contexts (e.g., all callees of a procedure from a caller)

```
main()
A: x = 7;
B: r = p(x);
C: y = 80;
D: t = p(y);
E: print t, r;

int p (int v) {
F: if (v < 10)
G: m = 1;
else
H: m = 2;
I: return m;
}
```

Summary: $p(7)$ returns 1
Context sensitivity

• Simplest solution: 1 copy per procedure
• Simple solution: make a small number of copies of contexts (e.g., all callees of a procedure from a caller)

main

IN = \{A\}

IN = \{A, B\}

IN = \{A, B, C\}

IN = \{A, B, C, D\}

Summary: \(p(7)\) returns 1
\(p(80)\) returns 2

main() {
A: \(x = 7;\)
B: \(r = p(x);\)
C: \(y = 80;\)
D: \(t = p(y);\)
E: print \(t, r;\)
}

int p (int v) {
F: if (v < 10)
G: m = 1;
else
H: m = 2;
I: return m;
}
Context sensitivity

• Simplest solution: 1 copy per procedure
• Simple solution: make a small number of copies of contexts (e.g., all callees of a procedure from a caller)
• Advanced solutions: use context information to determine when to share a copy
• Choice of what to use for context will produce different tradeoffs between precision and scalability
• Common choice: approximation of call stack
main() {
1: int v = p(4);
2: v += p(3);
}
p (int f) {
3: return q(f)
}
q(int f){
4: return f;
}
main() {
  1: p();
  2: p();
}

p () {
  3: q()
}

q(){
  4: return;
}

Context sensitivity example
Other contexts

• Context sensitivity distinguishes between different calls of the same procedure
  • Choice of contexts determines which calls are differentiated

• Other choices of context are possible
  • Caller stack
    • Less precise than call-site stack
    • E.g., context “2::2” and “2::3” would both be “fib::fib”

• Object sensitivity: which object is the target of the method call?
  • For OO languages
  • Maintains precision for some common OO patterns
  • Requires pointer analysis to determine which objects are possible targets
  • Can use a stack (i.e., target of methods on call stack)
Designing an inter-procedural analysis

Summary information
- e.g., constant/not-constant
- e.g., no summary (single compilation)

What to summarize

How to propagate

Number of summary nodes

Sensitivity
- e.g., context

58
Outline

① Sensitivity of analysis

② Single compilation

③ Separate compilations

④ Caller -> callee vs. callee -> caller propagations

⑤ Final remarks
Inter-procedural analysis

• What to propagate through the call graph

• How to propagate through the call graph

• Example
Two types of information

• Track information that flows into a procedure
  • Also known as propagation problems
    e.g., What formals are constant?
    e.g., Which formals are aliased to globals?

• Track information that flows out of a procedure
  • Also known as side effect problems
    e.g., Which globals are def’d/used by a procedure?
    e.g., Which locals are def’d/used by a procedure?
    e.g., Which actual parameters are def’d by a procedure?

Summary: \( p(7) \) returns 1
\( p(80) \) returns 2
Summary: \( p \) modifies Global \( @X \) if parameter is < 10
Summary examples

• Propagation Summaries
  • MAY-ALIAS: The set of formals that may be aliased to globals and each other
  • MUST-ALIAS: The set of formals that are definitely aliased to globals and each other
  • CONSTANT: The set of formals that must be constant

• Side-effect Summaries
  • MOD: The set of variables possibly modified (defined) by a call to a procedure
  • REF: The set of variables possibly read (used) by a call to a procedure
  • KILL: The set of variables that are definitely killed by a procedure (e.g., in the liveness sense)
Inter-procedural analysis

• What to propagate through the call graph

• How to propagate through the call graph

• Example
Computing inter-procedural summaries

• Top-down (from callers to callees)
  • Summarize information about the caller (MAY-ALIAS, MUST-ALIAS)
  • Use this information inside the procedure body
    int a;
    void foo(int &b, &c){
      ...
    }
    foo(a,a);

• Bottom-up (from callees to callers)
  • Summarize the effects of a call (MOD, REF, KILL)
  • Use this information around procedure calls
    x = 7;
    foo(x);
    y = x + 3;
Bi-directional inter-procedural summaries

• Inter-procedural Constant Propagation (ICP)
  • Information flows from caller to callee and back
  ```c
  int a, b, c, d;
  void foo(e){
    a = b + c;
    d = e + 2;
  }
  foo(3);
  ```
  The calling context tells us that the formal e is bound to the constant 3, which enables constant propagation within foo()
  After calling foo() we know that the constant 5 (3+2) propagates to the global d

• Inter-procedural Alias Analysis
  • Forward propagation: aliasing due to reference parameters
  • Side-effects: points-to relationships due to multi-level pointers
Inter-procedural analysis

• What to propagate through the call graph

• How to propagate through the call graph

• Example
Example: identify functions that might be affected by randomness

Problem:
Identify functions that might directly or indirectly invoke \texttt{rand()}

Output:
The set of functions affected by \texttt{rand()} and the length of the shortest path in the call graph to an invocation to \texttt{rand()}.

How can we do it?

You can find the solution shown in the next slides here: LLVM_callgraph
Example: identify functions that might be affected by `rand()`
Example: identify functions that might be affected by rand()

Functions affected:
Level 0: q
Level 1: p1
Level 2: p2
Level 2: main
Level 3: p3

Functions not affected:
Example: identify functions that might get affected by rand()

Data structures:

```c
enum randomUses {TBC, invoked, notInvoked};
struct random_info_t {
    randomUses r;
    uint32_t level;
};
std::map<Function *, random_info_t> randomInfo;
```
Example: identify functions that might get affected by rand()

```c
void printStatus (Module &M)
{
    errs() << " Functions affected:\n";
    for (auto &F : M) {
        if (randomInfo[&F].r == invoked)
            errs() << " Level " << randomInfo[&F].level << ": " << F.getName() << "\n";
    }
    errs() << " Functions not affected:\n";
    for (auto &F : M) {
        if (randomInfo[&F].r == notInvoked)
            errs() << " " << F.getName() << "\n";
    }
    return ;
}
```

Functions affected:
- Level 0: q
- Level 1: p1
- Level 2: p2
- Level 3: p3
- Level 2: main

Functions not affected:
Example: identify functions that might get affected by rand()

```java
bool runOnModule(Module &M) override {
    errs() << "Module " << M.getName() << "\n";

    errs() << " Identify functions affected directly\n";
    tagFunctionsDirectlyAffected(M);
    printStatus(M);

err() << " Identify functions affected indirectly\n";
identifyFunctionsIndirectlyAffected(M);
printStatus(M);

errs() << " Identify functions not affected\n";
identifyFunctionsNotAffected(M);
printStatus(M);

    return false;
}
```
Example: identify functions that might get affected by `rand()`
Example: identify functions that might get affected by rand()
Example: identify functions that might get affected by rand()
Example: identify functions that might get affected by `rand()`
Example: identify functions that might get affected by rand()
Example: identify functions that might get affected by rand()
Computing inter-procedural summaries

• Top-down
  • Summarize information about the caller (MAY-ALIAS, MUST-ALIAS)
  • Use this information inside the procedure body
    int a;
    void foo(int &b, &c){
      ...
    }
    foo(a,a);

  Is our pass Top-down or bottom-up?

• Bottom-up
  • Summarize the effects of a call (MOD, REF, KILL)
  • Use this information around procedure calls
    x = 7;
    foo(x);
    y = x + 3;
Outline

① Sensitivity of inter-procedural analysis

② Single compilation

③ Separate compilations

④ Caller -> callee vs. callee -> caller propagations

⑤ Final remarks
What about cycles in the call graph?

```cpp
f() {  
  1:  g();  
  2:  g();  
  3:  h();  
}

h() {  
  5:  f();  
  6:  i();  
}

```
Handling cycles in the call graph

• Long story short: iterate until a fixed point is reached
• It can take a while for naïve solutions ...

• **Strongly connected components:**
  A directed graph is called strongly connected if there is a path in each direction between each pair of vertices of the graph
Handling cycles in the call graph

To reach the fixed point faster:

① Identify strongly-connected-components (SCC)
② do{
  For each SCC in SCCs:
    Iterate among functions within SCC
    Iterate among every node in the call graph
  } while (anyChange);
Indirect calls

```c
void foo (int a, int (*p_to_f)(int v))
{
    int l = (*p_to_f)(5);
    a = l + 1;
    return a;
}
```

- How can we identify indirect calls in LLVM?

Is \( l \) constant?
Indirect calls

```c
void foo (int a, int (*p_to_f)(int v)){
    int l = (*p_to_f)(5);
    a = l + 1;
    return a;
}
```

- How can we identify indirect calls in LLVM?
- How can we handle indirect calls?
Procedure cloning

• Step 1: clone a function
  • A new function is created that is the exact clone of another one with only one difference:
    The name of the clone function is different than the original function

  ```
  void f(int p){
    int x = p + 3;
    ...
  }
  ```

  ```
  void f_2(int p){
    int x = p + 3;
    ...
  }
  ```

  ```
  void f_2(void){
    int x = 10;
    ...
  }
  ```

  ```
  void foo(){
    f(7)
    f(7)
  }
  ```

  ```
  void foo(){
    f_2();
    f_2();
  }
  ```

• Step 2: specialize the clone for a particular set of callers
  • Create a customized version of procedure for particular call sites
  • Compromise between inlining and inter-procedural optimization
Procedure cloning

• Pros
  • Less code bloat than inlining
  • Recursion is not an issue (as compared to inlining)
  • Better caller/callee optimization potential (versus inter-procedural analysis)

• Cons
  • Still some code bloat (versus inter-procedural analysis)
  • May have to do inter-procedural analysis anyway
    e.g. Inter-procedural constant propagation can guide cloning
Example: transform functions with level $\geq 3$ to be not affected by $\text{rand}()$

\[
\text{myF0}() \{ \\
\quad \vdots \\
\quad v = \text{rand}() \\
\quad \vdots \\
\}
\]

\[
\text{myF0}'() \{ \\
\quad \vdots \\
\quad v = 1 \\
\quad \vdots \\
\}
\]

\[
\text{myF1}() \{ \\
\quad \vdots \\
\quad \text{myF0}() \\
\quad \vdots \\
\}
\]

\[
\text{myF1}'() \{ \\
\quad \vdots \\
\quad \text{myF0}'() \\
\quad \vdots \\
\}
\]

\[
\text{myF2}() \{ \\
\quad \vdots \\
\quad \text{myF1}() \\
\quad \vdots \\
\}
\]

\[
\text{myF2}'() \{ \\
\quad \vdots \\
\quad \text{myF1}'() \\
\quad \vdots \\
\}
\]

\[
\text{myF3}() \{ \\
\quad \vdots \\
\quad \text{myF2}() \\
\quad \vdots \\
\}
\]

\[
\text{myFx}() \{ \\
\quad \vdots \\
\quad \text{myF0}() \\
\quad \vdots \\
\}
\]

Specialization
Example: transform functions with level $\geq 3$ to be not affected by $\text{rand}()$

Ideas?

```c
4 int q (void){
5    return rand() % 10;
6 }
7 int p1 (void){
8    return q();
9 }
10 int p2 (void){
11    return p1();
12 }
13 int p3 (void){
14    return p2();
15 }
16 int p4 (void){
17    return p2() + p1();
18 }
19 int main (int argc, char *argv[]){
20    int t = p3();
21    int r = p1();
22    printf("%d\n", t + r + p4());
23    return 0;
24 }
```
Example: transform functions with level $\geq 3$ to be not affected by rand()
Example: transform functions with level $\geq 3$ to be not affected by `rand()`
Example: transform functions with level $\geq 3$ to be not affected by rand()

Checking if
- the callee is \texttt{rand()}
  - Substitute call \texttt{rand()} with 1

- The callee invokes another function \texttt{F2} at level – 1
  - Clone \texttt{F2}: \texttt{F2'}
  - Call \texttt{F2'} instead of \texttt{F2}
  - Make \texttt{F2'} not affected by \texttt{rand()}

```c
bool transformFunction (Module &M, Function &F){
  bool modified = false;
  std::vector<Instruction *> toDelete;
  errs() << "START " << F.getName() << "\n";

  /* Reduce the impact to F. */
  for (auto &B : F) {
    for (auto &I : B) {
      if (auto call = dyn_cast<CallInst>(&I)){
        /* Fetch the callee. */
        auto *callee = call->getCalledFunction();
        if (callee == NULL){
          continue ;
        }
      }
    }
  }
}
```
Example: transform functions with level $\geq 3$ to be not affected by rand()
Example: transform functions with level $\geq 3$ to be not affected by rand()

```cpp
/* The callee needs to be cloned. */
errs() << "Cloning " << callee->getName() << " from " << F.getName() << "\n";
ValueToValueMapTy VMap;
auto clonedCallee = CloneFunction(callee, VMap);
call->replaceUsesOfWith(callee, clonedCallee);
randomInfo[clonedCallee] = randomInfo[callee];

/* Recursive check the callees of the cloned function. */
modified != transformFunction(M, *clonedCallee);
```
Example: transform functions with level $\geq 3$ to be not affected by `rand()`
Another solution using function inlining

```cpp
myF0()
{
  ...
  v = rand()
  ...
}

myF0()
{
  ...
  v =
  ...
}

myF1()
{
  ...
  myF0()
  ...
}

myF1()
{
  ...
}

myF2()
{
  ...
  myF1()
  ...
}

myF2()
{
  ...
}

myF3()
{
  ...
  myF2()
  ...
}

myF3()
{
  ...
  v = 1
  ...
}

myFx()
{
  ...
  myF0()
  ...
}

myFx()
{
  ...
}
```
Another solution using function inlining

```c
bool runOnModule(Module &M) override {
    analyzeFunctions(M);
    bool modified = transformFunctions(M);
    analyzeFunctions(M);
    printStatus(M);
    return modified;
}
```

Previous inter-procedural analysis

Inter-procedural transformation
Another solution using function inlining

```cpp
bool transformFunctions (Module &M)
{
    bool modified = false;

    for (auto &F : M) {
        if (randomInfo[&F].r != invoked) {
            continue;
        }
        if (randomInfo[&F].level <= 2) {
            continue;
        }
        modified |= transformFunction(M, F);
    }

    return modified;
}
```
bool transformFunction (Module &M, Function &F) {
  std::vector<Instruction *> toDelete;
  errs() << "START " << F.getName() << "\n";

  /* Reduce the impact to F. */
  bool modified = false;
  bool inlined = false;
  for (auto &B : F) {
    for (auto &i : B) {
      if (auto call = dyn_cast<CallInst>(&i)) {
        /* Fetch the callee. */
        auto *callee = call->getCalledFunction();
        if (callee == NULL) {
          continue;
        }
        errs() << "Inflining " << callee->getName() << " to " << F.getName() << "\n";
        InlineFunctionInfo IFI;
        inlined |= InlineFunction(call, IFI);
        if (inlined) {
          modified = true;
          break;
        } else {
          errs() << " Failed to inline\n";
        }
      }
    }
  }
  deleteDeadCode(toDelete);
  errs() << "END " << F.getName() << "\n";
  return modified;
}
bool transformFunction (Module &M, Function &F) {
    std::vector<Instruction *> toDelete;
    errs() << "START " << F.getName() << "\n";

    /* Reduce the impact to F. */
    bool modified = false;
    bool inlined = false;
    for (auto &B : F) {
        for (auto &I : B) {
            if (auto call = dyn_cast<CallInst>(&I)) {

                /* Fetch the callee. */
                auto *callee = call->getCalledFunction();
                if (callee == NULL) {
                    continue;
                }

                /* Recursive inlining. */
                /* */
                if (inlined) {
                    transformFunction(M, F);
                }

                30 lines: Check the callee.
            }
        }
    }

    /* Delete instructions that are dead. */
    for (auto i : toDelete) {
        i->eraseFromParent();
    }

    5 lines: Recursive inlining.
    errs() << "END " << F.getName() << "\n";
    return modified;
}
Today’s compilers

• Most old compilers avoid inter-procedural analysis
  • It’s expensive and complex
  • Not beneficial for most classical optimizations
  • Separate compilation + inter-procedural analysis requires recompilation analysis
  [Burke and Torczon’93]
  • Can’t analyze library code

• When are inter-procedural analyses useful?
  • Pointer analysis
  • Constant propagation
  • Object-oriented class analysis
  • Security and error checking
  • Program understanding and re-factoring
  • Code compaction
  • Parallelization
  • Vectorization

Modern uses of compilers
Other trends

• **Cost** of only having intra-procedural passes is growing
  • More functions than in the past and they’re smaller (OO languages)
  • Modern machines demand precise information (memory op aliasing)

• **Cost** of inlining is growing
  • Code bloat degrades efficacy of many modern structures
  • Procedures are being used more extensively

• Programs are becoming larger

• **Cost of inter-procedural analysis is shrinking**
  • Faster/more parallel machines
  • Better methods
Always have faith in your ability

Success will come your way eventually

Best of luck!