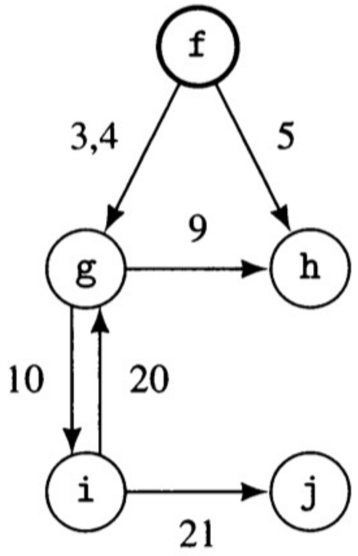
Interprocedural Analysis

- Why do we need this?
- We must conservatively assume that:
    1. a callee may use or change any variable it might be able to access,
    2. a caller can provide arbitrary values as parameters
- Intraprocedural analysis is fast, but the results are imprecise and conservative.
- Is procedure inlining always possible?
    - Virtual calls make it not possible
    - It also increases the memory footprint

Interprocedural Control-flow Analysis

- Deals with construction of program's call graph
- Given a program P with procedures p_1, p_2,..p_n, call graph $G = \langle N,S,E,r \rangle$
- N is $\{p_1, p_2,..p_n\}$
- S is set of call-site labels
- r is entry node
- $E \subseteq N \times S \times N$: An edge from (p_i, s_k, p_j) represents a call from p_i to p_j at site s_k.




```
1   procedure f( )
2   begin
3      call g( )
4      call g( )
5      call h( )
6   end   || f
7   procedure g( )
8   begin
9      call h( )
10     call i( )
11  end   || g
12  procedure h( )
13  begin
14  end   || h
15  procedure i( )
16     procedure j( )
17     begin
18     end   || j
19  begin
20     call g( )
21     call j( )
22  end   || i
```

Algorithm

```
LabeledEdge = Procedure × integer × Procedure

procedure Build_Call_Graph(P,r,N,E,numinsts)
   P: in set of Procedure
   r: in Procedure
   N: out set of Procedure
   E: out set of LabeledEdge
   numinsts: in Procedure ⟶ integer
begin
   i: integer
   p, q: Procedure
   OldN := ∅: set of Procedure
   N := {r}
   E := ∅
   while OldN ≠ N do
      p := ♦(N - OldN)
      OldN := N
      for i := 1 to numinsts(p) do
         for each q ∈ callset(p,i) do
            N ∪= {q}
            E ∪= {⟨p,i,q⟩}
         od
      od
   od
end    || Build_Call_Graph
```

Iterate over call sites
and add edges

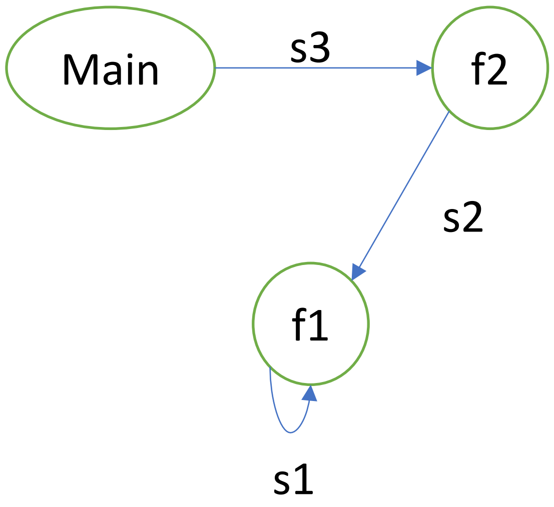
Call Graph Construction with Function Pointers

```
int (*fp)

int f1(int x) {
      if(x == 0) return x;
      return (*fp)(x-1);  // s1
}

int f2(int y) {
      fp = &f1;
      return (*fp)(y);     //s2
}

void main(){
     fp = &f2;
    (*fp)(10);             //s3
}
```



Context Sensitive vs. Context Insensitive

- Example: Interprocedural Constant Propagation
- Context Insensitive:
  - Does not distinguish between different call sites (call site independent)
  - For each procedure in a program, identifies subset of its parameter such that each parameter has the same constant value in every invocation.
- Context Sensitive:
  - Distinguishes between different call sites (call site dependent)
  - For each particular procedure called from each particular call site, the subset of parameters have the same constant values each time the procedure is called.

Interprocedural Constant Propagation

- Outline:
- The constant value of each formal argument is initialized to T.
- Compute the actuals of *a* call site *s* using the formals of a procedure *p*
- Compute the *meet* of the current values for the formals of callee *q* and the actuals at *s*
- Add *q* to the worklist if its constant values changed in the previous step

Jump Function

- A function that does all the computation required to compute the actual arguments to the callee in terms of the formal arguments of the caller.
- Jump function: $J(p,i,L,x)$
  - i - call site
  - p - caller procedure
  - L - formal arguments of caller
  - x - a formal parameter of the callee.

Example

```
        procedure e( )
        begin
e           x, c: integer
 1            c := f(x,1)
        end
        procedure f(i,j)
        begin
f           s, t: integer
 1            s := g(i,j)
 2            t := g(j,j)
 3            return s + t
        end
        procedure g(a,b)
g       begin
 1            a := 2
 2            b := b + a
 3            return a
        end
```

J(e,1,[],i) = $\perp$	Jsupport(e,1,[],i) = $\emptyset$
J(e,1,[],j) = 1	Jsupport(e,1,[],j) = $\emptyset$
J(f,1,[i,j],a) = i	Jsupport(f,1,[i,j],a) = {i}
J(f,1,[i,j],b) = j	Jsupport(f,1,[i,j],b) = {j}
J(f,2,[i,j],a) = j	Jsupport(f,2,[i,j],a) = {j}
J(f,2,[i,j],b) = j	Jsupport(f,2,[i,j],b) = {j}

Cval(i) = $\perp$
Cval(j) = 1
Cval(a) = $\perp$
Cval(b) = 1

Algorithm

```
procedure Intpr_Const_Prop(P,r,Cval)
   P: in set of Procedure
   r: in Procedure
   Cval: out Var ⟶ ICP
begin
   WL := {r}: set of Procedure
   p, q: Procedure
   v: Var
   i, j: integer
   prev: ICP
   Pars: Procedure ⟶ set of Var
   ArgList: Procedure × integer × Procedure
      ⟶ sequence of (Var ∪ Const)
   Eval: Expr × ICP ⟶ ICP
   || construct sets of parameters and lists of arguments
   || and initialize Cval( ) for each parameter
   for each p ∈ P do
      Pars(p) := ∅
      for i := 1 to nparams(p) do
         Cval(param(p,i)) := ⊤
         Pars(p) ∪= {param(p,i)}
      od
      for i := 1 to numinsts(p) do
         for each q ∈ callset(p,i) do
            ArgList(p,i,q) := []
            for j := 1 to nparams(q) do
               ArgList(p,i,q) ⊕= [arg(p,i,j)]
            od
         od
      od
   od
```

Initialize constant values for parameters

Initialize actual arguments at call sites

Algorithm

```
while WL ≠ Ø do
   p := ◆WL; WL -= {p}
   for i := 1 to numinsts(p) do
      for each q ∈ callset(p,i) do
         for j := 1 to nparams(q) do
            || if q( )'s jth parameter can be evaluated using values that
            || are arguments of p( ), evaluate it and update its Cval( )
            if Jsupport(p,i,ArgList(p,i,q),param(q,j)) ⊆ Pars(p) then
               prev := Cval(param(q,j))
               Cval(param(q,j)) ⊓= Eval(J(p,i,
                  ArgList(p,i,q),param(q,j)),Cval)
               if Cval(param(q,j)) ⊏ prev then
                  WL ∪= {q}
               fi
            fi
         od
      od
   od
od
end    || Intpr_Const_Prop
```

Take meet of the evaluated value and the previous value of parameter of a callee.

Precision of the Analysis

- The precision of the constant propagation will depend on the precision of J and Eval
- Examples:
  - Literal constant: If the argument passed is a constant, then a constant, else $\perp$
  - Pass-through parameter: If a formal parameter is directly passed or a constant, then pass the constant value, else $\perp$
  - Constant if intra-procedural constant.
  - Do a full-fledged analysis to determine its value.

Return-jump function

- Return-jump function: R(p, L)
  - p – procedure
  - L - formal parameters
  - Maps the formal parameters to the return value of the function.
  - If the language admits call-by references:
    R(p, L, x), where x - a formal parameter of the callee.
    Maps the value returned by the formal parameter x.

```
v = compute the meet of all the return values of p;
Set the return value of p to v;
foreach call function q that calls p do
      add q to the worklist
```

Algorithm

```
while WL ≠ Ø do
   p := ◆WL; WL -= {p}
   for i := 1 to numinsts(p) do
      for each q ∈ callset(p,i) do
         for j := 1 to nparams(q) do
            || if q( )'s jth parameter can be evaluated using values that
            || are arguments of p( ), evaluate it and update its Cval( )
            if Jsupport(p,i,ArgList(p,i,q),param(q,j)) ⊆ Pars(p) then
               prev := Cval(param(q,j))
               Cval(param(q,j)) ⊓= Eval(J(p,i,
                  ArgList(p,i,q),param(q,j)),Cval)
               if Cval(param(q,j)) ⊏ prev then
                  WL ∪= {q}
               fi
            fi
         od
      od
   od
od
end    || Intpr_Const_Prop
```

Take meet of the evaluated value and the previous value of parameter of a callee.

```
v = compute the meet of all the return values of p;
Set the return value of p to v;
foreach call function q that calls p do
      add q to the worklist
```