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} ⊆ Pts(p)
Copy:	p = q	Pts(q) ⊆ Pts(p)
Load:	p = *q	∀ o ∈ Pts(q), Pts(o) ⊆ Pts(p)
Store:	*p = q	∀ o ∈ Pts(p), Pts(q) ⊆ Pts(o)

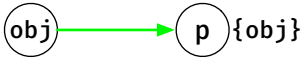
Andersen's Pointer Analysis

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
Address:	p = obj	%p = alloca // obj

Constraint Graph



pts(p) = {obj}

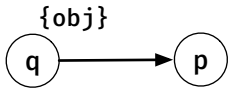
Andersen's Pointer Analysis

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

Constraint Graph



pts(p) = pts(q)

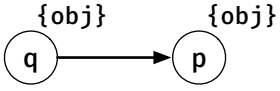
Andersen's Pointer Analysis

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)

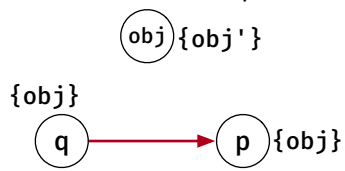
Andersen's Pointer Analysis

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



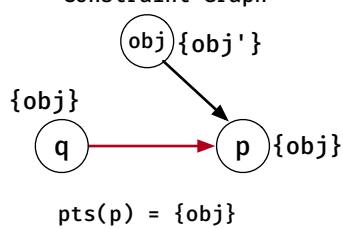
pts(p) = {obj}

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



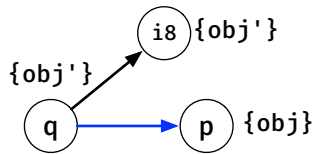
Andersen's Pointer Analysis

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

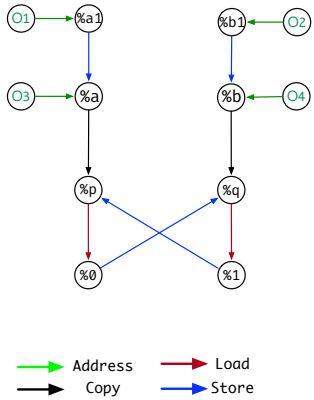
Constraint Graph



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
}
```

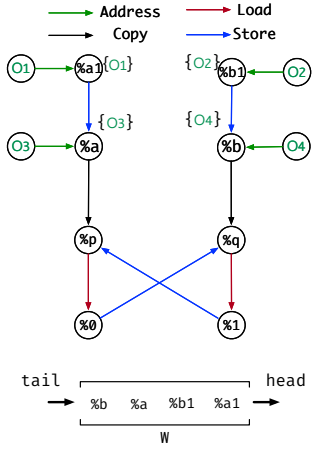



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
            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

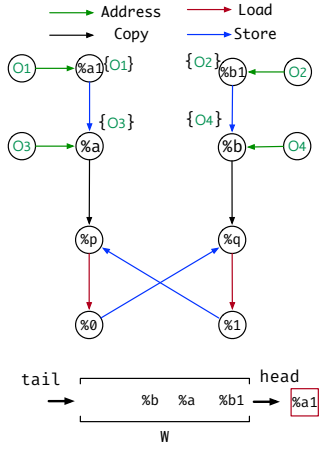



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
1 foreach address p = &o do
2     pts(p) = {o}
3     W ← W ∪ {p}
4 while W ≠ ∅ do
5     p ← select-from(W)
6     foreach o ∈ pts(p) do
7         foreach store *p = q do
8             if q→o ∉ E then
9                 E ← E ∪ {q→o}
10                W ← W ∪ {q}
11        foreach load r = *p do
12            if o→r ∉ E then
13                E ← E ∪ {o→r}
14                W ← W ∪ {o}
15    foreach p→x ∈ E do
16        pts(x) ← pts(x) ∪ pts(p)
17        if pts(x) changed then
18            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
}
```

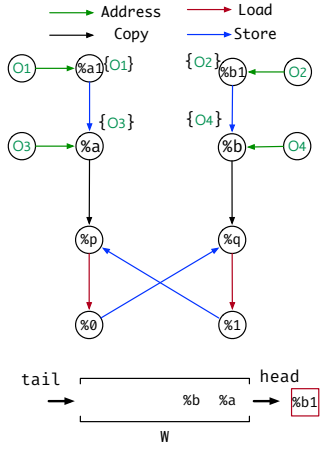



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
    p ← select-from(W)
    foreach o ∈ pts(p) do
        foreach store *p = q do
            if q→o ∉ E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

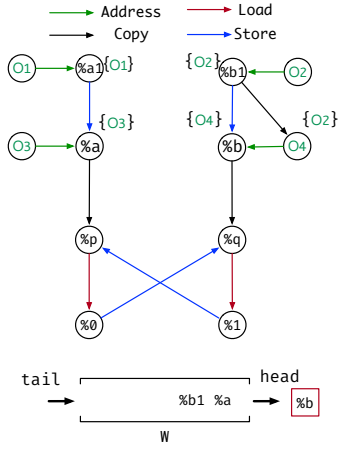



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
1  foreach address p = &o do
2      pts(p) = {o}
3      W ← W ∪ {p}
4  while W ≠ ∅ do
5      p ← select-from(W)
6      foreach o ∈ pts(p) do
7          foreach store *p = q do
8              if q→o ∉ E then
9                  E ← E ∪ {q→o}
10                 W ← W ∪ {q}
11         foreach load r = *p do
12             if o→r ∉ E then
13                 E ← E ∪ {o→r}
14                 W ← W ∪ {o}
15     foreach p→x ∈ E do
16         pts(x) ← pts(x) ∪ pts(p)
17         if pts(x) changed then
18             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
}
```

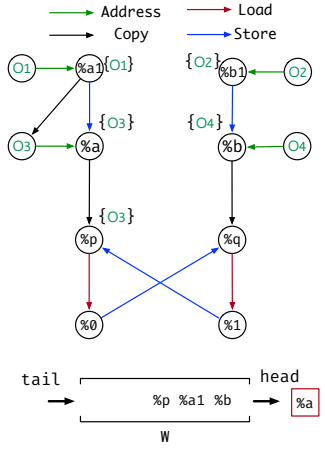



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
1  foreach address p = &o do
2      pts(p) = {o}
3      W ← W ∪ {p}
4  while W ≠ ∅ do
5      p ← select-from(W)
6      foreach o ∈ pts(p) do
7          foreach store *p = q do
8              if q→o ∉ E then
9                  E ← E ∪ {q→o}
10                 W ← W ∪ {q}
11         foreach load r = *p do
12             if o→r ∉ E then
13                 E ← E ∪ {o→r}
14                 W ← W ∪ {o}
15     foreach p→x ∈ E do
16         pts(x) ← pts(x) ∪ pts(p)
17         if pts(x) changed then
18             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
}
```

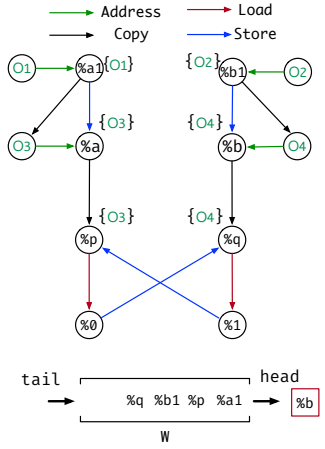



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
1  foreach address p = &o do
2      pts(p) = {o}
3      W ← W ∪ {p}
4  while W ≠ ∅ do
5      p ← select-from(W)
6      foreach o ∈ pts(p) do
7          foreach store *p = q do
8              if q→o ∉ E then
9                  E ← E ∪ {q→o}
10                 W ← W ∪ {q}
11         foreach load r = *p do
12             if o→r ∉ E then
13                 E ← E ∪ {o→r}
14                 W ← W ∪ {o}
15     foreach p→x ∈ E do
16         pts(x) ← pts(x) ∪ pts(p)
17         if pts(x) changed then
18             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
}
```

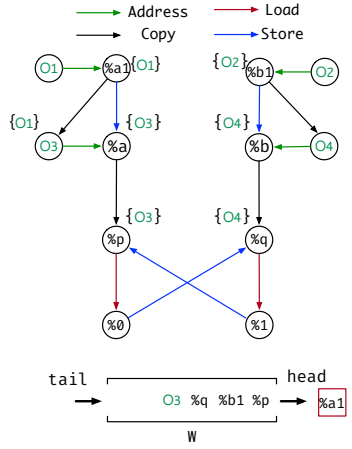



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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

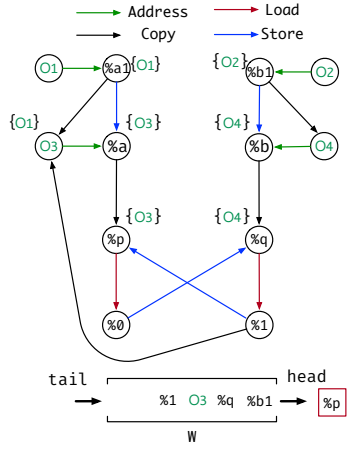



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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

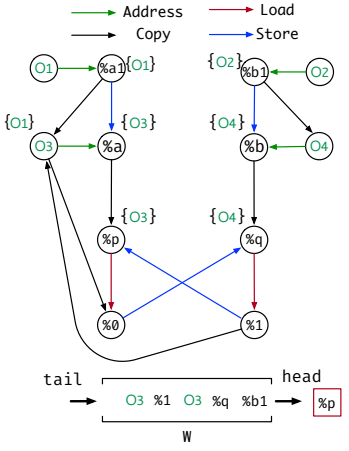



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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

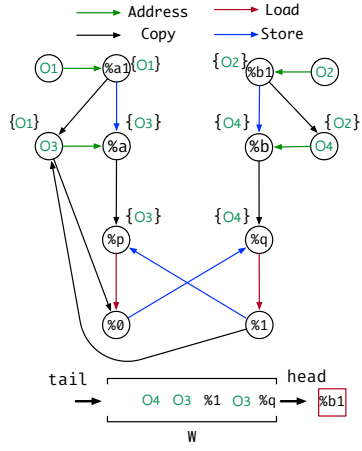



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
1  foreach address p = &o do
2      pts(p) = {o}
3      W ← W ∪ {p}
4  while W ≠ ∅ do
5      p ← select-from(W)
6      foreach o ∈ pts(p) do
7          foreach store *p = q do
8              if q→o ∉ E then
9                  E ← E ∪ {q→o}
10                 W ← W ∪ {q}
11         foreach load r = *p do
12             if o→r ∉ E then
13                 E ← E ∪ {o→r}
14                 W ← W ∪ {o}
15     foreach p→x ∈ E do
16         pts(x) ← pts(x) ∪ pts(p)
17         if pts(x) changed then
18             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
}
```

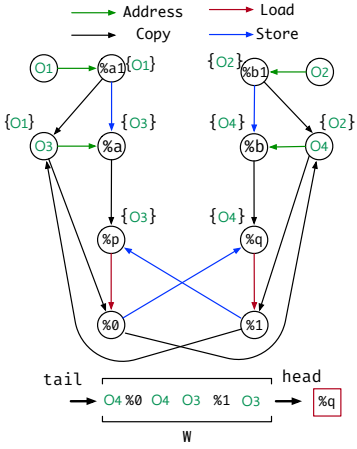



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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

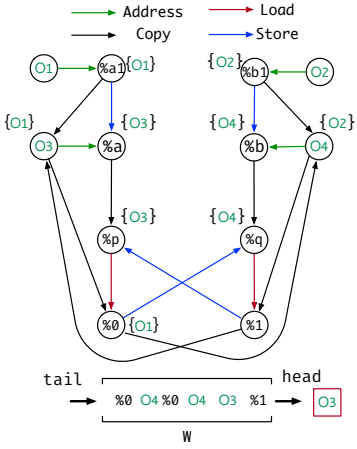



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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

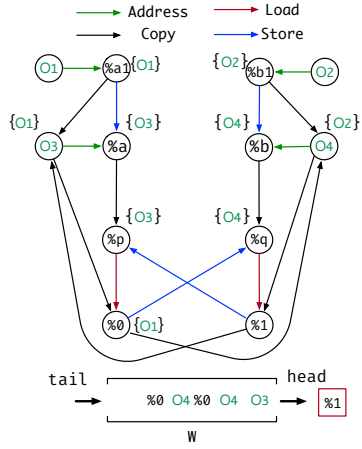



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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

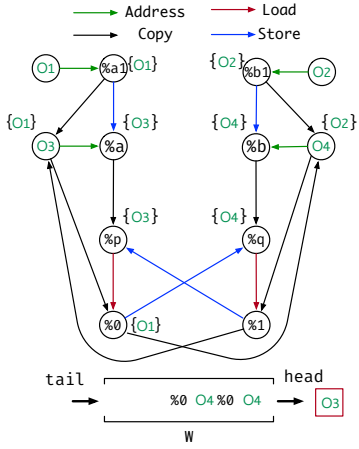



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
1 foreach address p = &o do
2     pts(p) = {o}
3     W ← W ∪ {p}
4 while W ≠ ∅ do
5     p ← select-from(W)
6     foreach o ∈ pts(p) do
7         foreach store *p = q do
8             if q→o ∉ E then
9                 E ← E ∪ {q→o}
10                W ← W ∪ {q}
11        foreach load r = *p do
12            if o→r ∉ E then
13                E ← E ∪ {o→r}
14                W ← W ∪ {o}
15    foreach p→x ∈ E do
16        pts(x) ← pts(x) ∪ pts(p)
17        if pts(x) changed then
18            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
}
```

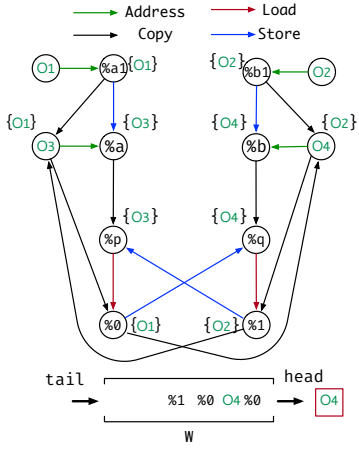



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
1 foreach address p = &o do
2     pts(p) = {o}
3     W ← W ∪ {p}
4 while W ≠ ∅ do
5     p ← select-from(W)
6     foreach o ∈ pts(p) do
7         foreach store *p = q do
8             if q→o ∉ E then
9                 E ← E ∪ {q→o}
10                W ← W ∪ {q}
11        foreach load r = *p do
12            if o→r ∉ E then
13                E ← E ∪ {o→r}
14                W ← W ∪ {o}
15    foreach p→x ∈ E do
16        pts(x) ← pts(x) ∪ pts(p)
17        if pts(x) changed then
18            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
}
```

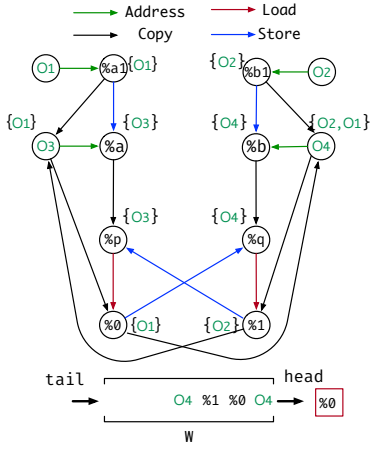



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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

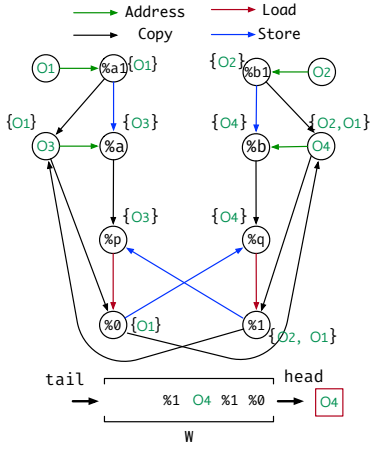



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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

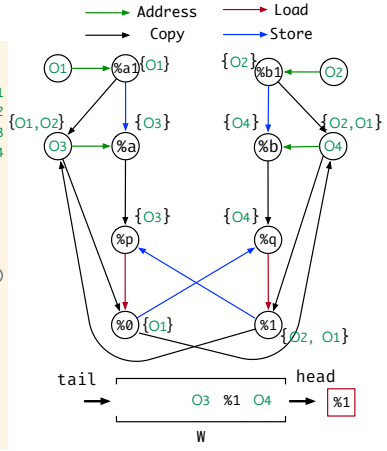



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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

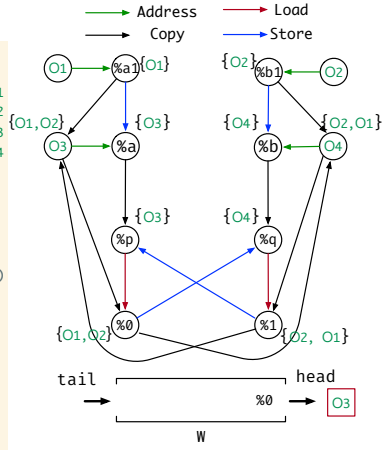



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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

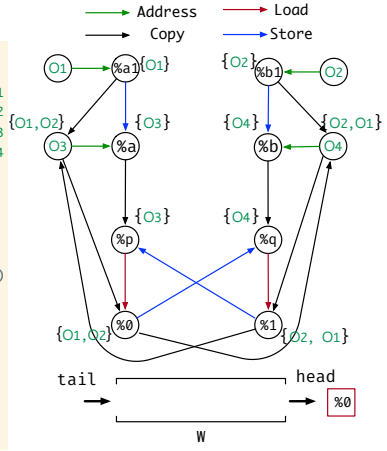



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
W: a vector of nodes (WorkList)
foreach address p = &o do
    pts(p) = {o}
    W ← W ∪ {p}
while W ≠ ∅ do
   p ← select-from(W)
   foreach o ∈ pts(p) do
      foreach store *p = q do
         if q→o ∉ E then
 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