Lab: Data-Dependence and Pointer Aliasing

(Week 3)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Quiz-1 + Lab-Exercise-1 + Assignment-1

- A set of quizzes on WebCMS (5 points)
  - LLVM compiler and its intermediate representation
  - Code graphs (including ICFG and PAG)
- Lab-Exercise-1 (5 points)
  - Implement a graph traversal on a general graph
- Assignment-1 (20 points)
  - **Control-flow**: Implement a context-sensitive graph traversal on a CodeGraph (i.e., ICFG) and print **feasible** paths from a source node to a sink node on the graph
  - **Data-flow**: Implement Andersen's inclusion-based constraint solving for points-to analysis
  - Implement a taint checker using control-flow analysis and data-flow analysis.

Quiz-1 + Lab-Exercise-1 + Assignment-1

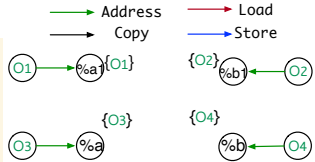
- A set of quizzes on WebCMS (5 points)
  - LLVM compiler and its intermediate representation
  - Code graphs (including ICFG and PAG)
- Lab-Exercise-1 (5 points)
  - Implement a graph traversal on a general graph
- Assignment-1 (20 points)
  - **Control-flow**: Implement a context-sensitive graph traversal on a CodeGraph (i.e., ICFG) and print **feasible** paths from a source node to a sink node on the graph
  - **Data-flow**: Implement Andersen's inclusion-based constraint solving for points-to analysis
  - Implement a taint checker using control-flow analysis and data-flow analysis.
  - **Specification and code template**: `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Assignment-1`
  - **SVF APIs for control- and data-flow analysis** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVF-CPP-API`

Andersen's Pointer Analysis

Algorithm

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

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

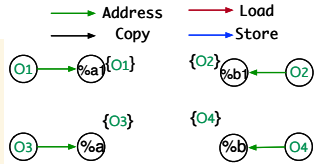


Andersen's Pointer Analysis

Algorithm

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

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

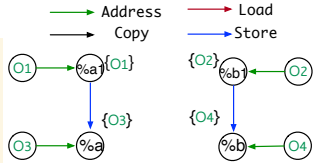


Andersen's Pointer Analysis

Algorithm

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

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

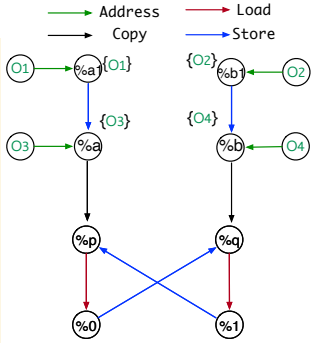


Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 1: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

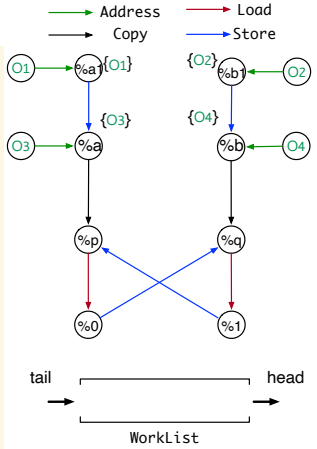
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 2: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

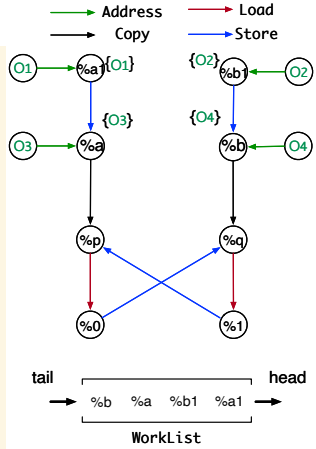
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 3: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

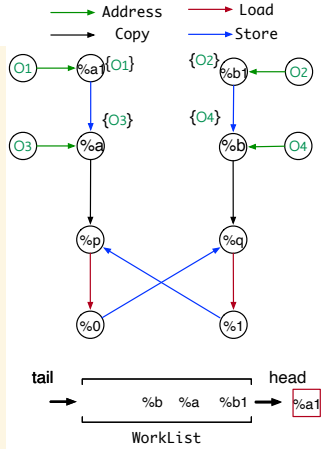
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 4: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

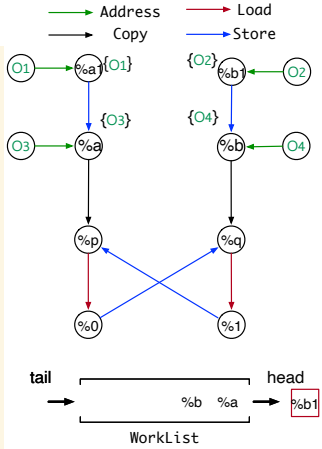
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 5: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

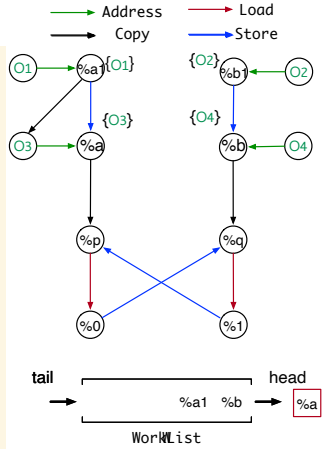
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 6: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

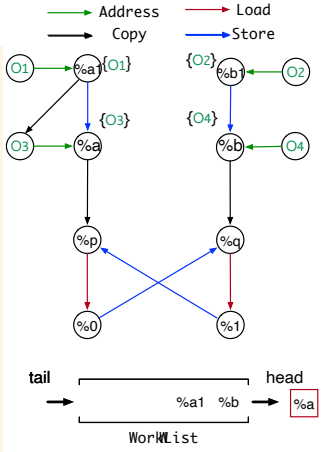
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 7: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

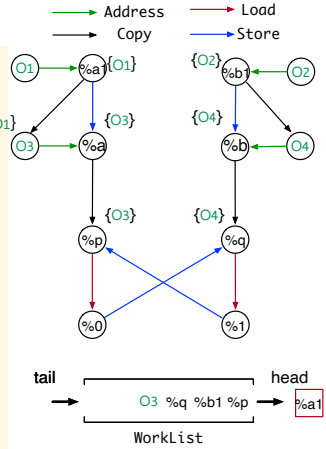
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 8: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

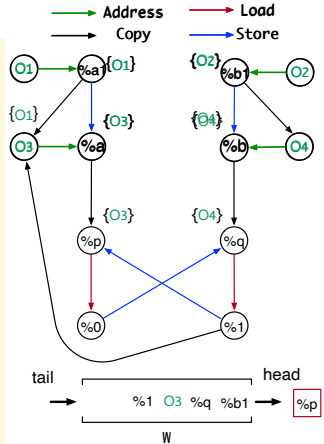
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 9: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

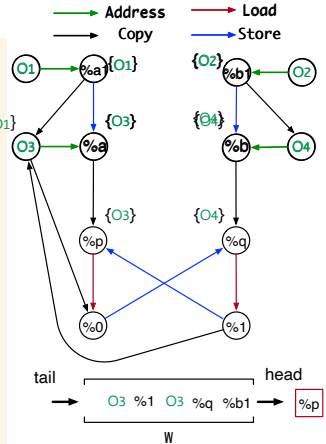
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 10: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

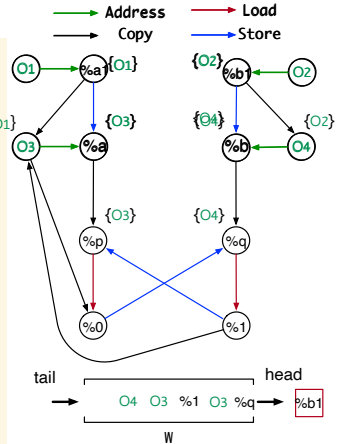
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                                  // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                        // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};                    // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                         // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};                    // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                             // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                              // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 11: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

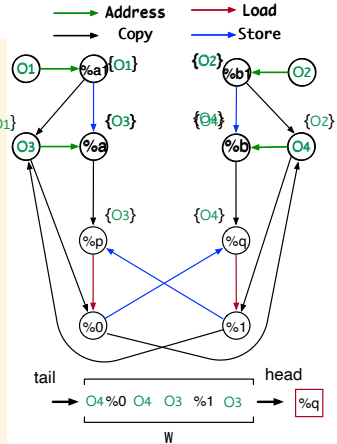
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 12: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

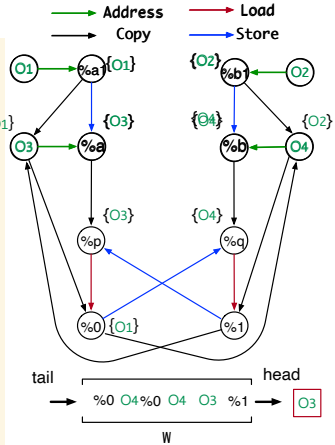
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                                  // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                        // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};                    // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                         // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};                    // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                             // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                              // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 13: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

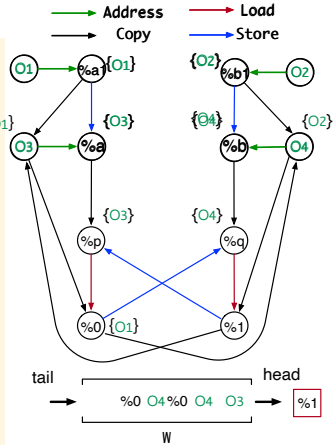
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                                  // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                        // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};                    // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                         // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};                    // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                             // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                              // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 14: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

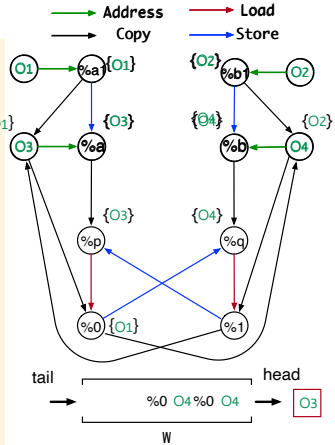
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                                  // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                        // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};                    // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                         // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};                    // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                             // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                              // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 15: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

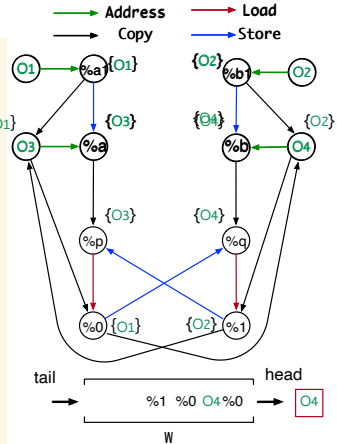
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                                  // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                        // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};                    // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                         // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};                    // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                             // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                              // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 16: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

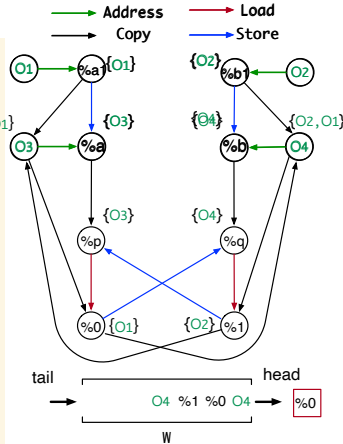
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 17: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

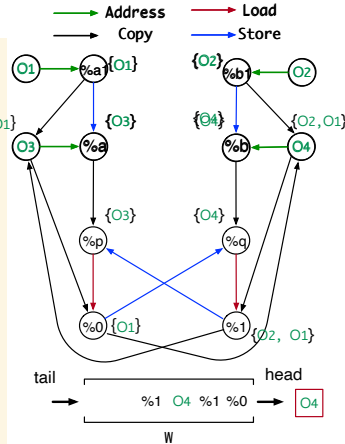
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 18: 1 Anderson's Pointer Analysis

Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

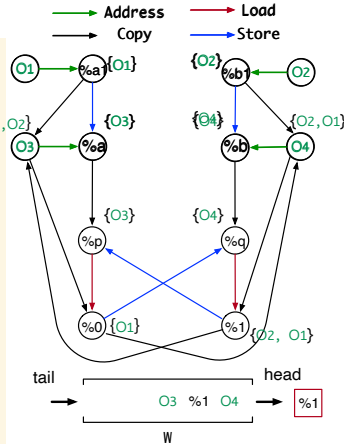
```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Andersen's Pointer Analysis

Algorithm

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

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



Algorithm 19: 1 Anderson's Pointer Analysis

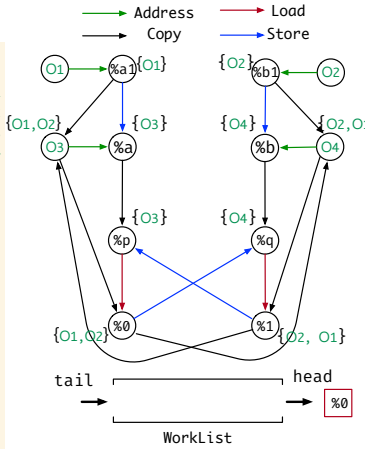
Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(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
}
```



Algorithm 20: 1 Anderson's Pointer Analysis

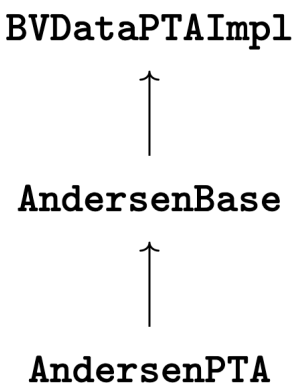
Input : G =< V, E >: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
1  WorkList := an empty vector of nodes;
2  foreach o --Address--> p do                          // Address rule
3      pts(p) = o;
4      pushIntoWorklist(p);
5  while WorkList ≠ do
6      p := popFromWorklist();
7      foreach o ∈ pts(p) do
8          foreach q --Store--> p ∈ E do                // Store rule
9              if q --Copy--> o ∉ E then
10                 E := E ∪ {q --Copy--> o};            // Add copy edge
11                 pushIntoWorklist(q);
12         foreach p --Load--> r ∈ E do                 // Load rule
13             if o --Copy--> r ∉ E then
14                 E := E ∪ {o --Copy--> r};            // Add copy edge
15                 pushIntoWorklist(o);
16     foreach p --Copy--> x ∈ E do                     // Copy rule
17         pts(x) := pts(x) ∪ pts(p);
18         if pts(x) changed then
19             pushIntoWorklist(x);
20     foreach p --Gep--> x ∈ E do                      // Gep rule
21         foreach o ∈ pts(p) do
22             pts(x) := pts(x) ∪ {o.fld};
23         if pts(x) changed then
24             pushIntoWorklist(x);
```

Assignment Structure

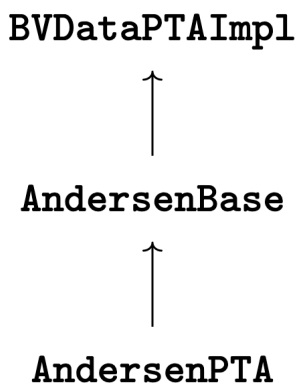
- You will be working on `AndersenPTA`'s `solveWorklist` method.

Assignment Structure



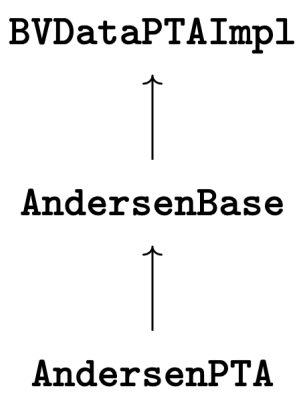
- You will be working on `AndersenPTA`'s `solveWorklist` method.
- Constraint graph is the field `consCG`.

Assignment Structure



- You will be working on `AndersenPTA`'s `solveWorklist` method.
- Constraint graph is the field `consCG`.
- Address edge processing is done for you.

Assignment Structure



- You will be working on `AndersenPTA`'s `solveWorklist` method.
- Constraint graph is the field `consCG`.
- Address edge processing is done for you.
- Note in the API there is a `getDirectInEdges`/`getDirectOutEdges` but no `getCopyIn/OutEdges`. This is intentional, use the `Direct` variant.
- You will reuse this assignment for assignment 4, make sure it is clean. :)

Lab: Information Flow Tracking

(Week 3)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Assignment 1: Taint Tracker

- Implement method `readSrcSnkFromFile` in `Assignment-4.cpp` using C++ file reading to configure sources and sinks.
- Implement method `printICFGPath` to collect the tainted ICFG paths and add each path (a sequence of node IDs) as a string into `std::set<std::string> paths` similar to Assignment 2
- Implement method `aliasCheck` to check aliases of the variables at source and sink.

Coding Task

- Code template and specification: `https://github.com/SVF-tools/Teaching-Software-Analysis/wiki/Assignment-4`
- Make sure your previous implementations in `Assignment-2.cpp` and `Assignment-3.cpp` are in place.
  - Class `TaintGraphTraversal` in Assignment 4 is a **child class** of 'ICFGTraversal'. `TaintGraphTraversal` will use the `DFS` method implemented in Assignment 2 for **control-flow traversal**.
  - Andersen's analysis implemented in Assignment 3 will also be used for **checking aliases** between two pointers.

C++ File Reading

Implement method `readSrcSnkFormFile` in `Assignment-4.cpp` to parse the two lines from `SrcSnk.txt` in the form of

```
source -> { source src set getname update getchar tgetstr }
sink -> { sink mysql_query system require chmod broadcast }
```

Please refer to the following links (among many others) for C++ file reading:

- `https://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm`
- `https://www.cplusplus.com/doc/tutorial/files/`
- `https://linuxhint.com/cplusplus_read_write/`
- `https://opensource.com/article/21/3/ccc-input-output`