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

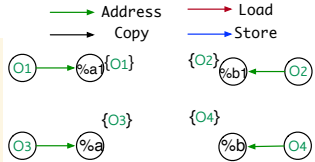
- A set of quizzes on WebCMS (5 points)
  - LLVM compiler and its intermediate representation
  - Code graphs (including ICFG and PAG)
- Lab-Exercise-1 (5 points)
  - Implement a graph traversal on a general graph
- Assignment-1 (20 points)
  - **Control-flow**: Implement a context-sensitive graph traversal 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
}
```

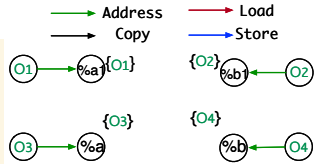


Andersen's Pointer Analysis

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
}
```

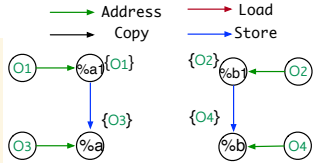


Andersen's Pointer Analysis

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
}
```

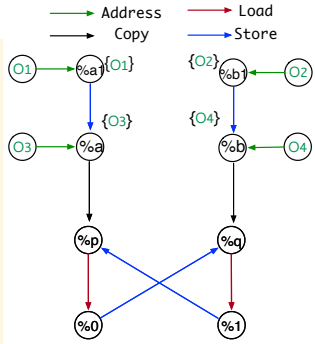


Andersen's Pointer Analysis

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
}
```



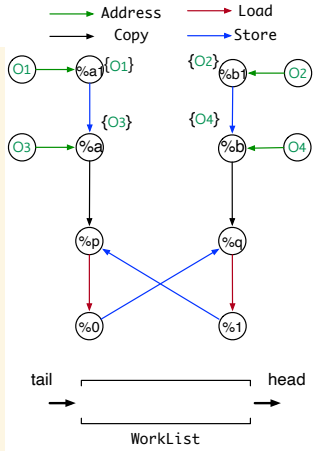

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



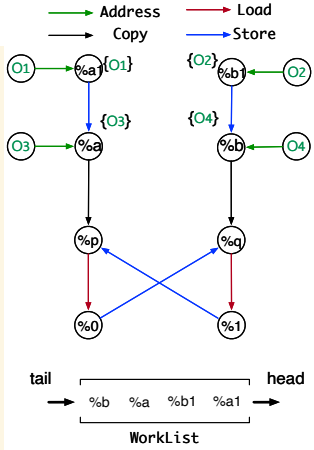

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



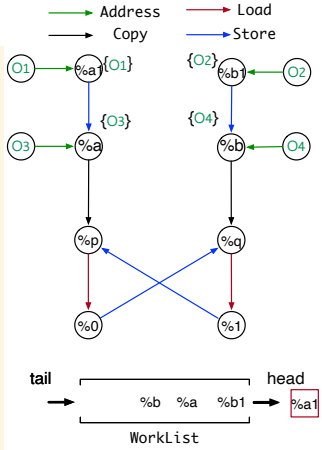

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5      p <- popFromWorklist()
6      foreach o ∈ pts(p) do
7          foreach q --Store--> p do              // Store rule
8              if q --Copy--> o ∉ E then
9                  E <- E ∪ { q --Copy--> o }     // Add copy edge
10                 pushIntoWorklist(q)
11         foreach p --Load--> r do               // Load rule
12             if o --Copy--> r ∉ E then
13                 E <- E ∪ { o --Copy--> r }     // Add copy edge
14                 pushIntoWorklist(o)
15     foreach p --Copy--> x ∈ E do               // Copy rule
16         pts(x) <- pts(x) ∪ pts(p)
17         if pts(x) changed then
18             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
}
```



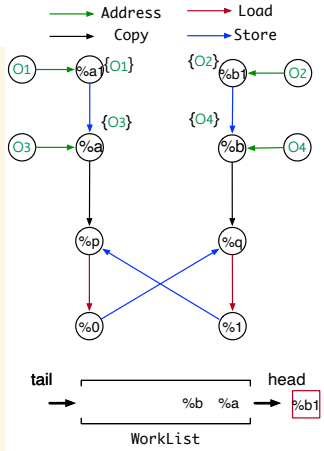

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5      p <- popFromWorklist()
6      foreach o ∈ pts(p) do
7          foreach q --Store--> p do              // Store rule
8              if q --Copy--> o ∉ E then
9                  E <- E ∪ { q --Copy--> o }     // Add copy edge
10                 pushIntoWorklist(q)
11         foreach p --Load--> r do               // Load rule
12             if o --Copy--> r ∉ E then
13                 E <- E ∪ { o --Copy--> r }     // Add copy edge
14                 pushIntoWorklist(o)
15     foreach p --Copy--> x ∈ E do               // Copy rule
16         pts(x) <- pts(x) ∪ pts(p)
17         if pts(x) changed then
18             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
}
```



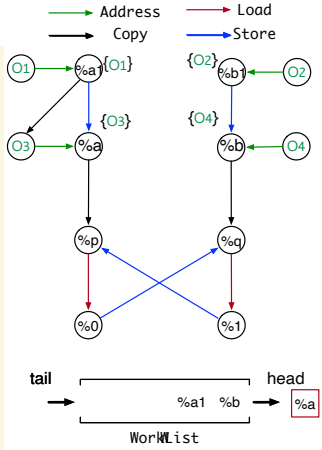

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5      p <- popFromWorklist()
6      foreach o ∈ pts(p) do
7          foreach q --Store--> p do              // Store rule
8              if q --Copy--> o ∉ E then
9                  E <- E ∪ { q --Copy--> o }     // Add copy edge
10                 pushIntoWorklist(q)
11         foreach p --Load--> r do               // Load rule
12             if o --Copy--> r ∉ E then
13                 E <- E ∪ { o --Copy--> r }     // Add copy edge
14                 pushIntoWorklist(o)
15     foreach p --Copy--> x ∈ E do               // Copy rule
16         pts(x) <- pts(x) ∪ pts(p)
17         if pts(x) changed then
18             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
}
```



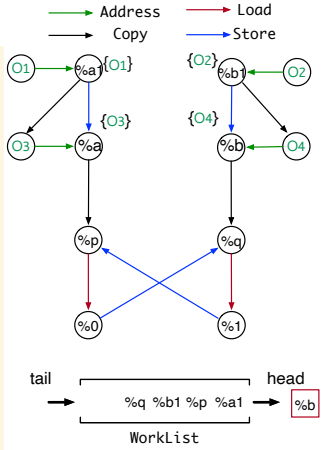

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



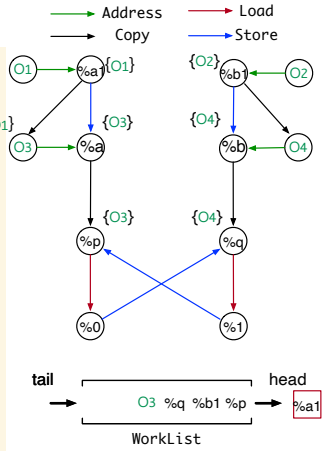

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5      p <- popFromWorklist()
6      foreach o ∈ pts(p) do
7          foreach q --Store--> p do              // Store rule
8              if q --Copy--> o ∉ E then
9                  E <- E ∪ { q --Copy--> o }     // Add copy edge
10                 pushIntoWorklist(q)
11         foreach p --Load--> r do               // Load rule
12             if o --Copy--> r ∉ E then
13                 E <- E ∪ { o --Copy--> r }     // Add copy edge
14                 pushIntoWorklist(o)
15     foreach p --Copy--> x ∈ E do               // Copy rule
16         pts(x) <- pts(x) ∪ pts(p)
17         if pts(x) changed then
18             pushIntoWorklist(x)
```

Andersen's Pointer Analysis

Algorithm

```
define i32 @main() #0 {
entry:
  %a1 = alloca i8, align 1          // O1
  %a = alloca ptr, align 8          // O3{O1}
  %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
}
```



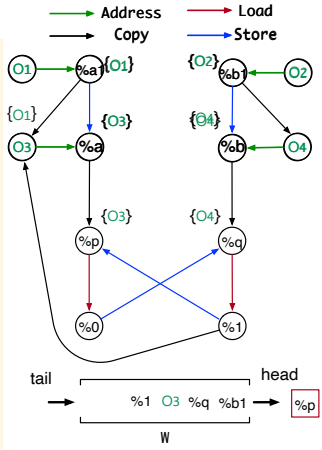

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



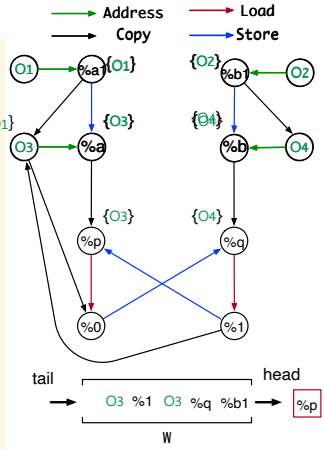

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p  do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ø do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p  do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }              // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r  do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }              // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x  ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



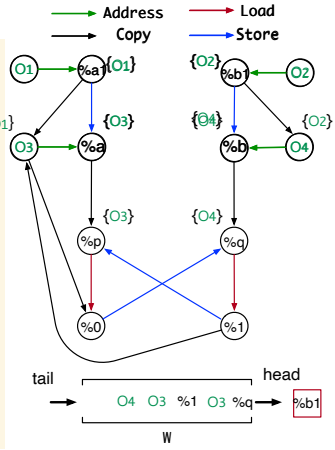

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ø do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



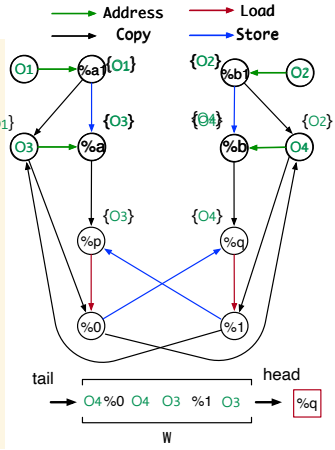

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ø do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



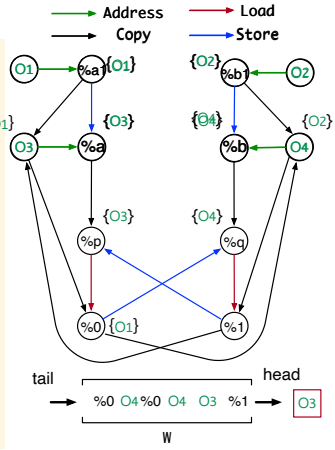

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



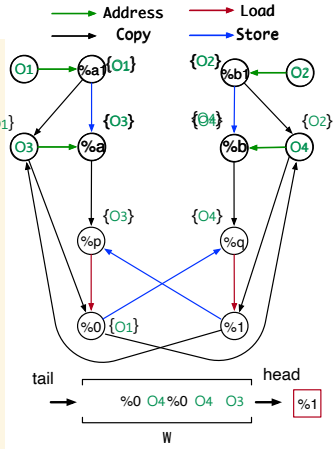

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



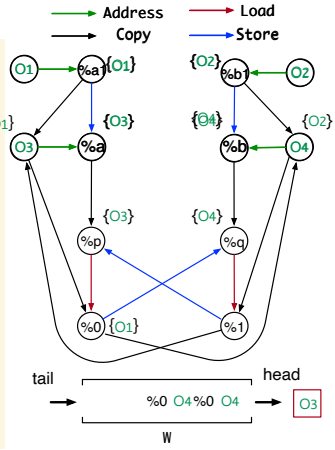

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



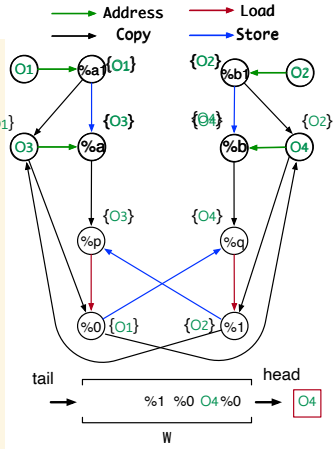

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2        pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



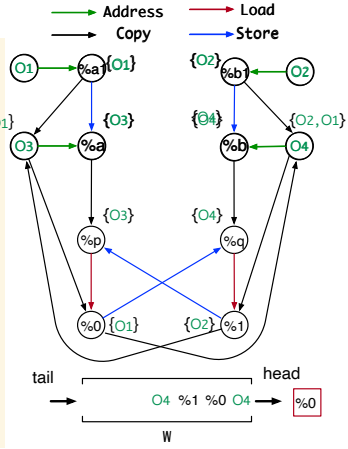

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p  do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p  do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }             // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r  do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }             // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x  ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



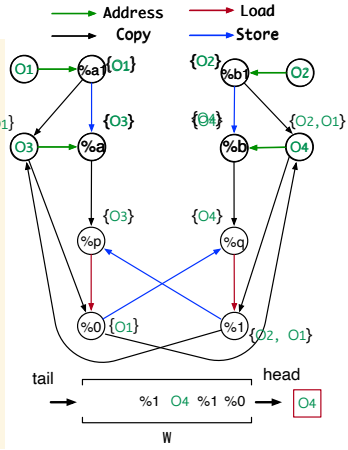

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2        pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```



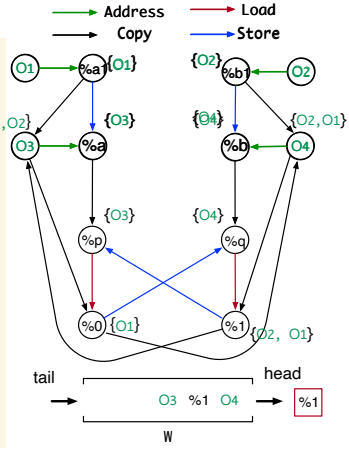

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```

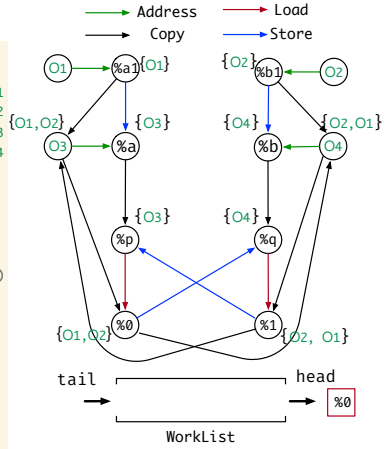



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do                    // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p <- popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do                  // Store rule
8        if q --Copy--> o ∉ E then
9          E <- E ∪ { q --Copy--> o }            // Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do                   // Load rule
12       if o --Copy--> r ∉ E then
13         E <- E ∪ { o --Copy--> r }            // Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do                 // Copy rule
16     pts(x) <- pts(x) ∪ pts(p)
17     if pts(x) changed then
18       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
}
```

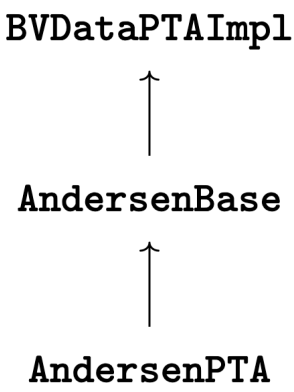



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
1  foreach o --Address--> p do          // Address rule
2      pts(p) = {o}
3      pushIntoWorklist(p)
4  while WorkList ≠ ∅ do
5    p ← popFromWorklist()
6    foreach o ∈ pts(p) do
7      foreach q --Store--> p do        // Store rule
8        if q --Copy--> o ∉ E then
9          E ← E ∪ {q --Copy--> o}// Add copy edge
10         pushIntoWorklist(q)
11     foreach p --Load--> r do         // Load rule
12       if o --Copy--> r ∉ E then
13         E ← E ∪ {o --Copy--> r}// Add copy edge
14         pushIntoWorklist(o)
15   foreach p --Copy--> x ∈ E do     // Copy rule
16     pts(x) ← pts(x) ∪ pts(p)
17     if pts(x) changed then
18         pushIntoWorklist(x)
```

Assignment Structure

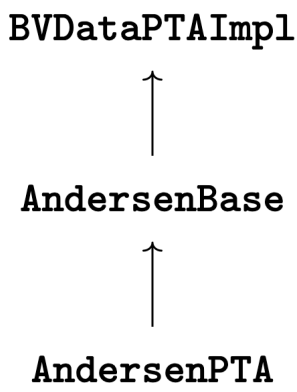
- You will be working on `AndersenPTA`'s `solveWorklist` method.

Assignment Structure



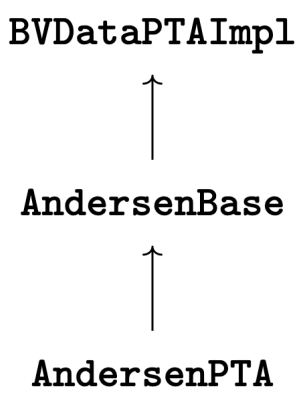
- You will be working on `AndersenPTA`'s `solveWorklist` method.
- Constraint graph is the field `consCG`.

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. :)