Inter-procedural CAT

Simone Campanoni
simonec@eecs.northwestern.edu
Procedures

• 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 (typically across 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

- Computes 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

- 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;
else
    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?

\[ a = \text{id}(4); \quad b = \text{id}(5); \]

**Context-sensitive analysis**
- Computes 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**
- 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

```plaintext
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

```cpp
bool runOnFunction(Function &F) override {
  errs() << " Function \" << F.getName() << "\n";
  for (auto &B : F) {
    for (auto &I : B) {
      if (auto call = dyn_cast<CallInst>(&I)) {
        Function *callee = call->getCalledFunction();
        errs() << "  Callee " << callee->getName() << "\n";
        for (auto &B2 : *callee) {
          for (auto &I2 : B) {
            errs() << I2 << "\n";
          }
        }
      }
    }
  }
  return false;
}
```
DEMO
Using CallGraphWrappingPass

• Declaring your pass dependence

```cpp
void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<CallGraphWrapperPass>();
}
```

• Fetching the call graph

```cpp
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

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

• From node to callees

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

• From node to Function

```c++
auto calleeF = calleeNode->getFunction();
errs() << " " << calleeF->getName() << "\n";
```
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 did we do previously?

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-sensitive

• How can we handle procedure calls?
• Obvious idea: 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: if (v < 10)
  G: m = 1;
  else
  H: m = 2;
  I: return m;
}

• Problem of invalid paths: dataflow facts from one call site “tainting” results at other call site
• p analyzed with merge of dataflow facts from all call sites
• How to address this problem?
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;
}
Inlining

• Inlining
  • Use a new copy of a procedure’s CFG at each call site
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;
}

IN = {A}
IN = {A, G, H}
IN = {A, C, G, H}
IN = {A, C, G, H, H'}

• What did it change?
• Solutions?
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
Inter-procedural dataflow analysis
flow/context/path-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;
}

• Accuracy?
• Performance?
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: exponentially expensive
• 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;
    else
        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
Summary context: example

• Summary context: summarize effect of called procedure for callers

Accuracy? compared to
• Intra-procedural
• Inter-procedural
  flow-sensitive
• Inter-procedural
  flow/context sensitive

Summary: p doesn’t return a constant
Summary context: example 2

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

Accuracy? compared to
• Intra-procedural
• Inter-procedural
  flow-sensitive
• Inter-procedural
  flow/context sensitive

Summary: p returns
Constant 1 if parameter is < 10
Constant 2 otherwise
Designing an inter-procedural analysis

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

What to summarize

Sensitivity
e.g., context
Context sensitivity

• Simple solution: make a small number of copies of contexts (e.g., even just 1 per procedure) (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: p();
    2: p();
}

p() {
    3: q();
}

q() {
    ...
}
main() {
  1: p();
  2: p();
}

p() {
  3: q();
}

q() {
  ...
}
Fibonacci: context insensitive

```java
main() {
    1: fib(7);
}

fib(int n) {
    if n <= 1
        x := 0
    else
        2: y := fib(n-1);
        3: z := fib(n-2);
        x:= y+z;
    return x;
}
```
Fibonacci: context sensitive, stack depth 1

```java
main() {
  1: fib(7);
}

fib(int n) {
  if n <= 1
    x := 0
  else
    2: y := fib(n-1);
    3: z := fib(n-2);
    x := y+z;
  return x;
}
```
main() {
    1: fib(7);
}

fib(int n) {
    if n <= 1
        x := 0
    else
        2: y := fib(n-1);
        3: z := fib(n-2);
        x := y+z;
    return x;
}
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)
Other contexts

• More choices
  • Assumption sets
    What state (i.e., dataflow facts) hold at the call site?

• Combinations of contexts
  • e.g., Assumption set and object
Designing an inter-procedural analysis

How to propagate

Sensitivity
e.g., context

What to summarize

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

① Sensitivity of analysis

② Single compilation

③ Separate compilations

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

⑤ Final remarks
Inter-procedural analysis: summary contexts

• Compute summary information for each procedure
  • Summarize effect of called procedure for callers
  • Summarize effect of callers for called procedure

• Store summaries in database
  • Use later when optimizing procedures

• Pros
  • Concise
  • Can be fast to compute and use
  • Separate compilation is practical

• Cons
  • Imprecise if only have one summary per procedure
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?
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);
    ```

• Bottom-up
  • 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
  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
Example: identify functions that might get affected by `rand()`

How can we do it?
Example: identify functions that might get affected by rand()

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()

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()`
Example: identify functions that might get affected by rand()

```c
void tagFunctionsDirectlyAffected(Module &M) {
    for (auto &F : M) {
        /* Initialize the information about F. */
        randomInfo[F].r = TBC;
        randomInfo[F].level = 0;

        /* Analyze F. */
        for (auto &B : F) {
            for (auto &I : B) {
                if (auto call = dyn_cast<CallInst>(&I)) {
                    /* Analyze a call instruction included in F. */
                    Function *callee = call->getCalledFunction();
                    if (callee == NULL || (callee->getName() == "rand") ) {
                        randomInfo[F].r = invoked;
                        randomInfo[F].level = 0;
                    }
                }
            }
        }
    } return ;
}
```
Example: identify functions that might get affected by rand()

```c
bool runOnModule(Module &M) override {
  errs() << "Module " << M.getName() << "\n";
  errs() << " Identify functions affected directly\n";
  tagFunctionsDirectlyAffected(M);
  printStatus(M);

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

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

  return false;
}
```
Example: identify functions that might get affected by \texttt{rand()}.

```c++
void identifyFunctionsIndirectlyAffected (Module &M) {
    bool changed;
    do {
        changed = false;
        for (auto &F : M) {
            if (randomInfo[F].r != invoked){
                continue;
            }
        }
    } while (changed);
    return;
}
```
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;
Other 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)
Context-sensitivity of summaries

• None (zero levels of the call path )
  • Forward propagation: Meet information from all callers to particular callee
  • Side-effects: Use side-effect information for callee at all call-sites

• Call-site (one level of the call path )
  • Forward propagation: Label data-flow information with call-site
  • Side-effects: Affects alias analysis, which in turn affects side-effects
Context-sensitivity of summaries (2)

- k levels of call path (k-limiting)
  - Forward propagation: Label data-flow information with k levels of the call path
  - Side-effects: Affects alias analysis, which in turn affects side-effects
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?

```plaintext
f() {
  1:  g();
  2:  g();
  3:  h();
}
g() {
  4:  h();
}
h() {
  5:  f();
  6:  i();
}
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

void foo (int a, int (*p_to_f)(int))
int l = (*p_to_f)(5);
a = l + 1;
return a;

• How can we identify indirect calls in LLVM?

Is l constant?
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

• Procedure Cloning/Specialization
  • Create a customized version of procedure for particular call sites
  • Compromise between inlining and inter-procedural optimization

• 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 \texttt{rand()}

```c
myF0(){
    ...
    v = \texttt{rand()}
    ...
}

myF0’(){
    ...
    v = 1
    ...
}

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

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

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

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

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

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

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

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

```
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 $\text{rand}()$ 

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

Checking if
- the callee is “rand()”
  - Substitute call $\text{rand}()$ with “1”
- The callee invokes another function $F_2$ at level – 1
  - Clone $F_2$: $F_2'$
  - Invoke $F_2'$ instead of $F_2$
  - Make $F_2'$ not affected by $\text{rand}()$
Example: transform functions with level >=3 to be not affected by rand()

```c
/* Check the callee. */
if (callee->getName() == "rand"){
  errs() << "Changing invocations to \"rand\" from " << F.getName() << "\n";
  Value *constValue = ConstantInt::get(call->getType(), 1, true);
  call->replaceAllUsesWith(constValue);
  toDelete.push_back(call);
  modified = true;
  continue ;
}
if (randomInfo[callee].r != invoked) continue ;
if (randomInfo[callee].level >= randomInfo[F].level) 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()`

Does this inter-procedural transformation converge always?
Another solution using function inlining

```c
myF0()
{
    ...
    v = rand();
    ...
}

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

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

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

myFx()
{
    ...
    myF0();
    ...
}
```
Another solution using function inlining

```cpp
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;
                }

                /* Check the callee.
                 */
                if (callee->getName() == "rand") {
                    errs() << "Changing invocations to \"rand\" from " << F.getName() << "\n";
                    Value *constValue = ConstantInt::get(call->getType(), 1, true);
                    call->replaceAllUsesWith(constValue);
                    toDelete.push_back(call);
                    modified = true;
                    continue;
                }
                if (randomInfo[callee].r != invoked) continue;
                if (randomInfo[callee].level >= randomInfo[F].level) continue;

                /* The callee needs to be cloned.
                 */
                errs() << "Inlining " << callee->getName() << " to " << F.getName() << "\n";
                InlineFunctionInfo IFI;
                inlined |= InlineFunction(call, IFI);
                if (inlined) {
                    modified = true;
                    break;
                } else {
                    errs() << "Failed to inline\n";
                }
            }
        }
    }

    /* Delete instructions that are dead.
     */
    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;
                }
            }
        }
    }

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

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

    errs() << "END " << F.getName() << "\n";
    return modified;
}
Today’s compilers

• Most 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 is it useful?
  • Pointer analysis
  • Constant propagation
  • Object oriented class analysis
  • Security and error checking
  • Program understanding and re-factoring
  • Code compaction
  • Parallelization

Modern uses of compilers
Other trends

• Cost of only having intra-procedural passes is growing
  • More of them 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