Interprocedural Data-Flow Analysis

Y.N. Srikant
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560012

NPTEL Course on Compiler Design
Motivation for Interprocedural DFA

- All DFA and optimizations that we have studied so far are **intraprocedural**
  - are performed on one procedure at a time
  - assume that procedures invoked may alter all the “visible” variables
  - imprecise, conservative, but simple
- Interprocedural analysis operates across an entire program
  - makes information flow from caller to callee and vice-versa
Procedure *inlining* is a simple method to enable such information flow
- applicable only if target of a call is known
- not possible if call is via a pointer or is “virtual”

Interprocedural analysis in O-O languages can sometimes determine if the target of even a “virtual call” is “static”
- now, either a “static” call or inlining can be used

However, inlining should be applied with care
- increases memory footprint
Applications of Interprocedural Analysis

- Converting virtual method calls to static method calls
- Interprocedural pointer analysis helps in making “points-to” sets more precise
  - reaching definitions, available expressions etc., can now be computed with more precision
- Interprocedural analysis eliminates spurious data dependencies, interprocedural constant propagation makes loop bounds known
  - exposes more parallelism during parallelization
- Interprocedural analysis helps in detecting
  - lock-unlock pattern of critical regions
  - disable-enable of interrupts
  - SQL injection (lack of input validation in Web applications)
  - vulnerabilities due to buffer overflows (frequently, array bounds are not checked)
Call Graphs

- A call graph for a program is a set of nodes and edges such that
  - There is one node for each procedure
  - There is one node for each call site
  - If call site $c$ may call procedure $p$, then there is an edge $c \rightarrow p$

- C and Fortran make procedure calls directly by name
  - hence call target of each invocation can be determined statically
Call Graphs

- If the program includes a procedure parameter or a function pointer
  - target is not known until runtime
  - target may vary from one invocation to another
  - call site can link to many or even all procedures in the call graph (considering only return types of functions)

- Ex: virtual method invocations in C++/Java
  - calls through the base class pointer cannot be resolved till runtime
Example of Call Graph

```c
int (*fp) (int);
int  f1(int x) {
    if (x > 100) return (*fp)(x-1); // csite 1
    else    return x;
}
int  f2(int y) {
    fp = &f1; return (*fp)(y); // csite 2
}
void main() {
    fp = &f2;  (*fp)(200); // csite 3
}
```
Call Graph Example

Conservative call graph

Exact call graph
Analysis of Call Graph

- Presence of references or pointers to functions or methods
  - helps us in getting a static approximation of the values of all procedure parameters, function pointers, and receiver object types
- With interprocedural analysis
  - more targets can be discovered and new edges can be inserted into the call graph
- This iterative procedure is repeated until convergence is reached
Context Sensitivity

i = 9;
while (i >= 0) {
    t1 = test(100);   // call site 1
    t2 = test(200);   // call site 2
    t3 = test(300);   // call site 3
    val[i--] = t1 + t2 + t3;
}

int test (int v) {
    return (v*2);
}

- Function test is invoked with a constant in each of the call sites, but the value of the constant is context-dependent
- It is not possible to infer that t1, t2, and t3 are each assigned constant values (hence for val[i] as well) unless we recognize the context
- A naive analysis would infer that test can return 200, 400, or 600 from any of the three calls

A context-sensitive analysis returns 200, 400, and 600 for t1, t2, and t3 (resp.), and 1200 for val[i]
Context Insensitive Analysis

- Treat each call and return as `goto` operations
- Create a **super control flow graph**
  - contains all the normal intraprocedural control-flow edges
  - edge connecting each call site to the beginning of the procedure it calls
  - edge connecting return statement back to the call site
  - assignment statements to assign
    - each actual parameter to its corresponding formal parameter
    - the returned value to the receiving variable
- Apply standard analysis on the super CFG
- Simple, but imprecise, because a function is analyzed as a common entity for all its calls and only its input-output behaviour abstracted out
Super Control Graph and Context-Insensitive Analysis Example

\[ i = 9 \]

\[ \text{if } i < 0 \text{ goto L} \]

\[ v = 100 \text{ // call site 1} \]

\[ v = 200 \text{ // call site 2} \]

\[ v = 300 \text{ // call site 3} \]

\[ \text{retval} = v^2 \text{ // func test} \]

\[ v (B6): 100, 200, \text{ or } 300 \]

\[ t1 (B4): 200, 400, \text{ or } 600 \]

\[ t2 (B5): 200, 400, \text{ or } 600 \]

\[ t3 (B7): 200, 400, \text{ or } 600 \]

\[ \text{val}[i] (B7): 600, 800, 1000, 1200, 1400, 1600, \text{ or } 1800 \]
Call Strings

- In the previous example, we needed just the call site to distinguish among the contexts.
- In general, the entire call stack defines a calling context.
- The string of call sites in the call stack is known as the call string.
- We may choose to use the $k$ entries just below any call site in the stack to distinguish between contexts.
  - $k$-limiting context analysis
  - reduces precision and makes results more conservative.
  - We take each call string, follow the calls, and perform data flow analysis, replacing the parameters and result variables as we go up and down the call string.
k-limiting Call Strings

i = 9;
while (i >= 0) {
    t1 = f (100); // call site c1
    t2 = f (200); // call site c2
    t3 = f (300); // call site c3
    val[i--] = t1 + t2 + t3;
}
int f (int v) {
    return test (v); // call site c4
}
int test (int v) {
    return (v*2);
}

- There are 3 call strings to test: (c1,c4), (c2,c4), (c3,c4)
- The value of v in test does not depend on the last call site c4, but on the first element of each of the call strings
- In this case, 2-limiting context analysis is enough
Complete Call Strings

```c
i = 9;
while (i >= 0) {
    t1 = f (100); // call site c1
    t2 = f (200); // call site c2
    t3 = f (300); // call site c3
    val[i--] = t1 + t2 + t3;
}
int f (int v) {
    if (v > 101)
        return f (v-1); // call site c4
    else
        return test (v); // call site c5
}
int test (int v) {
    return (v*2);
}
```

- There are 3 call strings to test
  - (c1,c5), value returned is 200
  - (c2,c4,c4,...,c4,c5): c4 is repeated 100 times, value returned is 202
  - (c3,c4,c4,...,c4,c5): c4 is repeated 200 times, value returned is 202
- The value of v in test depends on the full call string
- In this case, k-limiting context analysis is not enough, for any k
Cloning-Based Context-Sensitive Analysis

Simple, context-insensitive analysis is enough on the cloned call graph

```c
i = 9;
while (i >= 0) {
    t1 = f1 (100); // call site c1
    t2 = f2 (200); // call site c2
    t3 = f3 (300); // call site c3
    val[i--] = t1 + t2 + t3;
}
int f1 (int v) {
    return test1 (v); // call site c4.1
}
int test1 (int v) {
    return (v*2);
}

int f2 (int v) {
    return test2 (v); // call site c4.2
}
int test2 (int v) {
    return (v*2);
}

int f3 (int v) {
    return test3 (v); // call site c4.3
}
int test3 (int v) {
    return (v*2);
}
```

Recursive programs cannot be handled
Summary-Based Context-Sensitive Analysis

- Each procedure is represented by a concise description ("summary") that encapsulates some observable behaviour of the procedure.
- In reaching definitions or available expressions analysis, the appropriate OUT sets of the "procedure end" blocks would serve the purpose.
- We now explain one method of doing such an analysis.
- Recursion can also be handled using fixpoint computation.
The Problem of Aliases

- \( b+x \) will change in B3 if \( y \) is an alias of either \( b \) or \( x \)
- How can aliases arise?
- Consider a procedure

```latex
\textbf{procedure} p(x,y)
```

and calls to \( p \): \( p(z,z) \) or a call of \( p(u,v) \) from another procedure \( q(u,v) \) but \( q \) is called as \( q(z,z) \).

```
\begin{align*}
a &= b+x \\
y &= c \\
d &= b+x
\end{align*}
```
Aliases

- In reaching definitions, it is conservative not to regard variables as aliases when in doubt
  - So, we do not kill definitions when in doubt
- But, in available expressions, it is exactly the opposite
  - In the above example, if \( b+x \) is to be available in B3, we must be *certain* that \( b \) and \( x \) are not aliases of \( y \)
  - If in doubt, we assume aliasing and kill \( b+x \)
Alias Analysis

- Assume a language with recursive procedures but no nesting of procedures
- Parameters are passed by reference

1. Rename variables in procedures (if necessary) so that all names are different
2. If there is a procedure \( p(x_1, x_2, \ldots, x_n) \) and a call \( p(y_1, y_2, \ldots, y_n) \), then set \( x_i \equiv y_i \), for all \( i \)
3. Take reflexive and transitive closure of \( \equiv \)
global g,h;
procedure main() {
  local i;
  g = ...; one(h,i);
}
procedure one(w,x) {
  x = ...;
  two(w,w); two(g,x);
}

procedure two(y,z) {
  local k;
  h = ...; one(k,y);
}

- **main**: \( h \equiv w, \ i \equiv x \)
- **one**: \( w \equiv y, \ w \equiv z, \ g \equiv y, \ x \equiv z \)
- **two**: \( k \equiv w, \ y \equiv x \)
- All variables are aliases of each other
Change Computation

- **change**[p]: a set of global variables and formal parameters of p, that might be changed during an execution of p. No aliasing is considered at this time.

- **def**[p]: a set of formal parameters of p and globals having explicit definitions within p (not including those defined because of procedure calls made within p).
Change Computation

- \( \text{change}[p] = \text{def}[p] \cup \text{A}[p] \cup \text{G}[p] \), where
- \( \text{A}[p] = \{a \mid a \text{ is a global variable or formal param of } p, \text{ such that, for some proc } q \text{ and integer } i, p \text{ calls } q \text{ with } a \text{ as the } i^\text{th} \text{ actual param and the } i^\text{th} \text{ formal param of } q \text{ is in } \text{change}[q] \} \)
- \( \text{G}[p] = \{g \mid g \text{ is a global in } \text{change}[q] \text{ and } p \text{ calls } q \} \)
- We use a simplified calling graph whose nodes are procedures. There is an edge from \( p \) to \( q \) if \( p \) calls \( q \) somewhere in the program.
Example for the set $A[p]$

```
procedure p(...)
{ call q(...,a,...)
  ...
}
```

$i^{th}$ actual parameter

```
procedure q(b_1,b_2,...,b_i,...,b_n)
{   ...
}
```

$i^{th}$ formal parameter and $b_i$ is in change[q]
Change Computation

- Input: A calling graph with a collection of procedures, \( p_1, p_2, \ldots, p_n \). If the calling graph is acyclic, then we assume that \( p_i \) calls \( p_j \) only if \( j < i \), otherwise, no assumptions.

- Output: \text{change}[p]

- It is assumed that \text{def}[p] is precomputed.
for each proc $p$ do change[$p$] = def[$p$];
while changes to any change[$p$] occur do {
    for $i$ = 1 to $n$ do {
        for each proc $q$ called by $p_i$ do {
            1. add any globals in change[$q$] to change[$p_i$]; // adding $G[p_i]$
            2. for each formal parameter $x$ ($j^{th}$) of $q$ do
                if $x$ is in change[$q$] then
                    for each call of $q$ by $p_i$ do
                        if $a$, the $j^{th}$ actual param of the call is a
                            global or formal parameter of $p_i$ then
                            add $a$ to change[$p_i$] // adding $A[p_i]$
            }
        }
    }
}
Alias Analysis Example

global g,h;
procedure main() {
    local i;
    g = ...; one(h,i);
}
procedure one(w,x) {
    x = ...;
two(w,w); two(g,x);
}

procedure two(y,z) {
    local k;
    h = ...; one(k,y);
}

- **main**: h ≡ w, i ≡ x
- **one**: w ≡ y, w ≡ z, g ≡ y, x ≡ z
- **two**: k ≡ w, y ≡ x
- All variables are aliases of each other
def(main) = \{g\} = \text{change(main)}, G(main) = \Phi 
def(two) = \{h\} = \text{change(two)}, G(two) = \Phi 
def(one) = \{x\} = \text{change(one)}, G(one) = \{h\}, \text{since} 
“one” calls “two”, h is a global and change(two) contains h 

Consider “two”. “two” calls “one” 
one(k, y) – actual params, k is local 
one(w,x) – formal params, x is in change(one) 
Therefore, A(two) = \{y\}, change(two) = \{h,y\} 

Consider “one”. “one” calls “two” twice 
two(w, w) – actual params 
two(y, z) – formal params, y is in change(two) 
Therefore, A(one) = \{w\} 
two(g, x) – actual params 
two(y, z) – formal params, y is in change(two) 
Therefore, A(one) = \{w,g\}, change(one) = \{w,g,h,x\} 

Consider “main”. “main” calls “one” 
one(h, i) – actual params, i is local 
one(w, x) – formal params, w is in change(one) 
Therefore, A(main) = \{h\}, change(main) = \{g,h\}
Use of Change Information in computing Available Expressions – Method 1

- Each procedure call is a separate basic block
- Method 1: B is a block for call to proc p
  - $a_{\text{gen}}[B] = \Phi$, for all proc call basic blocks
  - $a_{\text{kill}}[B]$: if a variable $b$ is in $\text{change}[p]$, then $b$ kills all expressions involving $b$ and its aliases
  - $a_{\text{gen}}$ and $a_{\text{kill}}$ for all other types of blocks are computed in the usual manner
  - Knowing $a_{\text{gen}}[B]$ and $a_{\text{kill}}[B]$ for proc call blocks, computing $\text{IN}[B]$ and $\text{OUT}[B]$ for all blocks in the whole procedure proceeds in the usual manner
Use of Change Information in computing Available Expressions – Method 2

- Compute IN and OUT for all blocks in all procedures as usual, after computing \( \text{a\_gen} \) and \( \text{a\_kill} \) for procedure calls as in method 1.

- \( \text{a\_out} \) at the return point from a procedure \( p \) can be taken as \( \text{a\_gen}[p] \) for a block with a call to \( p \) (with no aliases applied).
  - However, consider only those expressions in \( \text{a\_out} \) with all their variables in \( \text{change}[p] \).
  - We substitute actual params for the formal params and see what expressions are generated by the call.

- Without changing \( \text{a\_kill} \) for proc call blocks, computations of IN and OUT are repeated.

- This procedure is repeated until no changes occur.