

Interprocedural Analysis

Class notes

**Program Analysis course given by Prof. Mooly Sagiv
Computer Science Department, Tel Aviv University
Fourth lecture given by Noam Rinetzky**

Liat Ramati and Dor Nir

Introduction

In the first three lectures we have been introduced to data flow analysis techniques. However, these techniques deal with languages without procedures and functions. Analyzing those languages is called **intraprocedural** analysis. Any programmer knows that in real life those languages are scarce and in most cases they are not applicable for humans (like assembler). This brings us to the need of extending our analysis techniques to languages with functions and procedures, or in other words- **interprocedural** analysis. **Interprocedural** analysis is more demanding and challenging than **intraprocedural** analysis; here we need to take into account the call-return and parameter passing mechanisms, local variable of the function and function recursion (can be unbounded). In some cases the called procedure is known only in run-time like with function pointers or with virtual functions. If we run in multithreaded environment, function concurrency can be a tough issue. If we are working with third party libraries, the source code is not always available. In contrast to dynamically allocated memory, the call-return mechanism is easier to track and thus better algorithmic solutions exist.

In order to simplify the problem, we assume:

- Parameters are passed by value – This means that the parameter value is being copied and a change to this parameter inside the function has no effect outside the function.
- No object oriented.
- No nesting functions – there is no definition of a function inside another function. This also means that we have only two kinds of variable scopes:
 - Local scope – the variable is known only inside the function and has no effect outside the function.
 - Global scope – the variable is known everywhere.
- No concurrency – No parallel execution of functions.

Reduction to intraprocedural analysis

The first kind of solutions will exploit the fact that we already know how to analyze programs with no function calls. We will try to perform reduction from a program with function calls to an equivalent program without any function calls and then to analyze it in the same manner we have up till now.

Trivial treatment of procedures

Analyze each procedure separately. After a function call continue with conservative information, global variables have unknown value.

Example: Here g is a global variable and x is local variable whose address is not taken (by means of $\&x$) when entering to the function p .

```
proc p()
{
    g := 1
}
[g → T]
[g → 1]

main()
{
    g := 2
    x := 3
    call p()
    print g
}
[g → T, x → T]
[g → 2, x → T]
[g → 2, x → 3]
[g → T, x → 3]
```

This technique is easy to implement, and procedures can be written in different languages since there is no dependency of information between the analyses. Although we still achieve soundness in this technique, this oversimplifying may not yield an analysis which is precise enough for our need. It seems that in every function call we loose a lot of information. Modular programming encourages small frequently called functions, which decrease the ability of the algorithm to give interesting results.

Procedure Inlining

Another way for reduction to intraprocedural analysis is to inline functions. This is a very simple technique but it doesn't handle recursion since it leads to unbounded in-lining even for recursion which is known to be bounded at runtime. Even for non-recursive programs the real problem with this technique is that it has an exponential blow up.

For example:

```
P1()
{
    [P1 staff]
    call P2
    [P1staff]
}

P2()
{
    [P2 staff]
    call P3
    [P2 staff]
}

P3
{
    ...[P3 staff]
}

call P2
call P3
```

```
}
}
```

Will be translated to:

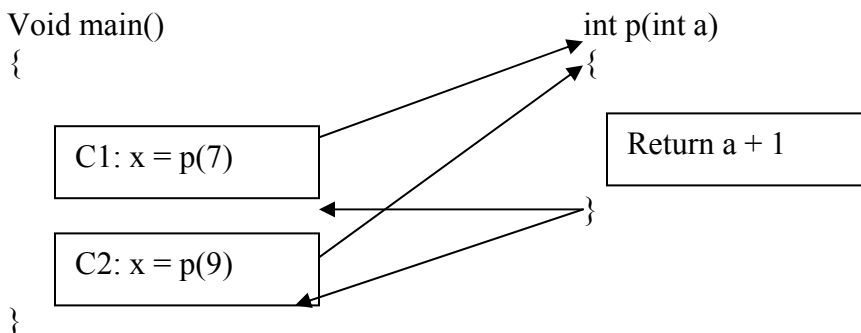
```
P1()
{
    [P1 staff]
    [P2 staff]
    [P3 staff]
    [P2 staff]
    [P3 staff]
    [P1 staff]
    [P2 staff]
    [P3 staff]
    [P2 staff]
    [P3 staff]
}
```

Naïve Interprocedural solution

A first attempt for an integrated Intra-Interprocedural analysis is the naïve solution in which we look at a global control flow graph, containing the control flow graphs of all procedures and:

- Treating all procedure calls like a goto from the call instruction to the first instruction of the procedure.
- Treating all return statements like a (non-deterministic) goto to the instruction following each call to the given procedure.

For example:

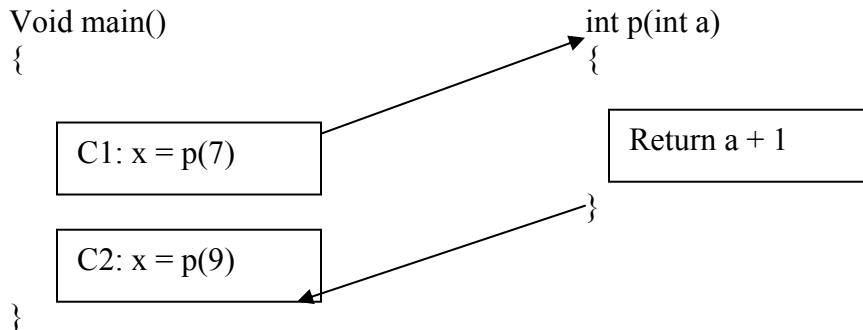


After the transformation we have a legitimate intraprocedural program where we can obtain conservative solution by applying the chaotic iteration algorithm and find the least fixed point using the following equations:

- $df_{entry}[s] = i$
- $df_{entry}[v] = \{ \bigsqcup \{ f(u, v) (df_{entry}[u]) \mid (u, v) \in E \}$

One problem with the naïve approach is that we allow impossible paths in our analysis and by that we loose accuracy.

For example we allow the following path:



Running the chaotic iteration algorithm for constant propagation on this example will result with non-useful information. As we are entering function `p` with two different parameters `a` will map to `T` and the function will return value will be `T`. From the function exit point we return to the two call points and this is why `x` is mapped to `T` in both places.

```

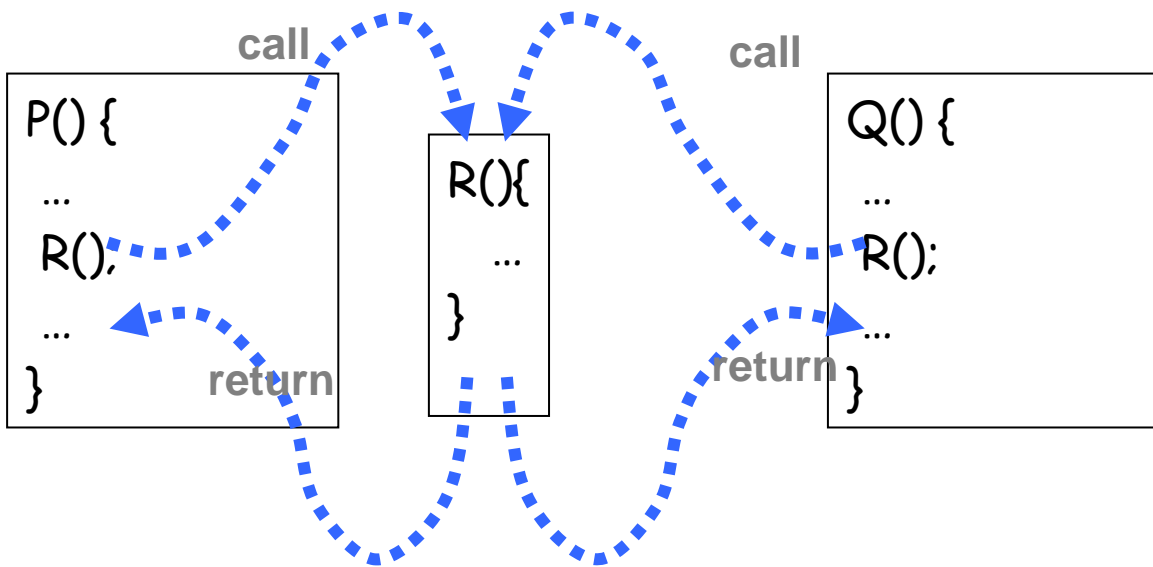
void main()
{
    int x ;
    x = p(7) ;
    [x → T]
    x = p(9) ;
    [x → T]
}

int p(int a)
{
    [a → T]
    return a + 1;
    [a → T, return → T]
}

```

Call String

In order to overcome the previous problem we want to be able to distinct between two different function calls and thus avoid invalid paths. In the call string approach we “remember” where we came from, i.e., the context of the call and where we return to by keeping a string that “simulates” the stack. The ideal solution is to keep track of the whole stack. A valid path is a control flow path that can be taken by respecting the stack regime.



Intuitively, this is a very natural concept: every return jumps to the instruction following the call that matches it as parenthesis match in a legal arithmetical expression. By adding the context of the call to the information of the step we can overcome the passing through invalid paths.

For example:

```
void main()
{
  int x ;
  c1: x = p(7);
  ε: [x → 8]
  c2: x = p(9) ;
  ε: [x → 10]
}
```

```
int p(int a)
{
  c1:[a →7]
  c2:[a →9]
  return a + 1;
  c1:[a →7, $$ →8]
  c2:[a →9, $$ →10]
}
```

Here we save the labels `c1` and `c2` in the state information of `p` so the variable `a` gets different values in each different call. Saving `c1` and `c2` help us with the return value, we know that when returning from `c1` the return value is 8 and for `c2` it's 10.

This algorithm is easy for implementation and efficient for small strings. If the string is too big we can save only constant number of strings and by that we limit the dept of function call. Using this technique with recursion is problematic since the same label can't distinct between two calls to the method. Because a recursion is unbounded we can use only constant depth of the string for which we loose accuracy.

In the following example, where we lose precision, we follow the first step of the analysis for a recursive procedure.

```
void main()
{
  c1: p(7);
  ε: [x → T]
}
```

```
int p(int a)
{
  c1: [a → 7]  c1.c2+: [a → T]
  if (...)
  {
    c1: [a → 7]  c1.c2+: [a → T]
    a = a - 1 ;
    c1: [a → 6]  c1.c2+: [a → T]
    c2: p (a);
    c1.c2*: [a → T]
    a = a + 1;
    c1.c2*: [a → T]
  }
  c1.c2*: [a → T]
  x = -2*a + 5;

  c1.c2*: [a → T, x→T]
}
```

In the example we can see that we use regular expressions to express sequence of C2. C2+ means a sequence of one or more C2s. C2* means a sequence of zero or more C2. This technique is easy to implement and efficient for small call strings. Sometimes finite domains can be precise even with recursion. However this technique guarantees only limited precision due to space limitation. Sometimes the order of calls can be abstracted by sets counting occurrences of call cites.

Functional approach

The reduction views the procedure in a degenerated fashion. It executes the functionality line by line, but the abstraction of the function idea was lost. In the functional approach we would like to compute the meaning or effect of the procedure and address the procedure as a transformation from input to output. We would like to see how the procedure maps input states to output states. The main idea is to recover the meaning of the procedure independent of the concrete values. For example, the meaning of a function $f(x)$ `{return x + 1 ;}` is to compute $x + 1$ for ever value of x without affecting any other variables.

The approach uses two phase algorithm:

- First phase: Compute the dataflow solution at the exit of a procedure as a function of the initial values at the procedure entry (functional values).
- Second phase: Compute the dataflow values at every point using the functional values. The second phase is similar to the standard intraprocedural analysis and uses the functional values computed in the first phase to compute the effect of procedure calls.

Here is an intuitive example. Note that a is local and x is global. In the first phase we compute the least fixed point of the function p by chaotic iteration.

```
int p(int a)
{
    [a →a0, x →x0]
    if (...)
    {
        a = a - 1 ;
        [a →a0-1, x →x0]
        p (a);
        [a →a0-1, x →T]
        a = a + 1;
        [a →a0, x →T]
    }

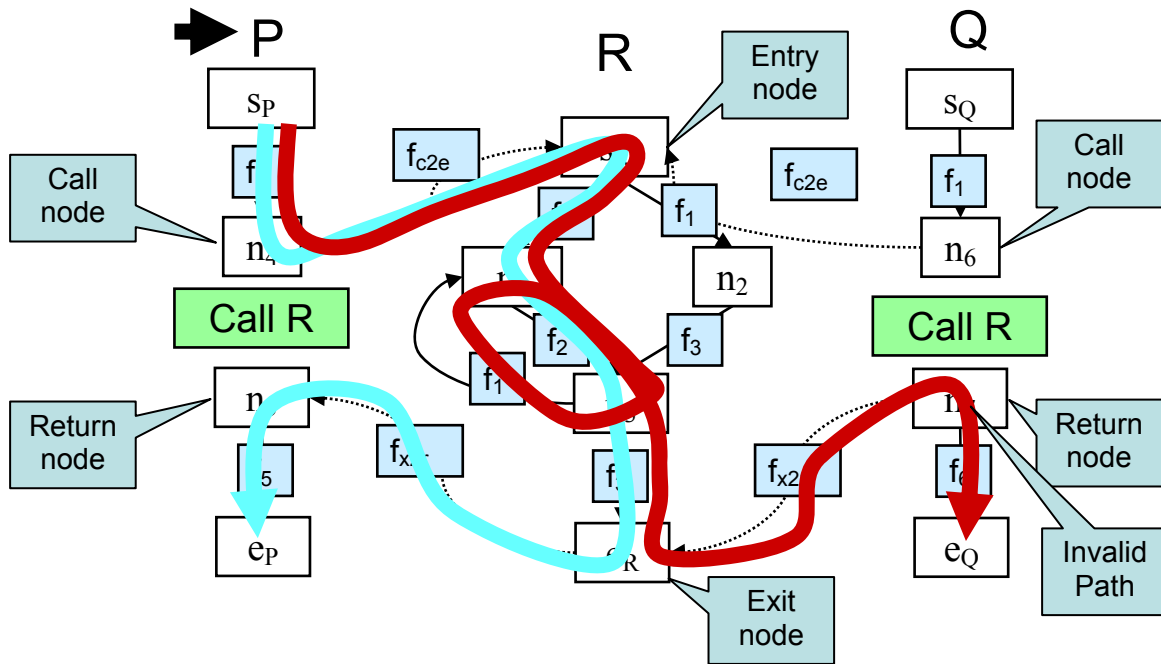
    [a →a0, x →T]
    x = -2*a + 5;
    [a →a0, x →-2a0 + 5]
}
```

When the chaotic iteration reaches the recursive call to P, we use the current function effect taken from the exit point of the procedure.

So here the function effect is: $p(a_0, x_0) = [a \rightarrow a_0, x \rightarrow -2a_0 + 5]$

In phase two we form again the chaotic iteration, but now when we reach the procedure call, we take the effect of the procedure.

We want to adopt the concept of JOP used in the intraprocedural analysis for interprocedural analysis. Note that in the latter, due to the call-return nature of the stack we will deal with invalid path as shown in the following graph.



Here the path marked in blue is valid meaning it conforms to the stack regime; however the red path is invalid as it starts in the call from P but returns to Q.

We want to formally define the set of all valid paths, and to show that it can be defined with a Context Free Grammar.

Arithmetical expressions and syntactical structure resemblance demand that we define valid paths using context free grammar. Our alphabet of terminal symbols will be composed of the union of the union of these two sets:

- $C = \{\text{the symbol } (c_i \mid c_i \text{ is a procedure call label})\}$
- $R = \{\text{the symbol })r_i \mid r_i \text{ is a procedure return label})\}$

The non terminal symbols are **{matched, valid}**

Assumption: For each i , r_i and c_i are matching call and return sites.

The following will be the derivation rules in our context free grammar

matched $\rightarrow e \mid \text{matched matched} \mid (c_i \text{ matched})r_i \mid$
for all matching c_i and r_i

valid \rightarrow **matched** | (c_i **valid** for all c_i in C)

Matched means that a path starts at a certain invocation instance and ends at the same invocation instance, i.e. every call is matched by the appropriate return. On the other hand, **valid** means that some of the calls were matched but others still not; we still have some frames in the stack to unwind. This only specifies the sequence of call/return labels of a valid path. The following Context Free Grammar rules describe the rest of the paths.

intra \rightarrow e | (l_i, l_j) **intra** for labels l_i, l_j that are neither call or return

matched \rightarrow e | **intra** | **matched matched** | (l_c, l_s) **matched** (l_e, l_r)

for all 4-tuples of labels l_c, l_r, l_s, l_e where l_c and l_r are matching call/return labels and l_s, l_e the starting/ending label of the procedure called in l_c .

valid \rightarrow **matched** | l_c **valid** for all procedure call labels l_c

A point to notice is that this Context Free Grammar defines too many paths. For example, the **intra** \rightarrow derivation rule doesn't constraint edges to be from labels that really have an edge between in the transfer graph, or even labels in the same procedure. But, it can easily be proved that the set of paths in the extended control flow graph presented in the previous section is a regular language. The intersection between that set and the Context Free language here defined is the set of valid paths. Because the intersection of a context free language and a regular language yields a context free language, we have that the set of valid paths can be defined by a context free language. The conclusion from this section is that the set of valid paths can be defined by a Context Free Grammar.

So now we can define the **Join Over all Valid Paths**.

For each sequence of edges e_1, e_2, \dots, e_n that constitute a valid path, define:

$f[e_1, e_2, \dots, e_n]: L \rightarrow L$ by composing the effects of the basic statements:

$f[](s) = s$ i.e., the empty path does not modify the state

$f[e, p](s) = f[p](f_e(s))$ i.e., apply the effect of the last statement to the overall effect of the previous statements

The Join over all Valid Paths, JVP, is defined by the following expression:

$JVP[n] = \sqcup \{f[e_1, e_2, \dots, e](i) \mid \text{for all valid paths } [e_1, e_2, \dots, e] \text{ ending in label } n\}$

Where i is the information known at the beginning of the program. ■

JVP at node n is defined to be the least upper bound on the set of abstract values that result by applying the abstract transformers along all *valid* paths that lead to n .

Now we can apply the join over all **valid** paths in the functional approach. When we move along the path from node N1 to node N2 we assemble the transition function.

As we mentioned above, the main idea of the functional approach is to **first** find out what is the overall effect of each function call and **second**, once we know the effect of every function, we can do chaotic iteration, and when we encounter a procedure call, we can use the knowledge acquired in the first phase about the actual effect of the call.

The advantage of this method is that it can compute the JVP, however, it requires that the transfer function be distributive.

Here is an example with constant propagation.

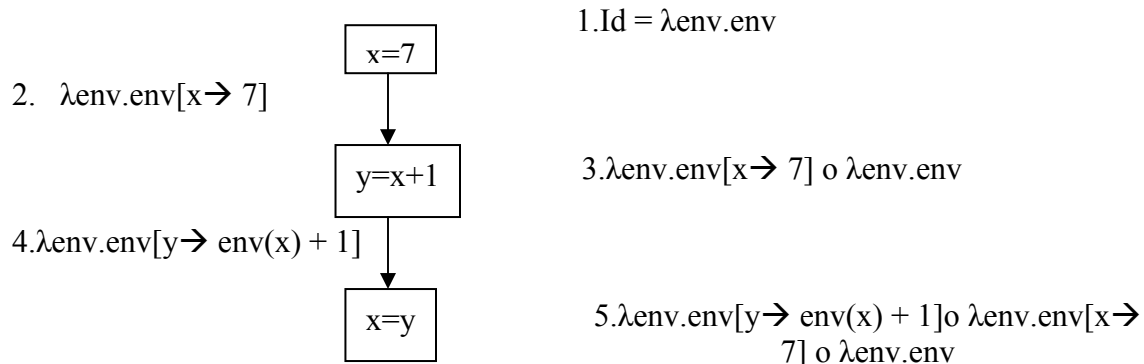
The lattice - $L = \text{Var} \rightarrow \mathbb{N} \cup \{T, \perp\}$

For each procedure we want to find a summary that gives us the net effect of the procedure on the lattice. Thus, we perform chaotic iterations over the domain of functions from L to L Domain - $F: L \rightarrow L$

Our functions are distributive i.e. $(f1 \sqcup f2)(x) = f1(x) \sqcup f2(x)$

We will use lambda notation.

Here is a simple example:



1. $\text{Id} = \lambda \text{env}.\text{env}$

This is the Id of the function that maps each environment to itself (the identity function). This is how we start.

2. $\lambda \text{env}.\text{env}[x \rightarrow 7]$

This is the function describes the effect of the statement $x=7$ on the state. The rest of the variables belong to this environment, stay the same.

3. $\lambda \text{env}.\text{env}[x \rightarrow 7] \circ \lambda \text{env}.\text{env}$

This is the function that describes the effect of the compound statement $x=7$ and the Id function. It is the composition of the transformers of each of the atomic statements.

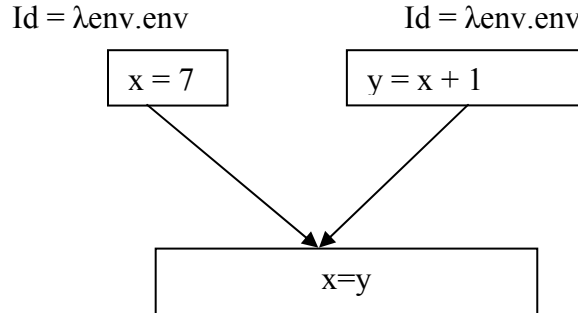
4. $\lambda \text{env}.\text{env}[y \rightarrow \text{env}(x) + 1]$

This is the function describes the effect of the statement $y=x+1$ on the state.

3 $\lambda \text{env}.\text{env}[y \rightarrow \text{env}(x) + 1] \circ \lambda \text{env}.\text{env}[x \rightarrow 7] \circ \lambda \text{env}.\text{env}$

This is the function that describes the effect of the compound statement of $y=x+1; x=7$ and the Id function. It is the composition of the transformers of each of the atomic statements.

In the following CFG the two statements preceded a join point:



$$\lambda \text{env}. \text{env}[y \rightarrow \text{env}(x) + 1] \sqcup \lambda \text{env}. \text{env}[x \rightarrow 7]$$

We join to compute the combined effect of both paths on the last statement:

$\lambda \text{env}. \text{env}[x \rightarrow 7]$

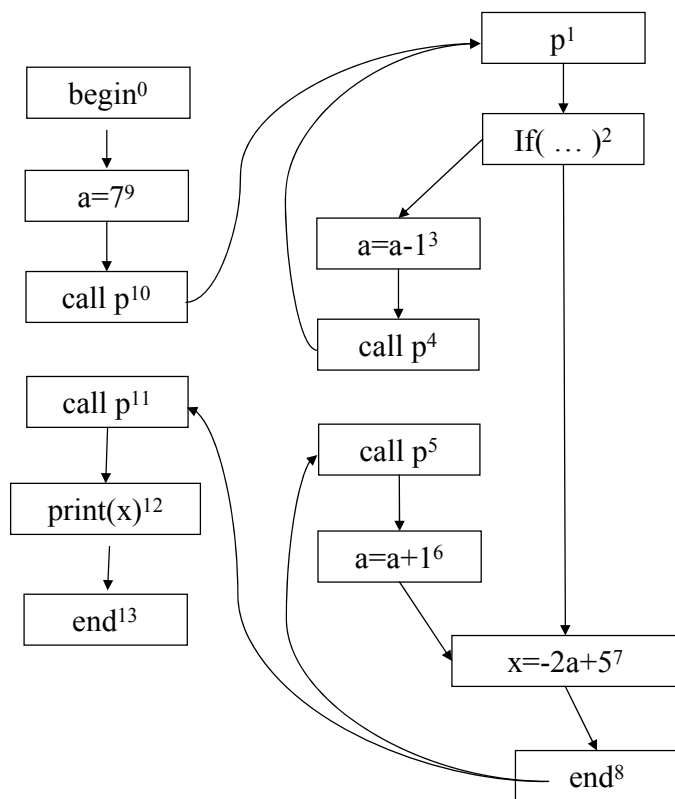
$\lambda \text{env}. \text{env}[y \rightarrow \text{env}(x) + 1]$

The algorithm works by chaotic iterations. Its input is a CFG whose edges are annotated with functions describing the effect of atomic. It begins by setting the function at the entry node to the procedure to be the identity function and at all other nodes we set it to \perp . In the first phase, the algorithm iterates over the graph from the leaves upwards, however, instead of propagating elements in L , it propagates functions in $L \rightarrow L$, modifying them according to the atomic statements of the graph edges. Whenever the algorithm reaches a call, it propagates to the return node the function at the call-site, combined with the function found at the procedure exit – which is the current estimation of the procedure net effect. In the second phase the algorithm performs the constant propagation using the functional values.

Example:

Initialization: each statement is indexed.

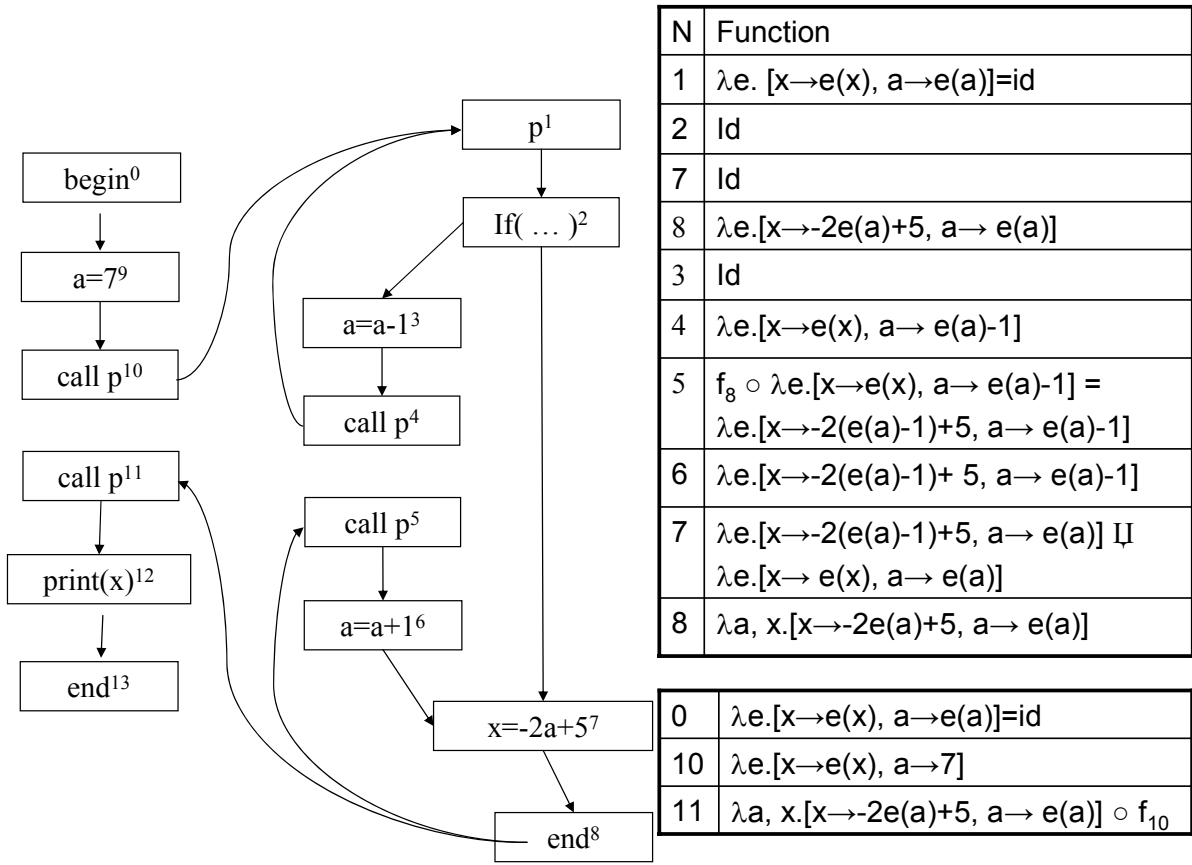
The procedures entries point as associated with the identity function.



N	Function
0	$\lambda e.[x \rightarrow e(x), a \rightarrow e(a)] = \text{id}$
1	$\lambda e.[x \rightarrow e(x), a \rightarrow e(a)] = \text{id}$
3-13	$\lambda e.\perp$

Phase 1:

We start from the leafs going upwards in the call graph. We compute for each procedure its net effect. Note that when we compute 5 we take into account the current effect of p – stored in the exit point 8.

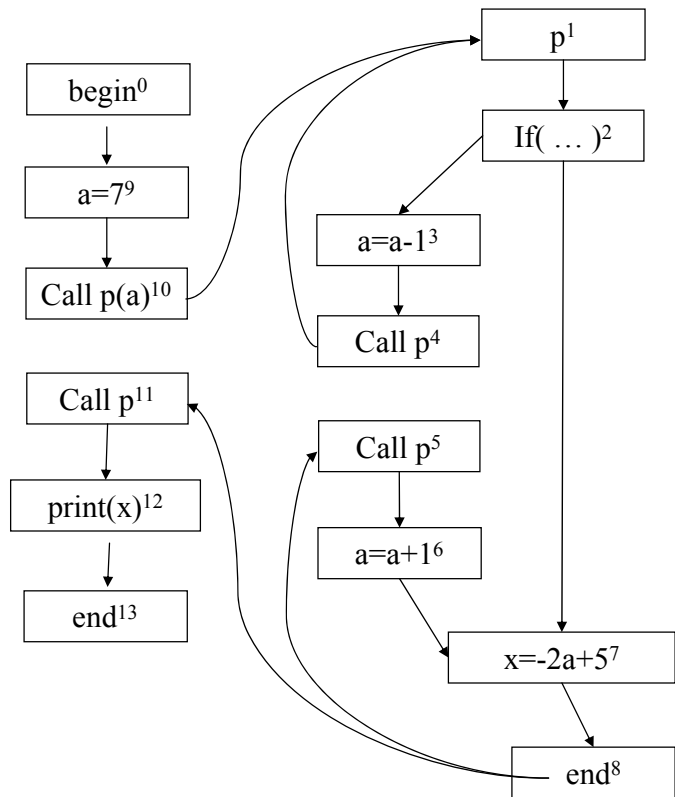


Phase2:

Now we proceed to do constant propagation analysis.

This phase proceeds as follows.

- The state at the entry of a procedure is the join of the states at all call sites.
- At return labels, apply the procedure transformer (found in the exit node of the procedure from the first iteration) to the abstract state at the call-site.
- To all other labels apply the label's transformer (found in the first iteration) to the initial state of the procedure they are in.



N	Function
1	[x→0, a→7]
2	[x→0, a→7]
7	[x→0, a→7]
8	[x→-9, a→7]
3	[x→0, a→7]
4	[x→0, a→6]
1	[x→-7, a→6]
6	[x→-7, a→7]
7	[x→T, a→7]
8	[x→-9, a→7]
1	[x→T, a→T]

0	[x→0, a→0]
10	[x→0, a→7]
11	[x→-9, a→7]

Interprocedural Distributive Finite Subset

The work “Precise Interprocedural Dataflow analysis via Graph Reachability” of Thomas Reps, Susan Horwitz and Mooly Sagiv shows how a large class of interprocedural dataflow-analysis problems can be solved precisely in polynomial time by transforming them into a special kind of graph-reachability problem. In the IFDS framework, a program is represented using a directed graph $G^* = (N^*, E^*)$ called a supergraph. For example:

```

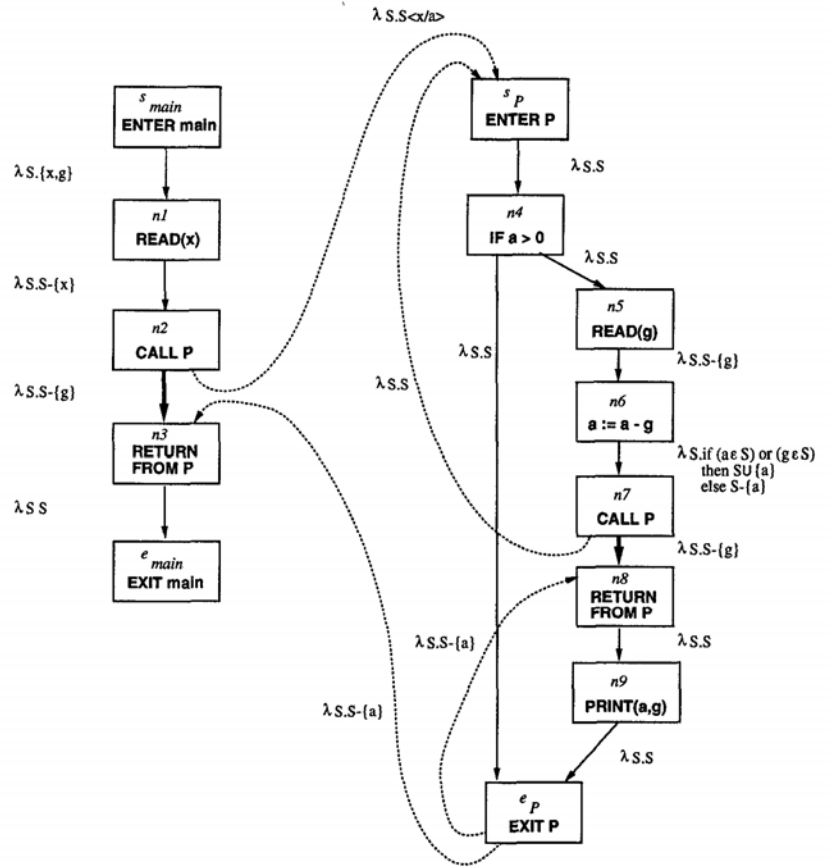
declare g: integer

program main
begin
  declare x: integer
  read(x)
  call P(x)
end

procedure P(value a: integer)
begin
  if (a > 0) then
    read(g)
    a := a - g
    call P(a)
    print(a, g)
  fi
end

```

(a) Example program



(b) Its supergraph G^*

The novel idea in this paper is in the way distributive functions are being represented. The representation relation of function f is defined as follows:

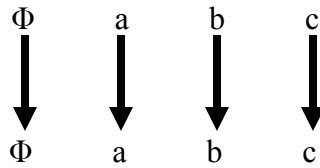
$$\begin{aligned}
 R_f =_{df} & \{ (\mathbf{0}, \mathbf{0}) \} \\
 & \cup \{ (\mathbf{0}, y) \mid y \in f(\emptyset) \} \\
 & \cup \{ (x, y) \mid y \in f(\{x\}) \text{ and } y \notin f(\emptyset) \}.
 \end{aligned}$$

For example:

The Identity function will be represented like this:

$$f = \lambda V.V$$

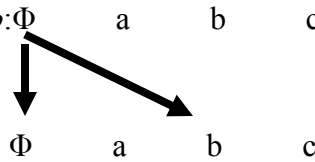
$$f(\{a, b\}) = \{a, b\}$$



And constant function that maps each variable to b :

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

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

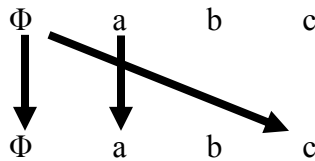


Here we can see that because each group contains the empty group, it's enough to have one arrow that goes from the empty group to b .

“Gen/Kill” function: remove b and add c .

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

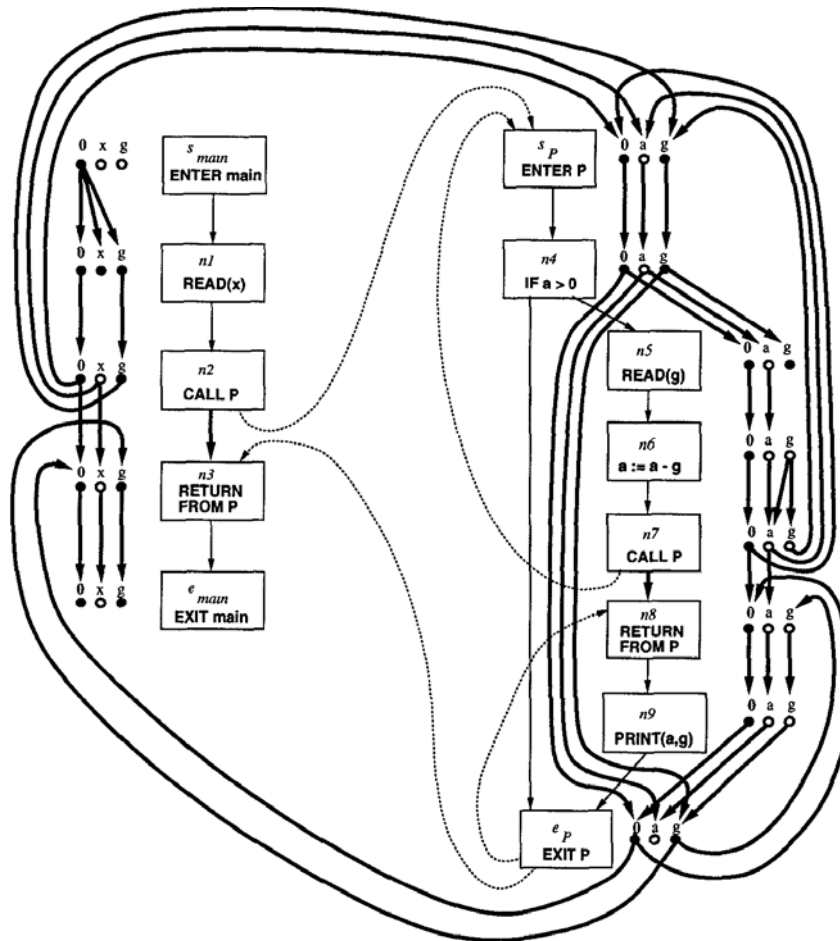
$$f(\{a, b\}) = \{b\}$$



Here we can see that because each group contains the empty group, it's enough to have one arrow that goes from the empty group to c . We need the arrow from a to a will be in the result set if it was in the input set.

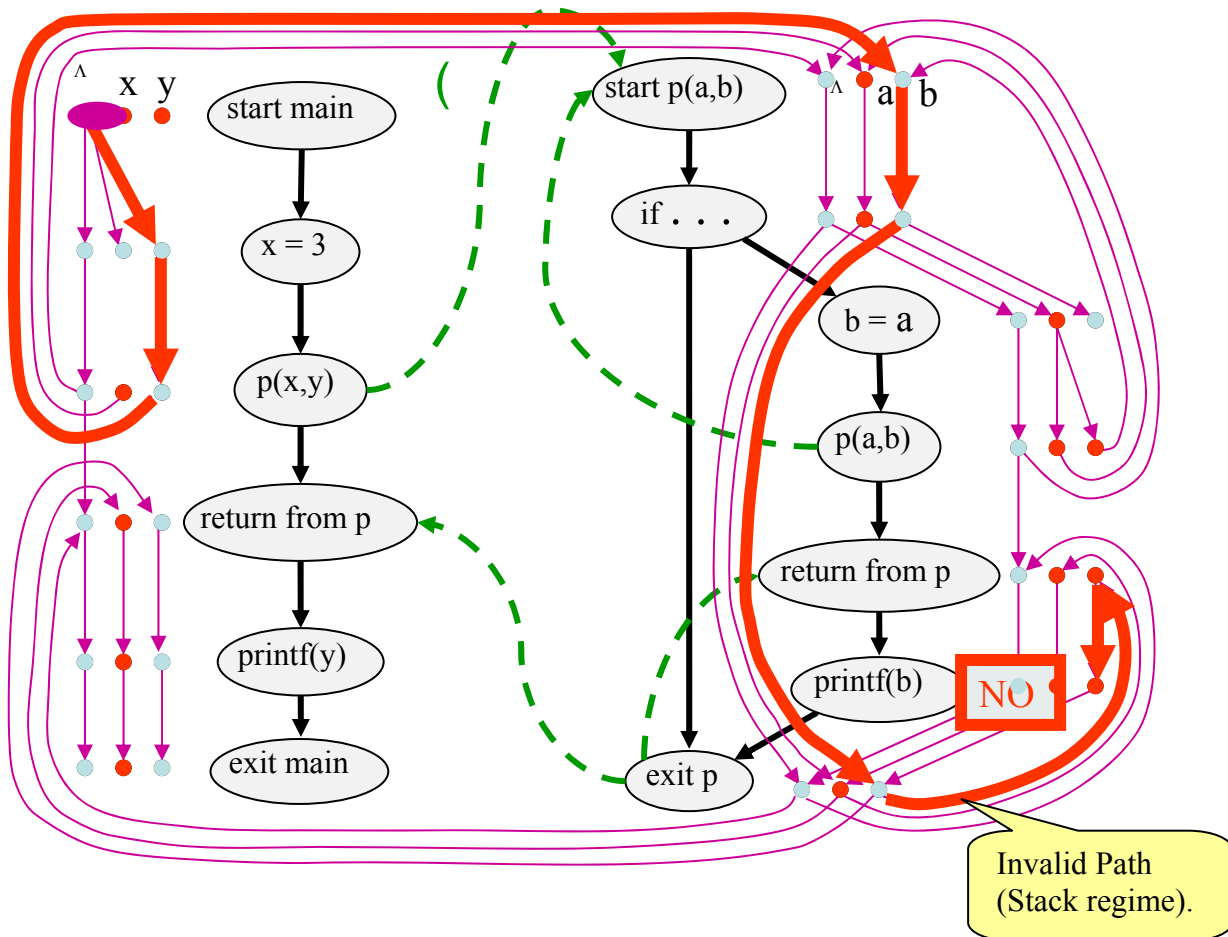
Using the dataflow functions we build the graph $G\#$. The edges of the $G\#$ corresponded to the representation relations of the dataflow functions on the edges of G^* .

For example if we take graph G^* and build the corresponding $G\#$ graph for the “possibly-uninitialized variables” problem, we will get the following graph:



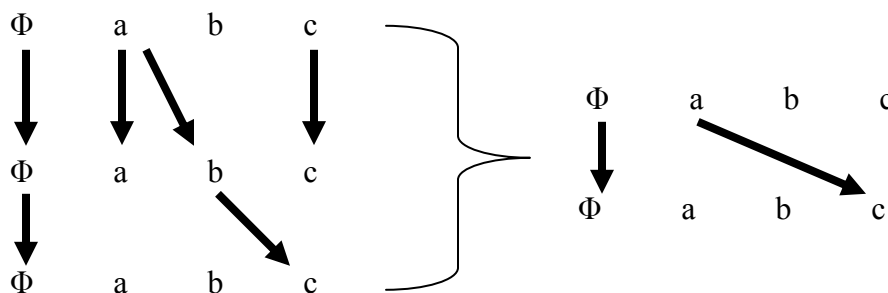
Because of the relationship between function composition and paths in composed representation-relation graphs, the path problem can be shown to be equivalent to IP. i.e. dataflow-fact d holds at supergraph node n if and only if there is a “realizable path” (valid path) from a distinguished node in $G\#$ to the node in $G\#$ that represents fact d at node n .

In the previous $G\#$ graph for the “possibly-uninitialized variables” problem we can see that closed circles represent nodes that are reachable along realizable paths from $\langle S_{main}, 0 \rangle$. Open circle represent unreachable nodes along realizable paths. For example node $\langle n8, g \rangle$ is reachable only along non-realizable paths from $\langle S_{main}, 0 \rangle$ (the red path in the following graph).



The running time of his algorithm is $O(ED^3)$.
 Typically: $E \approx N$, hence $O(ED^3) \approx O(ND^3)$.
 In the "Gen/Kill" problems: $O(ED)$.

Under certain conditions it is possible to combine two dataflow functions, which reduce the number of nodes and edges.



Conclusions

Interprocedural analysis is a very important feature of abstract interpretation as most of the real world programs include procedure calls. However there are several difficulties involved. The naïve solution that suggest few reductions to intraprocedural analysis such as inline, using ‘gotos’ and labeling or even call string all enjoy the more simpler nature of analysis when there is one procedure, but since this is only a reduction from a much more difficult problem we have high price to pay – either in exponential blow up and no support in recursion (inline) or lost of precision (call string) etc. The Join over all Valid Paths is a lower limit on the result we can reach.

The functional approach may yield an optimal solution (JVP) but there are several issues that must be resolved. It may work in cases where the transfer functions are distributive, closed under composition and join and can be efficiently represented, computed and checked for equality.