# Graph Representation of Code

Yulei Sui

University of Technology Sydney, Australia

# SVF : Static Value-Flow Analysis Framework for Source Code

**A scalable, precise and on-demand** interprocedural program dependence analysis framework for both sequential and multithreaded programs.

- The SVF project
  - **Publicly available** since early 2015 and actively maintained: `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 10.0.0) with over 100 KLOC C/C++ code and **530+ stars with 30+ contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, and awarded an **ICSE 2018 Distinguished Paper**, an **SAS Best Paper 2019** and an **OOPSLA 2020 Distinguished Paper**.

# SVF : Static Value-Flow Analysis Framework for Source Code

**A scalable, precise and on-demand** interprocedural program dependence analysis framework for both sequential and multithreaded programs.

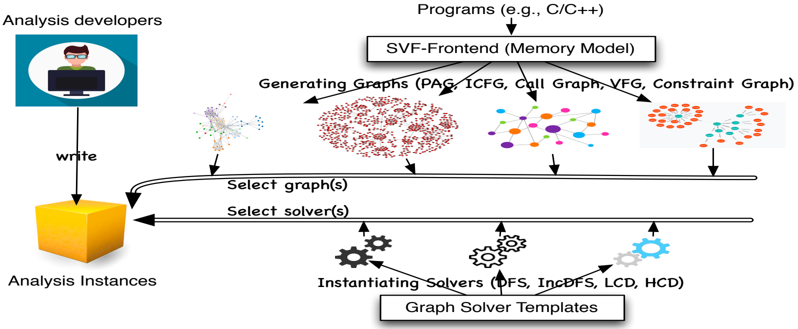
- The SVF project
  - **Publicly available** since early 2015 and actively maintained: `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 10.0.0) with over 100 KLOC C/C++ code and **530+ stars with 30+ contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, and awarded an **ICSE 2018 Distinguished Paper**, an **SAS Best Paper 2019** and an **OOPSLA 2020 Distinguished Paper**.
- Value-Flow Analysis: resolves **both control and data dependence**.
  - Does the information generated at program point $A$ flow to another program point $B$ along some execution paths?
  - Can function $F$ be called either directly or indirectly from some other function $F'$?
  - Is there an unsafe memory access that may trigger a bug or security risk?

# SVF : Static Value-Flow Analysis Framework for Source Code

**A scalable, precise and on-demand** interprocedural program dependence analysis framework for both sequential and multithreaded programs.

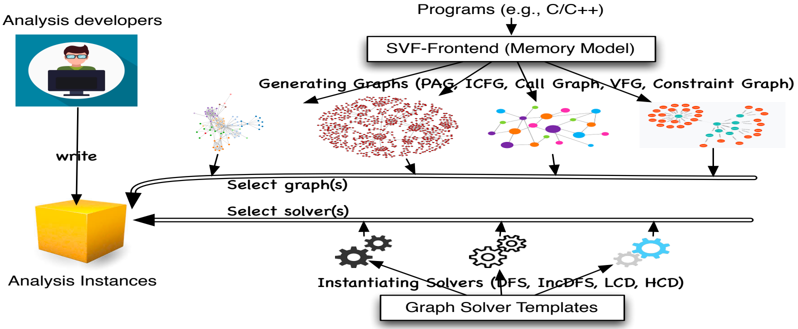
- The SVF project
  - **Publicly available** since early 2015 and actively maintained: `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 10.0.0) with over 100 KLOC C/C++ code and **530+ stars with 30+ contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, and awarded an **ICSE 2018 Distinguished Paper**, an **SAS Best Paper 2019** and an **OOPSLA 2020 Distinguished Paper**.
- Value-Flow Analysis: resolves **both control and data dependence**.
  - Does the information generated at program point $A$ flow to another program point $B$ along some execution paths?
  - Can function $F$ be called either directly or indirectly from some other function $F'$?
  - Is there an unsafe memory access that may trigger a bug or security risk?
- Key features of SVF
  - **Sparse**: compute and maintain the data-flow facts where necessary
  - **Selective** : support mixed analyses for precision and efficiency trade-offs.
  - **On-demand** : reason about program parts based on user queries.

# SVF: Design Principle




- Serving as an open-source foundation for building practical static source code analysis
  - Bridge the gap between research and engineering
  - Minimize the efforts of implementing sophisticated analysis (extendable, reusable, and robust via layers of abstractions)
  - Support developing different analysis variants (flow-, context-, heap-, field-sensitive analysis) in a sparse and on-demand manner.

# SVF: Design Principle




- Serving as an open-source foundation for building practical static source code analysis
  - Bridge the gap between research and engineering
  - Minimize the efforts of implementing sophisticated analysis (extendable, reusable, and robust via layers of abstractions)
  - Support developing different analysis variants (flow-, context-, heap-, field-sensitive analysis) in a sparse and on-demand manner.
- Client applications:
  - Static bug detection (e.g., memory leaks, null dereferences, use-after-frees and data-races)
  - Accelerate dynamic analysis (e.g., Google's Sanitizers and AFL fuzzing)

# Graph Representation of Code

- What is a graph representation of code?
  - Representing a program's control-flow (i.e., execution order) and/or data-flow (variable definition and use relations) using nodes and edges of a graph.

# Graph Representation of Code

- What is a graph representation of code?
  - Representing a program's control-flow (i.e., execution order) and/or data-flow (variable definition and use relations) using nodes and edges of a graph.
- Why a graph representation?
  - Abstracting code from low-level complicated instructions
  - Applying general graph algorithms
  - Easy to maintain and extend

# Call Graph

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1
%b1 = alloca i8, align 1
%a = alloca i8*, align 8
%b = alloca i8*, align 8
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q) #0
{
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

Program calling relations between methods

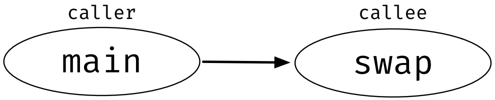

Call Graph

https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#3-call-graph

# Call Graph

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1
%b1 = alloca i8, align 1
%a = alloca i8*, align 8
%b = alloca i8*, align 8
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q) #0
{
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

- each node represents a program method
- each edge represents a calling relation between two program methods




Call Graph

https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#3-call-graph

# Control Flow Graph

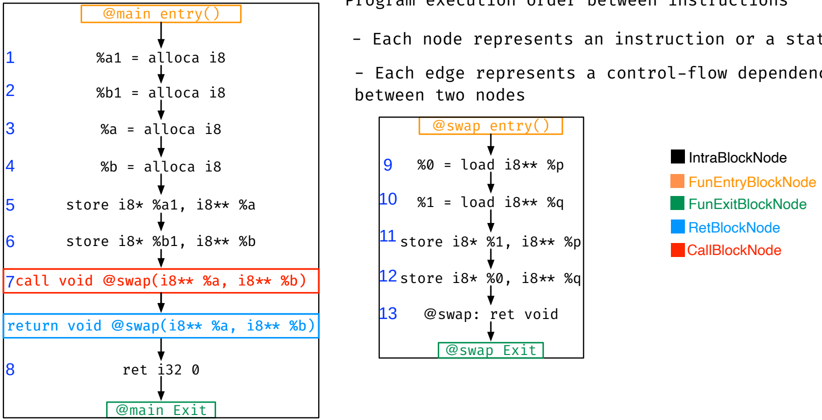
Program execution order between instructions.

- Intra-procedural control-flow graph: control-flow graph within a program method.
- Inter-procedural control-flow graph: control-flow graph across program methods.

# Intra-procedural Control Flow Graph

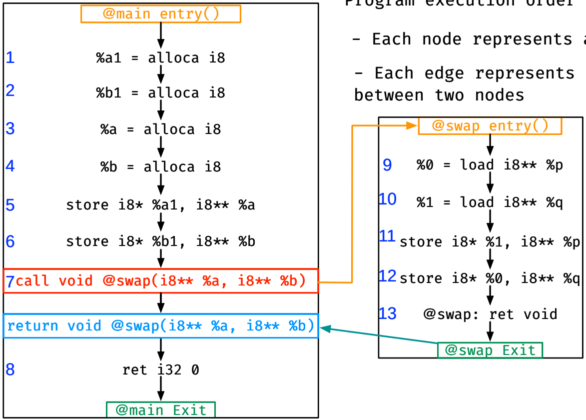
Program execution order between instructions

- Each node represents an instruction or a statement
- Each edge represents a control-flow dependence between two nodes



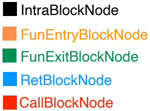

https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#4-interprocedural-control-flow-graph

# Inter-procedural Control Flow Graph (ICFG)




Program execution order between instructions

- Each node represents an instruction or a statement
- Each edge represents a control-flow dependence between two nodes




https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#4-interprocedural-control-flow-graph

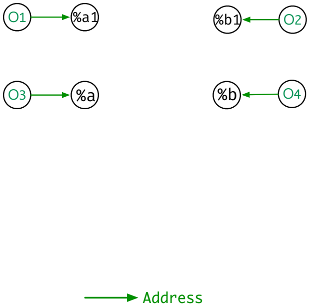
# Constraint Graph (or Program Assignment Graph)

- Constraint Graph represents the assignment constraints between variables at the instruction level.
- Constraint Graph and Program Assignment Graph (PAG) are essentially the same.
- The difference is that PAG can not be changed while you can add edges or nodes on the Constraint Graph to perform constraint solving.

# Constraint Graph (or Program Assignment Graph)

Program Assignment relation between two variables
- each node represent a pointer or an object
- each edge represents two nodes dependence or constraint relation

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1          O1
%b1 = alloca i8, align 1          O2
%a = alloca i8*, align 8          O3
%b = alloca i8*, align 8          O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
```




**alloca** instruction allocates typed integer
8 bytes of memory object as $O_1$ $O_2$ $O_3$ $O_4$

https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

# Constraint Graph

```
Program Assignment relation between two variables
   - each node represent a pointer or an object
   - each edge represents two nodes dependence or constraint relation
```

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        O1
%b1 = alloca i8, align 1        O2
%a = alloca i8*, align 8        O3
%b = alloca i8*, align 8        O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
```


`alloca` instruction allocates typed integer
8 bytes of memory object as $O_1$ $O_2$ $O_3$ $O_4$

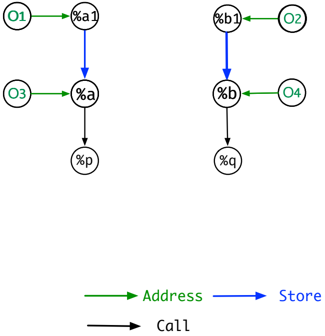
→ Address → Store

https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

# Constraint Graph

Program Assignment relation between two variables
- each node represent a pointer or an object
- each edge represents two nodes dependence or constraint relation

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        O1
%b1 = alloca i8, align 1        O2
%a = alloca i8*, align 8        O3
%b = alloca i8*, align 8        O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
```



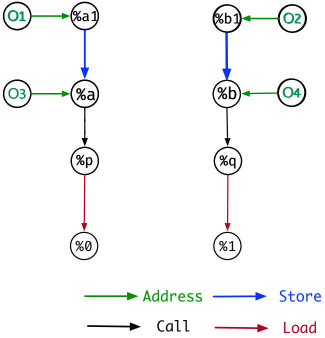

**alloca** instruction allocates typed integer
8 bytes of memory object as O1 O2 O3 O4

https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

# Constraint Graph

```
Program Assignment relation between two variables
   - each node represent a pointer or an object
   - each edge represents two nodes dependence or constraint relation
```

```
define void @swap(i8** %p, i8** %q) #0
{
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```



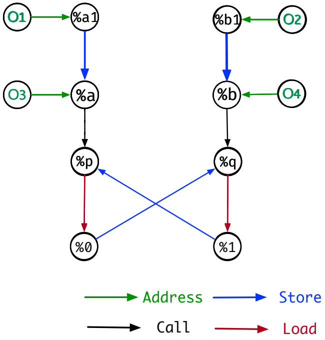

https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

# Constraint Graph

Program Assignment relation between two variables

- each node represent a pointer or an object
- each edge represents two nodes dependence or constraint relation

```
define void @swap(i8** %p, i8** %q) #0
{
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```




https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

# What's next?

- (1) Compile two C programs (`swap.c` and `example.c`) into their LLVM IR.
  - A guide can be found here https://github.com/SVF-tools/SVF-Teaching/wiki/CodeGraph#2-llvm-ir-generation
  - Understand the mapping from a C program to its corresponding LLVM IR.
- (2) Generate and visualize the graph representation of LLVM IR (`swap.ll` and `example.ll`).
  - https://github.com/SVF-tools/SVF-Teaching/wiki/CodeGraph#3-run-and-debug-your-codegraph
- (3) Write code to iterate nodes and edges of ICFG and PAG and print their contents.
  - https://github.com/SVF-tools/SVF-Teaching/blob/main/CodeGraph/CodeGraph.cpp#L65-L82
- (4) More about LLVM IR and SVF's graph representation
  - LLVM language manual https://llvm.org/docs/LangRef.html
  - SVF website https://github.com/SVF-tools/SVF

# Data-Dependence

Yulei Sui

University of Technology Sydney, Australia

# Data-Dependence

## Pointer analysis

- Points-to Analysis: Statically determine the possible runtime values of a pointer at compile-time.
- Alias Analysis: determine whether two pointer dereferences refer to the same memory location.
- e.g., p = &a; p = q;
- p and q both point to a. *p and *q are aliases.

# Data-Dependence

## Pointer analysis

Why do we need to learn pointer analysis

- Pointer analysis tells us what memory locations code uses or modifies

# Data-Dependence

## Pointer analysis

Why do we need to learn pointer analysis

- Pointer analysis tells us what memory locations code uses or modifies
- Essential for building data-dependence relations between variables
    - p = &a; p = q; *p = x; y = *q

# Data-Dependence

## Pointer analysis

Why do we need to learn pointer analysis

- Pointer analysis tells us what memory locations code uses or modifies
- Essential for building data-dependence relations between variables
  - p = &a; p = q; *p = x; y = *q
- Good for program understanding, bug detection and compiler optimisations
  - Constant propagation
    - p = 1; *q = x; r = p;
    - r is a constant value and equals 1, if p and q do not alias each other, otherwise, can not perform constant propagation.

# Data-Dependence

## Pointer analysis

Why do we need to learn pointer analysis

- Pointer analysis tells us what memory locations code uses or modifies
- Essential for building data-dependence relations between variables
  - p = &a; p = q; *p = x; y = *q
- Good for program understanding, bug detection and compiler optimisations
  - Constant propagation
    - p = 1; *q = x; r = p;
    - r is a constant value and equals 1, if p and q do not alias each other, otherwise, can not perform constant propagation.
  - Taint analysis
    - p = taintedInput; x = *q;
    - x is tainted if p and q alias each other.

# Data-Dependence

## Pointer analysis

An ongoing research topic classified into the following precision dimensions.

- Flow-insensitive analysis:
  - Ignore program execution order
  - A single solution at each program point
- Flow-sensitive analysis:
  - Respect the program execution order
  - A Separate solution at each program point

# Data-Dependence

## Pointer analysis

An ongoing research topic classified into the following precision dimensions.

- Flow-insensitive analysis:
  - Ignore program execution order
  - A single solution at each program point
- Flow-sensitive analysis:
  - Respect the program execution order
  - A Separate solution at each program point
- Field-insensitive analysis:
  - Accessing a field of an object is treated as accessing the entire object.
- Field-sensitive analysis:
  - Distinguish the fields of an aggregate object (e.g., struct object).

# Data-Dependence

## Pointer analysis

An ongoing research topic classified into the following precision dimensions.

- Flow-insensitive analysis:
  - Ignore program execution order
  - A single solution at each program point
- Flow-sensitive analysis:
  - Respect the program execution order
  - A Separate solution at each program point
- Field-insensitive analysis:
  - Accessing a field of an object is treated as accessing the entire object.
- Field-sensitive analysis:
  - Distinguish the fields of an aggregate object (e.g., struct object).
- Context-insensitive analysis:
  - Merges all of its calling contexts together when analysing a program method
- Context-sensitive analysis:
  - Distinguishes between different calling contexts of a program method

# Andersen's Pointer analysis

- The most popular and widely used pointer analysis
- Constraint solving (inclusion-based constraints between program variables)

We will practice a flow-insensitive, field-insensitive and context-insensitive Andersen's analysis through analyzing the PAG (or Constraint Graph) of a program.

# Andersen's Pointer Analysis

SVF transforms LLVM instructions into a PAG (or Constraint Graph).

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

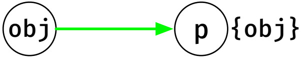
| Constraint Type | C code | Constraint rule |
|---|---|---|
| Address: | p = &obj | $\{obj\} \subseteq Pts(p)$ |
| Copy: | p = q | $Pts(q) \subseteq Pts(p)$ |
| Load: | p = *q | $\forall\ o \in Pts(q),\ Pts(o) \subseteq Pts(p)$ |
| Store: | *p = q | $\forall\ o \in Pts(p),\ Pts(q) \subseteq Pts(o)$ |

# Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a PAG (or Constraint Graph).

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

| Constraint Type | C code | LLVM IR | Constraint Graph |
|---|---|---|---|
| Address: | `p = obj` | `%p = alloca // obj` | |




pts(p) = {obj}

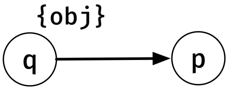
# Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a PAG (or Constraint Graph).

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

| Constraint Type | C code | LLVM IR |
|---|---|---|
| Address: | `p = obj` | `%p = alloca // obj` |
| Copy: | `p = q` | `%p = bitcast %q` |

Constraint Graph




`pts(p) = pts(q)`

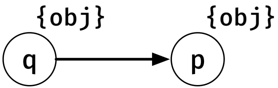
# Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a PAG (or Constraint Graph).

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

| Constraint Type | C code | LLVM IR |
|---|---|---|
| Address: | `p = obj` | `%p = alloca // obj` |
| Copy: | `p = q` | `%p = bitcast %q` |

Constraint Graph




pts(p) = pts(q)

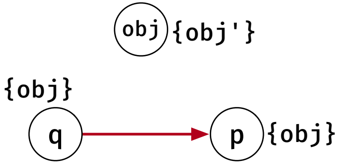
# Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a PAG (or Constraint Graph).

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

| Constraint Type | C code | LLVM IR |
|---|---|---|
| `Address:` | `p = obj` | `%p = alloca // obj` |
| `Copy:` | `p = q` | `%p = bitcast %q` |
| `Load:` | `p = *q` | `%q = load %p` |

`Constraint Graph`




`pts(p) = {obj}`

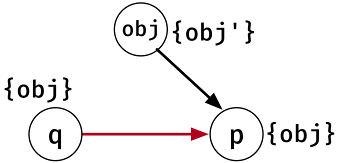
# Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a PAG (or Constraint Graph).

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

| Constraint Type | C code | LLVM IR |
|---|---|---|
| Address: | `p = obj` | `%p = alloca // obj` |
| Copy: | `p = q` | `%p = bitcast %q` |
| Load: | `p = *q` | `%q = load %p` |

Constraint Graph




`pts(p) = {obj}`

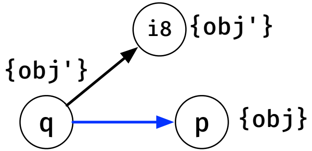
# Andersen’s Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a PAG (or Constraint Graph).

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

| Constraint Type | C code | LLVM IR |
|---|---|---|
| Address: | p = obj | %p = alloca // obj |
| Copy: | p = q | %p = bitcast %q |
| Load: | p = *q | %q = load %p |
| Store: | *p = q | store %q, %p |

Constraint Graph

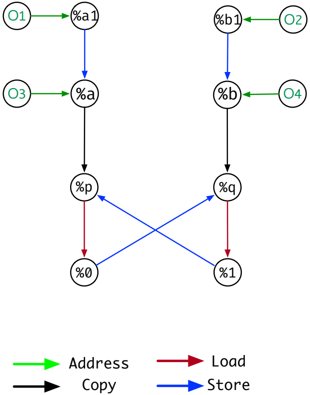



pts(p) = {obj'}
pts(p) = pts(q)

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1          // O1
%b1 = alloca i8, align 1          // O2
%a = alloca i8*, align 8          // O3
%b = alloca i8*, align 8          // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

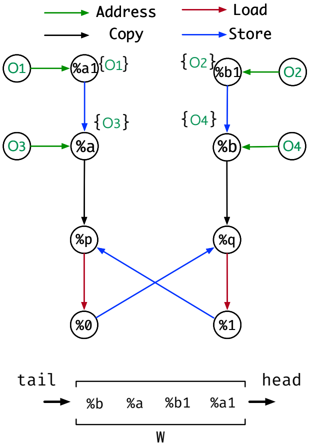



```
G = < V,E >   // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes
foreach address p = &o do
    pts(p) = {o}
     W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
              E ← E ∪ {q→o}
              W ← W ∪ {q}
      foreach load r = *p do
         if o→r ∉ E then
              E ← E ∪ {o→r}
              W ← W ∪ {o}
   foreach p→x ∈ E do
       pts(x) ← pts(x) ∪ pts(p)
       if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

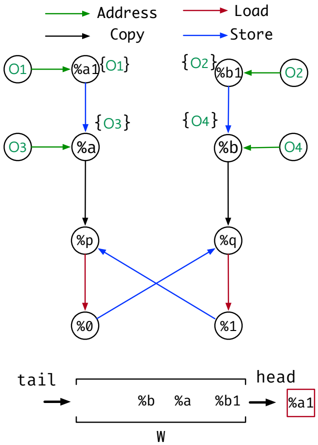



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
              E ← E ∪ {q→o}
              W ← W ∪ {q}
      foreach load r = *p do
         if o→r ∉ E then
              E ← E ∪ {o→r}
              W ← W ∪ {o}
   foreach p→x ∈ E do
       pts(x) ← pts(x) ∪ pts(p)
       if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

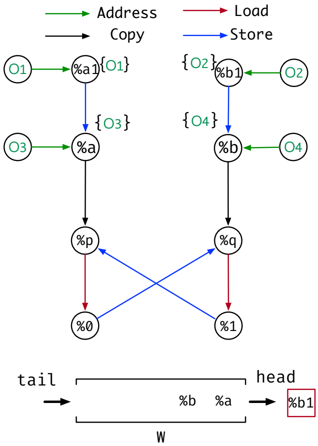



```
G = < V,E >   // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
     W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E then
                E ← E ∪ {q→o}
                W ← W ∪ {q}
        foreach load r = *p do
            if o→r ∉ E then
                E ← E ∪ {o→r}
                W ← W ∪ {o}
    foreach p→x ∈ E do
        pts(x) ← pts(x) ∪ pts(p)
        if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen’s Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

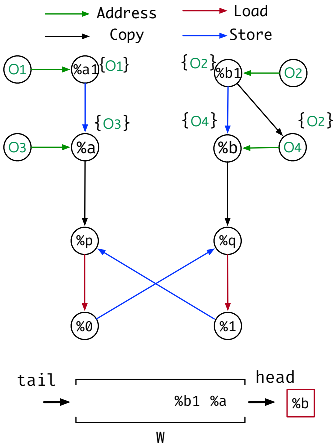



```
G = < V,E >   // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E then
                E ← E ∪ {q→o}
                W ← W ∪ {q}
        foreach load r = *p do
            if o→r ∉ E then
                E ← E ∪ {o→r}
                W ← W ∪ {o}
    foreach p→x ∈ E do
        pts(x) ← pts(x) ∪ pts(p)
        if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

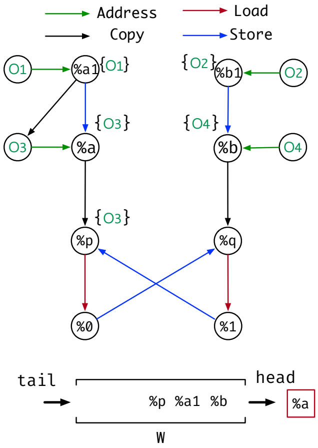



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E then
                E ← E ∪ {q→o}
                W ← W ∪ {q}
        foreach load r = *p do
            if o→r ∉ E then
                E ← E ∪ {o→r}
                W ← W ∪ {o}
    foreach p→x ∈ E do
        pts(x) ← pts(x) ∪ pts(p)
        if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

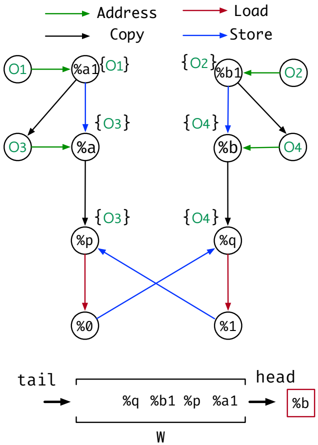



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E then
                E ← E ∪ {q→o}
                W ← W ∪ {q}
        foreach load r = *p do
            if o→r ∉ E then
                E ← E ∪ {o→r}
                W ← W ∪ {o}
    foreach p→x ∈ E do
        pts(x) ← pts(x) ∪ pts(p)
        if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

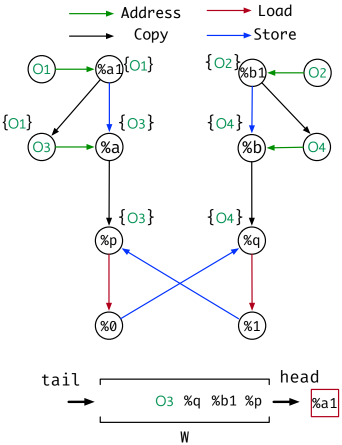



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
     W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
              E ← E ∪ {q→o}
              W ← W ∪ {q}
      foreach load r = *p do
         if o→r ∉ E then
              E ← E ∪ {o→r}
              W ← W ∪ {o}
   foreach p→x ∈ E do
       pts(x) ← pts(x) ∪ pts(p)
       if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

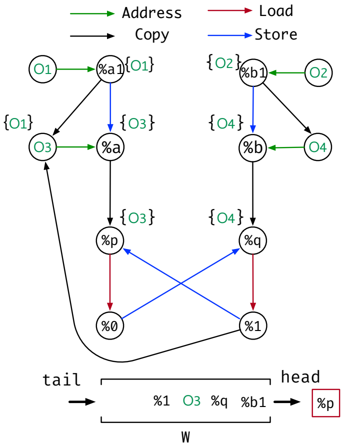



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
     W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
              E ← E ∪ {q→o}
              W ← W ∪ {q}
      foreach load r = *p do
         if o→r ∉ E then
              E ← E ∪ {o→r}
              W ← W ∪ {o}
   foreach p→x ∈ E do
       pts(x) ← pts(x) ∪ pts(p)
       if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

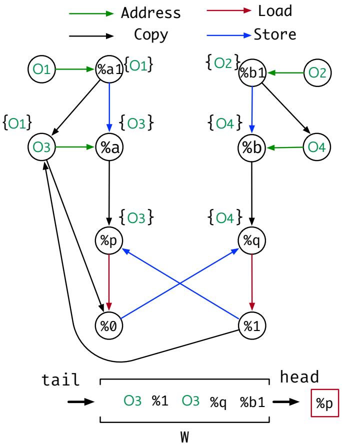



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E then
                E ← E ∪ {q→o}
                W ← W ∪ {q}
        foreach load r = *p do
            if o→r ∉ E then
                E ← E ∪ {o→r}
                W ← W ∪ {o}
    foreach p→x ∈ E do
        pts(x) ← pts(x) ∪ pts(p)
        if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

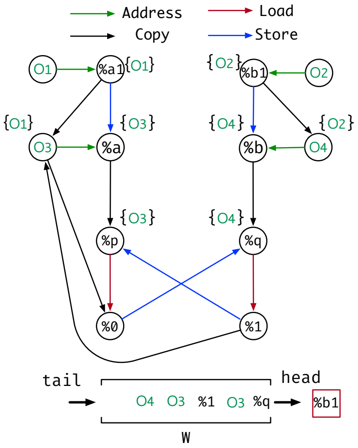



```
G = < V,E >   // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E then
                E ← E ∪ {q→o}
                W ← W ∪ {q}
        foreach load r = *p do
            if o→r ∉ E then
                E ← E ∪ {o→r}
                W ← W ∪ {o}
    foreach p→x ∈ E do
        pts(x) ← pts(x) ∪ pts(p)
        if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8**, %p, align 8
%1 = load i8**, %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

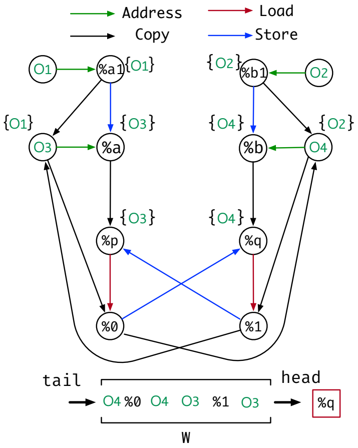



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
     W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
             E ← E ∪ {q→o}
             W ← W ∪ {q}
      foreach load r = *p do
         if o→r ∉ E then
             E ← E ∪ {o→r}
             W ← W ∪ {o}
   foreach p→x ∈ E do
      pts(x) ← pts(x) ∪ pts(p)
      if pts(x) changed then
         W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

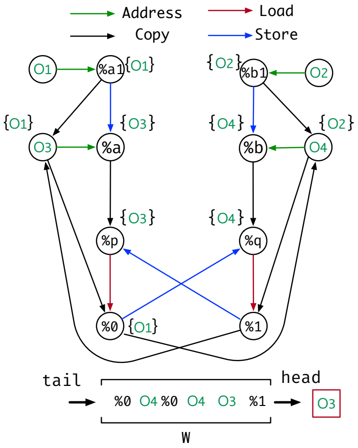



```
G = < V,E >   // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
              E ← E ∪ {q→o}
              W ← W ∪ {q}
      foreach load r = *p do
         if o→r ∉ E then
              E ← E ∪ {o→r}
              W ← W ∪ {o}
   foreach p→x ∈ E do
       pts(x) ← pts(x) ∪ pts(p)
       if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

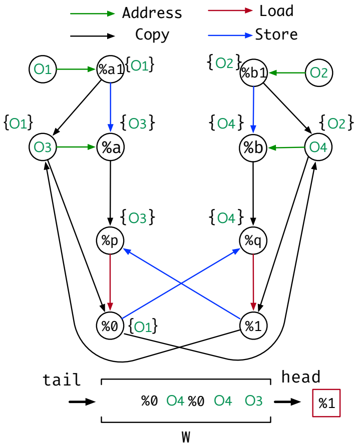



```
G = < V,E >   // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
     W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
              E ← E ∪ {q→o}
              W ← W ∪ {q}
      foreach load r = *p do
         if o→r ∉ E then
              E ← E ∪ {o→r}
              W ← W ∪ {o}
   foreach p→x ∈ E do
      pts(x) ← pts(x) ∪ pts(p)
      if pts(x) changed then
          W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

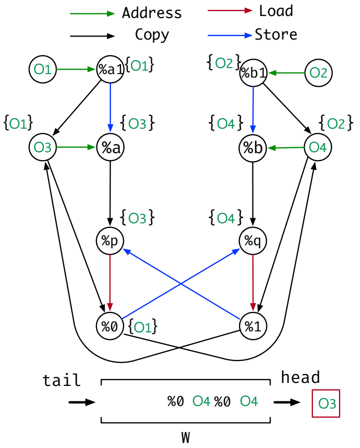



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
     W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E then
                E ← E ∪ {q→o}
                W ← W ∪ {q}
        foreach load r = *p do
            if o→r ∉ E then
                E ← E ∪ {o→r}
                W ← W ∪ {o}
    foreach p→x ∈ E do
        pts(x) ← pts(x) ∪ pts(p)
        if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1          // O1
%b1 = alloca i8, align 1          // O2
%a = alloca i8*, align 8          // O3
%b = alloca i8*, align 8          // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

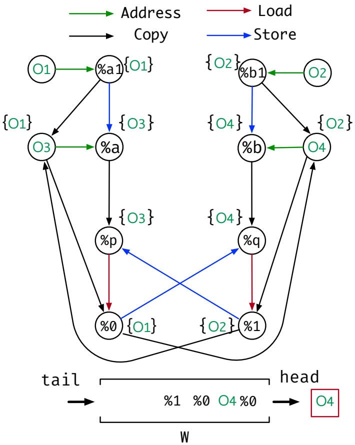



```
G = < V,E >   // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E then
                E ← E ∪ {q→o}
                W ← W ∪ {q}
        foreach load r = *p do
            if o→r ∉ E then
                E ← E ∪ {o→r}
                W ← W ∪ {o}
    foreach p→x ∈ E do
        pts(x) ← pts(x) ∪ pts(p)
        if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen’s Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

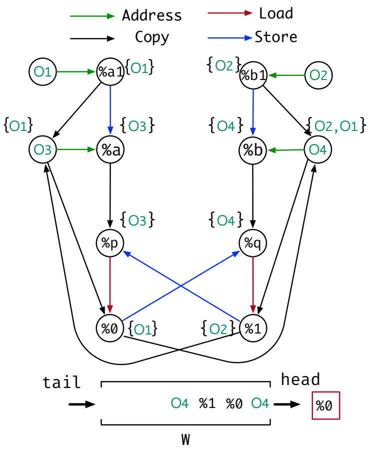



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
             E ← E ∪ {q→o}
             W ← W ∪ {q}
      foreach load r = *p do
         if o→r ∉ E then
             E ← E ∪ {o→r}
             W ← W ∪ {o}
   foreach p→x ∈ E do
      pts(x) ← pts(x) ∪ pts(p)
      if pts(x) changed then
         W ← W ∪ {x}
```

# Andersen’s Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1          // O1
%b1 = alloca i8, align 1          // O2
%a = alloca i8*, align 8          // O3
%b = alloca i8*, align 8          // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

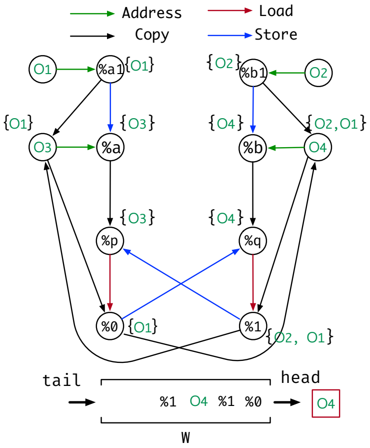



```
G = < V,E >   // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
     W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
              E ← E ∪ {q→o}
              W ← W ∪ {q}
      foreach load r = *p do
         if o→r ∉ E then
              E ← E ∪ {o→r}
              W ← W ∪ {o}
   foreach p→x ∈ E do
       pts(x) ← pts(x) ∪ pts(p)
       if pts(x) changed then
             W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

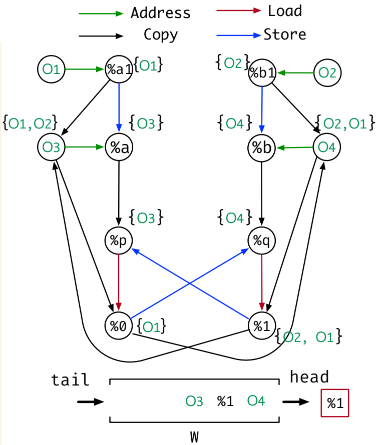



```
G = < V,E >   // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E then
                E ← E ∪ {q→o}
                W ← W ∪ {q}
        foreach load r = *p do
            if o→r ∉ E then
                E ← E ∪ {o→r}
                W ← W ∪ {o}
    foreach p→x ∈ E do
        pts(x) ← pts(x) ∪ pts(p)
        if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen's Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1          // O1
%b1 = alloca i8, align 1          // O2
%a = alloca i8*, align 8          // O3
%b = alloca i8*, align 8          // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

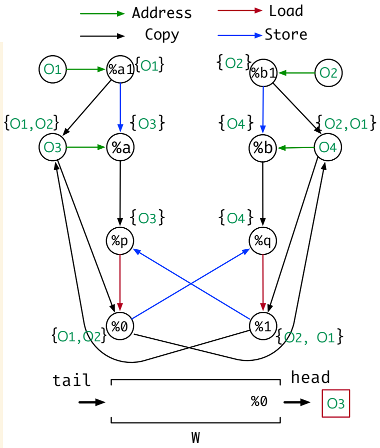



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
     W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
              E ← E ∪ {q→o}
              W ← W ∪ {q}
      foreach load r = *p do
         if o→r ∉ E then
              E ← E ∪ {o→r}
              W ← W ∪ {o}
   foreach p→x ∈ E do
       pts(x) ← pts(x) ∪ pts(p)
       if pts(x) changed then
            W ← W ∪ {x}
```

# Andersen’s Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

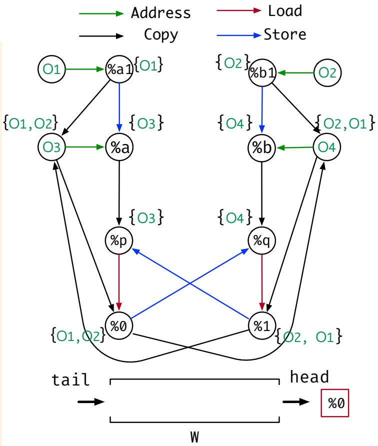



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
     W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
       foreach store *p = q do
          if q→o ∉ E then
              E ← E ∪ {q→o}
              W ← W ∪ {q}
       foreach load r = *p do
          if o→r ∉ E then
              E ← E ∪ {o→r}
              W ← W ∪ {o}
    foreach p→x ∈ E do
       pts(x) ← pts(x) ∪ pts(p)
       if pts(x) changed then
          W ← W ∪ {x}
```

# Andersen’s Pointer Analysis

## Algorithm

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```




```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E then
                E ← E ∪ {q→o}
                W ← W ∪ {q}
        foreach load r = *p do
            if o→r ∉ E then
                E ← E ∪ {o→r}
                W ← W ∪ {o}
    foreach p→x ∈ E do
        pts(x) ← pts(x) ∪ pts(p)
        if pts(x) changed then
            W ← W ∪ {x}
```