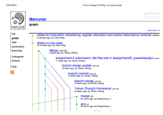


248 commits 3/5/2014

CSC 255/455

Beyond Dataflow: Interprocedural Analysis and CFL reachability

Instructor: Chen Ding

Reading:

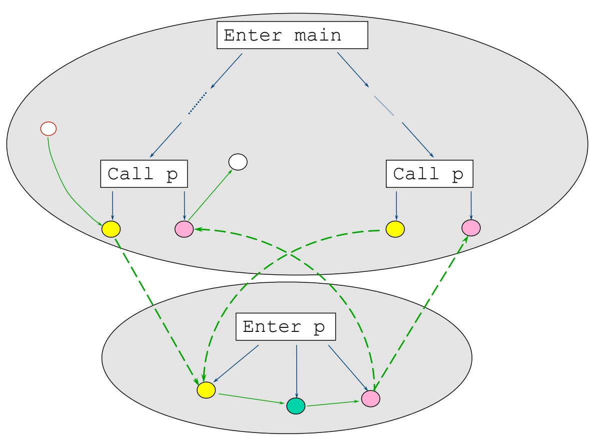
Problem classification: Allen-Kennedy book, Chapter 11 (11.2.1-2 required).

CFL-reachability: Reps' TR http://research.cs.wisc.edu/wpis/papers/tr1386.pdf

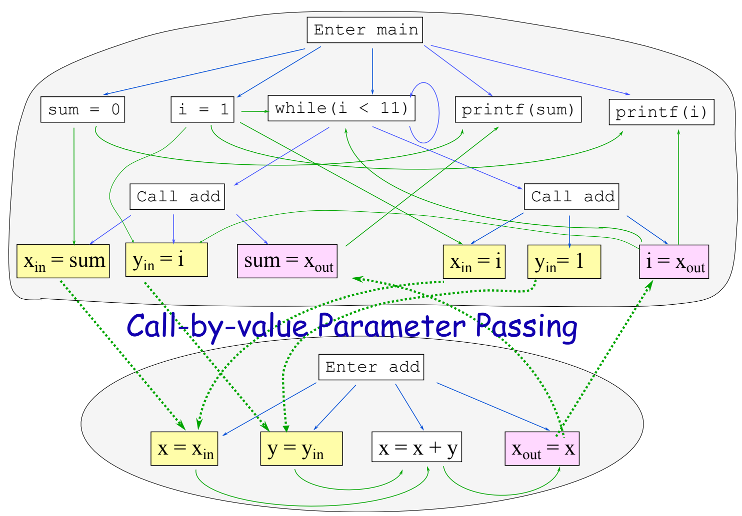
Introduction

- Interprocedural Analysis
  - Gathering information about the whole program instead of a single procedure
  - Do you know any such analysis?
- Interprocedural Optimization
  - Program transformation involving more than one function
  - Do you know a transformation?
- Other names of the procedure construct
  - function in C, subroutine in Fortran, method, lambda

Can we simply model procedure call/ return as control flow jumps?



Call-by-value Parameter Passing



Overview: Interprocedural Analysis

- Examples of Interprocedural problems
- Classification of Interprocedural problems
- Side-effect Analysis
- Jump functions in constant propagation
- Symbolic analysis
- Pointer analysis

Some Interprocedural Problems

- Modification and Reference Side-effect

```
COMMON X,Y
...
DO I = 1, N
S0:  CALL P
S1:  X(I) = X(I) + Y(I)
ENDDO
```

- Can S1 be parallelized?

Alias Analysis

```
SUBROUTINE S(A,X,N)
     COMMON Y
     DO I = 1, N
S0:       X = X + Y*A(I)
     ENDDO
END
```

- Could have kept X and Y in different registers and stored in X outside the loop
- What happens when there is a call, CALL S(A,Y,N)?
  - Then Y is aliased to X on entry to S
  - Can't delay update to X in the loop any more
- ALIAS(p,x): set of variables that may refer to the same location as formal parameter x on entry to p

Call Graph Construction

- Call Graph G=(N,E)
  - N: one vertex for each procedure
  - E: one edge for each possible call
    - Edge (p,q) is in E if procedure p calls procedure q
- Looks easy
- Construction difficult in presence of procedure parameters
- A call site
  - a control flow node inside the caller procedure
  - a procedure may have multiple call sites

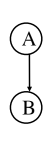
Live and Use Analysis

- Solve Live analysis using Use Analysis
- USE(s): set of variables having an upward exposed use in procedure p called at s
- At a call site, s is in a single basic block(b), x is live if either
  - x in USE(s) or
  - P doesn't assign a new value to x and x is live in some control flow successor of b

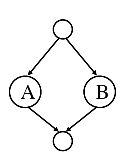
Interprocedural Problem Classification

- May and Must problems
  - MOD, REF and USE are 'May' problems
  - KILL is a 'Must' problem
- Flow sensitive and flow insensitive problems
  - Flow sensitive: control flow info important
  - Flow insensitive: control flow info unimportant

Flow Sensitive vs Flow Insensitive



R1



R2

$$MOD(R1) = MOD(A) \cup MOD(B)$$
$$MOD(R2) = MOD(A) \cup MOD(B)$$

$$REF(R1) = REF(A) \cup REF(B)$$
$$REF(R2) = REF(A) \cup REF(B)$$

$$KILL(R1) = KILL(A) \cup KILL(B)$$
$$KILL(R2) = KILL(A) \cap KILL(B)$$

$$USE(R1) = USE(A) \cup (\neg KILL(A) \cap USE(B))$$
$$USE(R2) = USE(A) \cup USE(B)$$

Classification (continued)

- A problem is flow insensitive iff solution of both sequential and alternately composed regions is determined by taking union of subregions
- Side-effect vs Propagation problems
  - MOD, REF, KILL and USE are side-effect
  - ALIAS, CALL and CONST are propagation

Flow Insensitive Side-effect Analysis

- Assumptions
  - No procedure nesting
  - All parameters passed by reference
  - Size of the parameter list bounded by a constant,
- MOD(s) can be solved in linear time
  - see the textbook

Constant Propagation

- Propagating constants between procedures can cause significant improvements.
- Dependence testing can be made more precise.

```
SUBROUTINE FOO(N)
   INTEGER N,M
   CALL INIT(M,N)
   DO I = 1,P
      B(M*I + 1) = 2*B(1)
   ENDDO
END
```

```
SUBROUTINE INIT(M,N)
   M = N
END
```

If N = 0 on entry to FOO, the loop is a reduction.
Otherwise, we can vectorize the loop.

Constant Propagation

- Definition: Let s = (p,q) be a call site in procedure p, and let x be a parameter of q. Then J_s^x, the jump function for x at s, gives the value of x in terms of parameters of p.

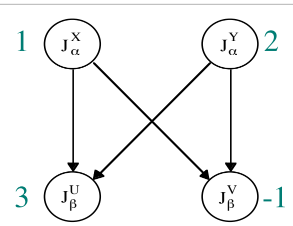
- The support of J_s^x is the set of p-parameters that J_s^x is dependent on.

Constant Propagation

- Instead of a Def-Use graph, we construct an interprocedural value graph:
  - Add a node to the graph for each jump function J_s^x
  - If x belongs to the support of J_t^y, where t lies in the procedure q, then add an edge between J_s^x and J_t^y for every call site s =(p,q) for some p.
- We can now apply the constant propagation algorithm to this graph.
  - Might want to iterate with global propagation

Example

```
PROGRAM MAIN
     INTEGER A,B
     A = 1
     B = 2
α    CALL S(A,B)
END
SUBROUTINE S(X,Y)
     INTEGER X,Y,Z,W
     Z = X + Y
     W = X - Y
β    CALL T(Z,W)
END
SUBROUTINE T(U,V)
     PRINT U,V
END
```



The constant-propagation algorithm will
Eventually converge to above values.
(might need to iterate with global)

Symbolic Analysis

- Prove facts about variables other than constancy:
  - Find a symbolic expression for a variable in terms of other variables.
  - Establish a relationship between pairs of variables at some point in program.
  - Establish a range of values for a variable at a given point.
    - Array section analysis

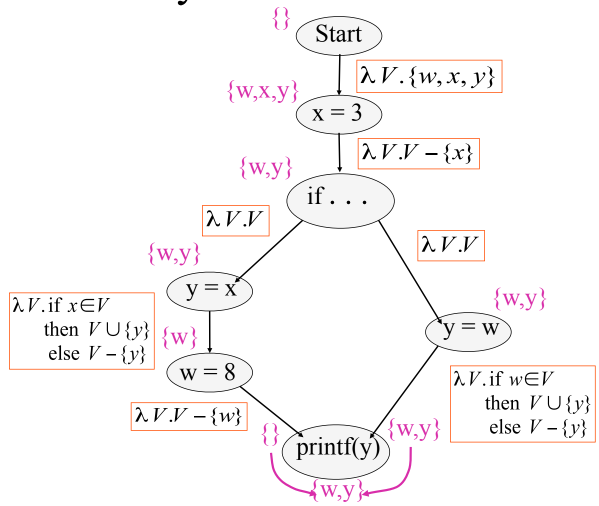
Thomas W. Reps

Professor
Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706-1685
USA

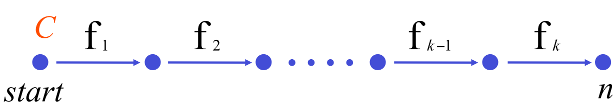
Program Analysis via CFL Reachability

http://research.cs.wisc.edu/wpis/papers/tr1386.pdf

Possibly Uninitialized Variables

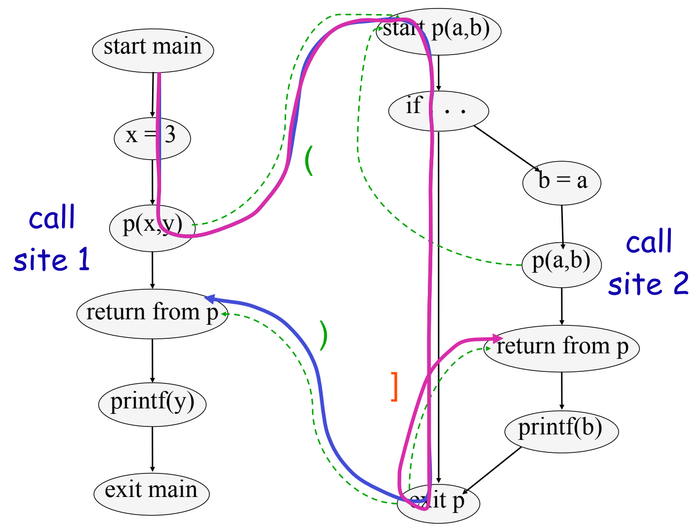


Precise *Intra*procedural Analysis

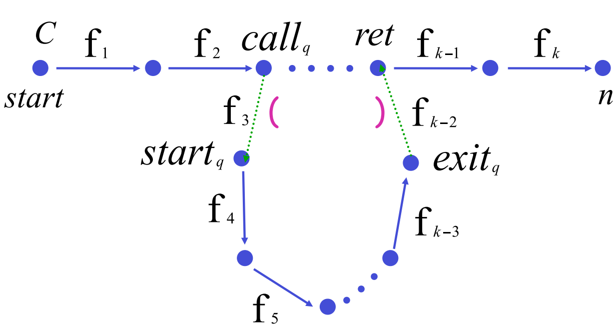


$$pf_p = f_k \circ f_{k-1} \cdots \circ f_2 \circ f_1$$

$$MOP[n] = \bigcup_{p \in \text{PathsTo}[n]} pf_p(C)$$



Precise *Inter*procedural Analysis



$$MOMP[n] = \bigcup_{p \in \text{MatchedPathsTo}[n]} pf_p(C)$$

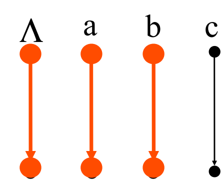
[Sharir & Pnueli 81]

Representing Dataflow Functions

Identity Function

$f = \lambda V.V$

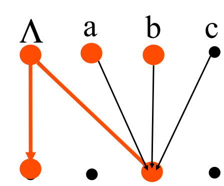
$\boxed{f(\{a,b\}) = \{a,b\}}$



Constant Function

$f = \lambda V.\{b\}$

$\boxed{f(\{a,b\}) = \{b\}}$



Representing Dataflow Functions

"Gen/Kill" Function

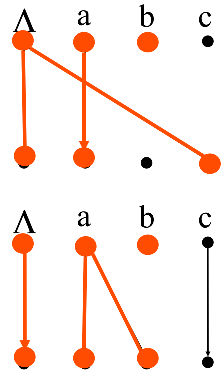
$$f = \lambda V.(V - \{b\}) \cup \{c\}$$

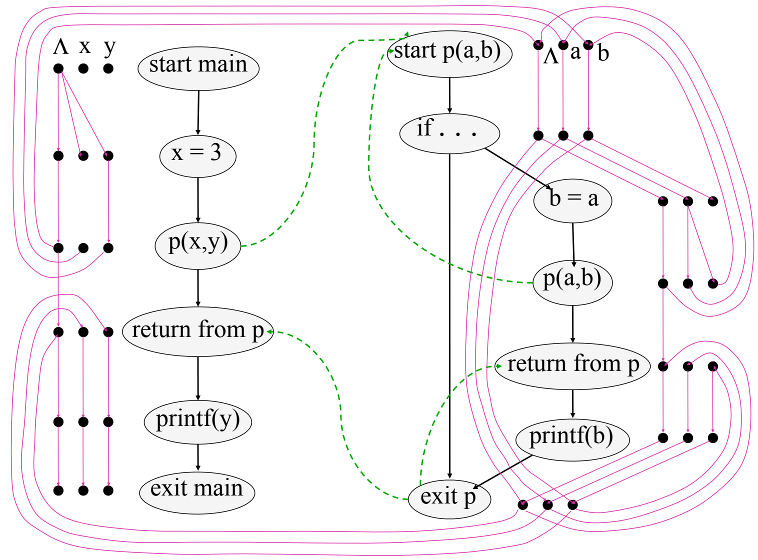
$$\boxed{f(\{a,b\}) = \{a,c\}}$$

Non-"Gen/Kill" Function

$$f = \lambda V.\text{if } a \in V \text{ then } V \cup \{b\} \text{ else } V - \{b\}$$

$$\boxed{f(\{a,b\}) = \{a,b\}}$$



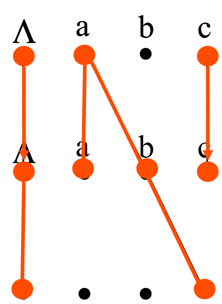


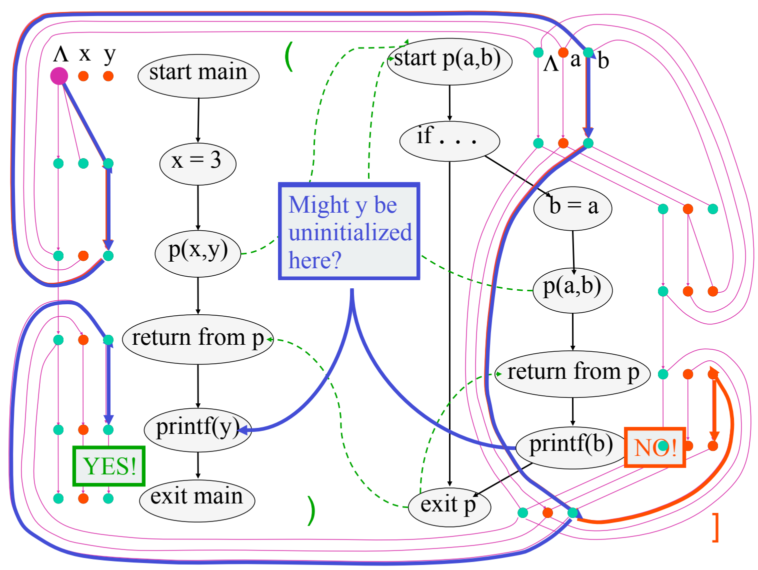
Composing Dataflow Functions

$$f_1 = \lambda V.\text{if } a \in V \text{ then } V \cup \{b\} \text{ else } V - \{b\}$$

$$f_2 = \lambda V.\text{if } b \in V \text{ then } \{c\} \text{ else } \phi$$

$$f_2 \circ f_1(\{a,c\}) = \boxed{\{c\}}$$





Exhaustive Versus Demand Analysis

- Exhaustive analysis: **All** facts at **all** points
- Optimization: Concentrate on inner loops
- Program-understanding tools: Only some facts are of interest

What Are Slices Useful For?

- Understanding Programs
  - What is affected by what?
- Restructuring Programs
  - Isolation of separate "computational threads"
- Program Specialization and Reuse
  - Slices = specialized programs
  - Only reuse needed slices
- Program Differencing
  - Compare slices to identify changes
- Testing
  - What new test cases would improve coverage?
  - What regression tests must be rerun after a change?

Line-Character-Count Program

```
void line_char_count(FILE *f) {
      int lines = 0;
      int chars;
      BOOL eof_flag = FALSE;
      int n;
      extern void scan_line(FILE *f, BOOL *bptr, int *iptr);
      scan_line(f, &eof_flag, &n);
      chars = n;
      while(eof_flag == FALSE){
            lines = lines + 1;
            scan_line(f, &eof_flag, &n);
            chars = chars + n;
      }
      printf("lines = %d\n", lines);
      printf("chars = %d\n", chars);
}
```

Character-Count Program

```
void char_count(FILE *f) {
      int lines = 0;
      int chars;
      BOOL eof_flag = FALSE;
      int n;
      extern void scan_line(FILE *f, BOOL *bptr, int *iptr);
      scan_line(f, &eof_flag, &n);
      chars = n;
      while(eof_flag == FALSE){
            lines = lines + 1;
            scan_line(f, &eof_flag, &n);
            chars = chars + n;
      }
      printf("lines = %d\n", lines);
      printf("chars = %d\n", chars);
}
```

Line-Character-Count Program

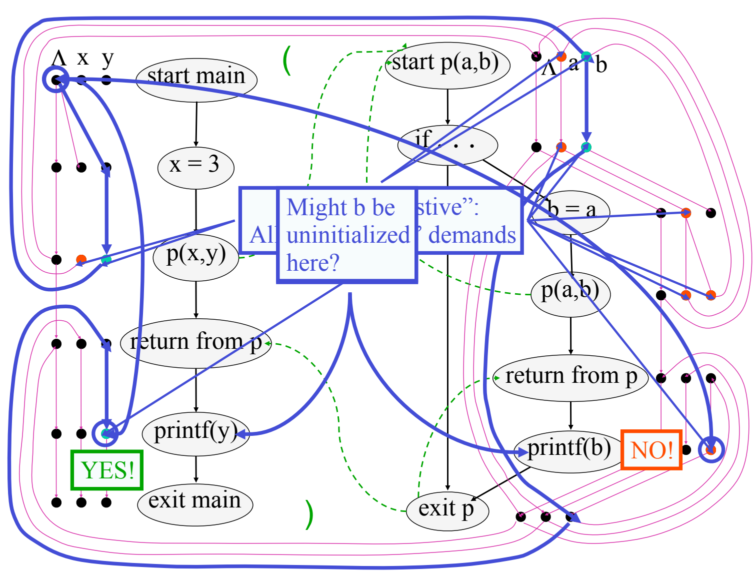
```
void line_char_count(FILE *f) {
      int lines = 0;
      int chars;
      BOOL eof_flag = FALSE;
      int n;
      extern void scan_line(FILE *f, BOOL *bptr, int *iptr);
      scan_line(f, &eof_flag, &n);
      chars = n;
      while(eof_flag == FALSE){
            lines = lines + 1;
            scan_line(f, &eof_flag, &n);
            chars = chars + n;
      }
      printf("lines = %d\n", lines);
      printf("chars = %d\n", chars);
}
```

Line-Count Program

```
void line_count(FILE *f) {
      int lines = 0;
      int chars;
      BOOL eof_flag = FALSE;
      int n;
      extern void scan_line2(FILE *f, BOOL *bptr, int *iptr);
      scan_line2(f, &eof_flag, &n);
      chars = n;
      while(eof_flag == FALSE){
            lines = lines + 1;
            scan_line2(f, &eof_flag, &n);
            chars = chars + n;
      }
      printf("lines = %d\n", lines);
      printf("chars = %d\n", chars);
}
```

Exhaustive Versus Demand Analysis

- Demand analysis:
  - Does a given fact hold at a given point?
  - Which facts hold at a given point?
  - At which points does a given fact hold?
- Demand analysis via CFL-reachability
  - single-source/single-target CFL-reachability
  - single-source/multi-target CFL-reachability
  - multi-source/single-target CFL-reachability



CFL-Reachability: Scope of Applicability

- Static analysis
  - Slicing, DFA, structure-transmitted dep., points-to analysis
- Verification
  - Security of crypto-based protocols for distributed systems [Dolev, Even, & Karp 83]
  - Model-checking recursive HFSMs
- Formal-language theory
  - CF-, 2DPDA-, 2NPDA-recognition
  - Attribute-grammar analysis

CFL-Reachability: Benefits

- Algorithms
  - Exhaustive & demand
- Complexity
  - Linear-time and cubic-time algorithms
  - PTIME-completeness
  - Variants that are undecidable
- Complementary to
  - Equations
  - Set constraints
  - Types
  - . . .