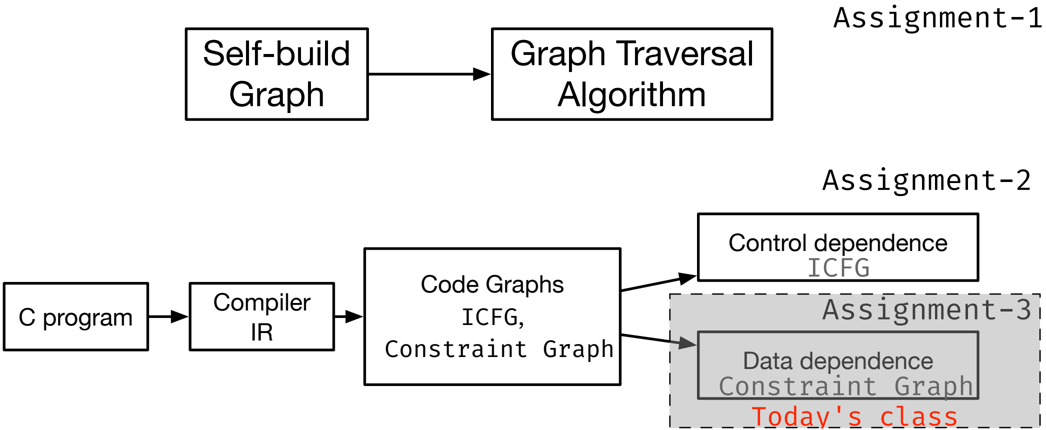
Data-Dependence

Yulei Sui

University of Technology Sydney, Australia

Today’s class



Data-Dependence

Definition-use relations between variables. Two types of variables on LLVM IR:

- **Top-level variables**, whose addresses are not taken (`ValPN` in SVF)
  - Including stack virtual **registers** (symbols starting with "%") and **global** variables (symbols starting with "@") are explicit, i.e., directly accessed.

Data-Dependence

Definition-use relations between variables. Two types of variables on LLVM IR:

- **Top-level variables**, whose addresses are not taken (`ValPN` in SVF)
  - Including stack virtual **registers** (symbols starting with "%") and **global** variables (symbols starting with "@") are explicit, i.e., directly accessed.
  - **Def-use for top-level variables are directly available on LLVM's SSA form.**
  - For example, def-use for %a1 from Instruction-1 to Instruction-2.
    - Instruction-1: `%a1 = alloca i8, align 1;`
    - Instruction-2: `store i8* %a1, i8** %a, align 8`

Data-Dependence

Definition-use relations between variables. Two types of variables on LLVM IR:

- **Top-level variables**, whose addresses are not taken (`ValPN` in SVF)
  - Including stack virtual **registers** (symbols starting with “%”) and **global** variables (symbols starting with “@”) are explicit, i.e., directly accessed.
  - **Def-use for top-level variables are directly available on LLVM’s SSA form.**
  - For example, def-use for %a1 from Instruction-1 to Instruction-2.
    - Instruction-1: `%a1 = alloca i8, align 1;`
    - Instruction-2: `store i8* %a1, i8** %a, align 8`
- **Address-taken variables** (abstract objects), accessed indirectly at load or store instructions via top-level variables (`ObjPN` in SVF)
  - A **stack object** created at an LLVM’s ‘alloca‘ instruction or a **heap object** created via (e.g., ‘malloc‘ callsite) or a **global object**.

Data-Dependence

Definition-use relations between variables. Two types of variables on LLVM IR:

- **Top-level variables**, whose addresses are not taken (`ValPN` in SVF)
  - Including stack virtual **registers** (symbols starting with "%") and **global** variables (symbols starting with "@") are explicit, i.e., directly accessed.
  - **Def-use for top-level variables are directly available on LLVM's SSA form.**
  - For example, def-use for %a1 from Instruction-1 to Instruction-2.
    - Instruction-1: `%a1 = alloca i8, align 1;`
    - Instruction-2: `store i8* %a1, i8** %a, align 8`
- **Address-taken variables** (abstract objects), accessed indirectly at load or store instructions via top-level variables (`ObjPN` in SVF)
  - A **stack object** created at an LLVM's 'alloca' instruction or a **heap object** created via (e.g., 'malloc' callsite) or a **global object**.
  - **Def-use for address-taken variables are computed via pointer analysis.**
  - For example, there is a def-use for object `o` from Instruction-1 to Instruction-2 if pointers `%a` and `%b` both point to `o`.
    - Instruction-1: `store i8* %a1, i8** %a, align 8`
    - Instruction-2: `%c = load i8** %b, align 8`

Pointer analysis

- Points-to Analysis: aims to statically determine the possible runtime values of a pointer at compile-time.
  - Compute the *points-to set* (**a set of address-taken variables**) of each *pointer* (**top-level variable)**
  - For example, `p = &a; q = p;`
  - The resulting points-to sets of `p` and `q` are: `pts(p) = pts(q) = {a}`

Pointer analysis

- Points-to Analysis: aims to statically determine the possible runtime values of a pointer at compile-time.
  - Compute the *points-to set* (**a set of address-taken variables**) of each *pointer* (**top-level variable)**
  - For example, `p = &a; q = p;`
  - The resulting points-to sets of `p` and `q` are: `pts(p) = pts(q) =` $\{$`a`$\}$
- Alias Analysis: determine whether two pointer dereferences refer to the same memory location.
  - If the points-to sets of two pointers `p` and `q` have overlapping elements (i.e., `pts(p)` $\cap$ `pts(q) is not empty`) then `p` and `q` are aliases. The derereferences of `p` and `q` may refer to the same memory location.

Pointer analysis

Why shall we learn pointer analysis?

- Essential for building data-dependence relations between variables (memory objects).

Pointer analysis

Why shall we learn pointer analysis?

- Essential for building data-dependence relations between variables (memory objects).
- Understanding aliases through different memory accesses

Pointer analysis

Why shall we learn pointer analysis?

- Essential for building data-dependence relations between variables (memory objects).
- Understanding aliases through different memory accesses
  - `p = &a; q = p; *p = x; y = *q;`
    `y` has the same value as `x` since `*p` and `*q` both refer to `a`.

Pointer analysis

Why shall we learn pointer analysis?

- Essential for building data-dependence relations between variables (memory objects).
- Understanding aliases through different memory accesses
  - `p = &a; q = p; *p = x; y = *q;`
    `y` has the same value as `x` since `*p` and `*q` both refer to `a`.
- Compiler optimizations and bug detection
  - Constant propagation
    - `*p = 1; x = *q;`
      `x` is a constant value and equals 1, if `p` and `q` are must-aliases (always point to the same memory location w.r.t every execution path).
    - `*p = 1; *q = r; x = p;`
      `x` is a constant value and equals 1, if `p` and `q` do not alias with each other.

Pointer analysis

Why shall we learn pointer analysis?

- Essential for building data-dependence relations between variables (memory objects).
- Understanding aliases through different memory accesses
  - `p = &a; q = p; *p = x; y = *q;`
    `y` has the same value as `x` since `*p` and `*q` both refer to `a`.
- Compiler optimizations and bug detection
  - Constant propagation
    - `*p = 1; x = *q;`
      `x` is a constant value and equals 1, if `p` and `q` are must-aliases (always point to the same memory location w.r.t every execution path).
    - `*p = 1; *q = r; x = p;`
      `x` is a constant value and equals 1, if `p` and `q` do not alias with each other.
  - Taint analysis
    - `*p = taintedInput; x = *q;`
      `x` is tainted if `p` and `q` are aliases.

Precision Dimensions

Can be generally classified into the following precision dimensions at different levels of abstractions.

Flow-insensitive analysis:

- Ignores program execution order
- A single solution across whole program

Context-insensitive analysis:

- Merges all of all calling contexts when analysing a program method

Path-insensitive analysis:

- Merges all incoming path information at the joint point of the control-flow graph

Flow-sensitive analysis:

- Respects the program execution order
- Separate solution at each program point

Context-sensitive analysis:

- Distinguishes between different calling contexts of a program method

Path-sensitive analysis:

- Computes a solution per (abstract) program path.

Precision Dimensions

Levels of Abstractions

Assume x is a tainted value

```
p = x

p = y
```

flow-sensitivity

at which
program point
p is tainted?

```
foo(x)      foo(y)
  call \    / call
      foo(p){

      }
```

context-sensitivity

under which
calling context
p is tainted?

```
if(cond)
     p = x
else
     p = y
```

path-sensitivity

along which
program path
p is tainted?

Andersen's Pointer analysis

Flow-, context-, and path-insensitive analysis

In this subject, we will practice **Andersen's analysis**[1], a **flow-insensitive, context-insensitive and path-insensitive Andersen's analysis** through analyzing the **Constraint Graph** of a program.

- One of the most popular and widely used pointer analyses
- Constraint solving, i.e., inclusion-based constraint solving between program variables (`PAGNode` in SVF)

[1] Andersen, L. O. (1994). Program analysis and specialization for the C programming language (Doctoral dissertation, University of Cophenhagen).

Andersen's Pointer Analysis

An inclusion-based analysis operating on top of the constraint graph of a program. SVF transforms each LLVM instruction into a constraint edge connecting two nodes

- A `ConstraintNode` represents
  - A pointer: (top-level variable) or
  - An object: (address-taken variable, i.e., heap, stack, global or function object)
- A `ConstraintEdge` represents a constraint between two nodes

Andersen's Pointer Analysis

An inclusion-based analysis operating on top of the constraint graph of a program. SVF transforms each LLVM instruction into a constraint edge connecting two nodes

- A `ConstraintNode` represents
  - A pointer: (top-level variable) or
  - An object: (address-taken variable, i.e., heap, stack, global or function object)
- A `ConstraintEdge` represents a constraint between two nodes

Constraint Type	C code	LLVM IR	Constraint rules
Address:	`p = &o`	`%p = alloca //o`	pts(p) = pts(p) ∪ {o}
Copy:	`q = p`	`%q = bitcast %p`	pts(q) = pts(q) ∪ pts(p)
Load:	`q =*p`	`%q = load %p`	∀ o ∈ pts(p): **pts(q) = pts(o) ∪ pts(q)**
Store:	`*p = q`	`store %q, %p`	∀ o ∈ pts(p): **pts(o) = pts(q) ∪ pts(o)**

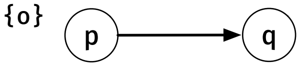
Andersen's Pointer Analysis

Constraint Type	C code	LLVM IR	Constraint rules
Address:	p = &o	%p = alloca //o	pts(p) = pts(p) ∪ {o}
Copy:	q = p	%q = bitcast %p	pts(q) = pts(q) ∪ pts(p)
Load:	q = *p	%q = load %p	∀ o ∈ pts(p): pts(q) = pts(o) ∪ pts(q)
Store:	*p = q	store %q, %p	∀ o ∈ pts(p): pts(o) = pts(q) ∪ pts(o)

pts(p) = {o}

Andersen's Pointer Analysis

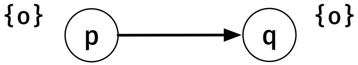
Constraint Type	C code	LLVM IR	Constraint rules
Address:	p = &o	%p = alloca //o	pts(p) = pts(p) ∪ {o}
Copy:	q = p	%q = bitcast %p	pts(q) = pts(q) ∪ pts(p)
Load:	q = *p	%q = load %p	∀ o ∊ pts(p): pts(q) = pts(o) ∪ pts(q)
Store:	*p = q	store %q, %p	∀ o ∊ pts(p): pts(o) = pts(q) ∪ pts(o)



pts(p) = {o}

Andersen's Pointer Analysis

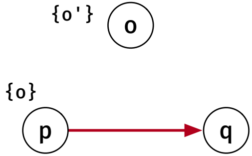
Constraint Type	C code	LLVM IR	Constraint rules
Address:	p = &o	%p = alloca //o	pts(p) = pts(p) ∪ {o}
Copy:	q = p	%q = bitcast %p	pts(q) = pts(q) ∪ pts(p)
Load:	q = *p	%q = load %p	∀ o ∈ pts(p): pts(q) = pts(o) ∪ pts(q)
Store:	*p = q	store %q, %p	∀ o ∈ pts(p): pts(o) = pts(q) ∪ pts(o)



pts(p) = {o}
pts(q) = {o}

Andersen’s Pointer Analysis

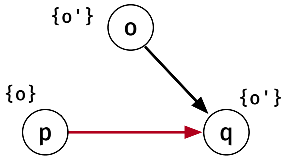
Constraint Type	C code	LLVM IR	Constraint rules
Address:	p = &o	%p = alloca //o	pts(p) = pts(p) U {o}
Copy:	q = p	%q = bitcast %p	pts(q) = pts(q) U pts(p)
Load:	q = *p	%q = load %p	∀ o ∈ pts(p): pts(q) = pts(o) U pts(q)
Store:	*p = q	store %q, %p	∀ o ∈ pts(p): pts(o) = pts(q) U pts(o)



pts(p) = {o}
pts(o) = {o'}

Andersen’s Pointer Analysis

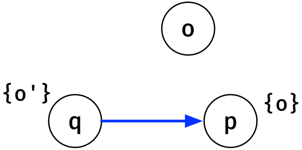
Constraint Type	C code	LLVM IR	Constraint rules
Address:	`p = &o`	`%p = alloca //o`	pts(p) = pts(p) ∪ {o}
Copy:	`q = p`	`%q = bitcast %p`	pts(q) = pts(q) ∪ pts(p)
Load:	`q = *p`	`%q = load %p`	∀ o ∈ pts(p): pts(q) = pts(o) ∪ pts(q)
Store:	`*p = q`	`store %q, %p`	∀ o ∈ pts(p): pts(o) = pts(q) ∪ pts(o)



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

Andersen’s Pointer Analysis

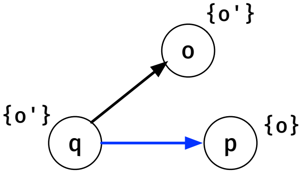
Constraint Type	C code	LLVM IR	Constraint rules
Address:	p = &o	%p = alloca //o	pts(p) = pts(p) ∪ {o}
Copy:	q = p	%q = bitcast %p	pts(q) = pts(q) ∪ pts(p)
Load:	q = *p	%q = load %p	∀ o ∈ pts(p): pts(q) = pts(o) ∪ pts(q)
Store:	*p = q	store %q, %p	∀ o ∈ pts(p): pts(o) = pts(q) ∪ pts(o)



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

Andersen's Pointer Analysis

Constraint Type	C code	LLVM IR	Constraint rules
Address:	p = &o	%p = alloca //o	pts(p) = pts(p) ∪ {o}
Copy:	q = p	%q = bitcast %p	pts(q) = pts(q) ∪ pts(p)
Load:	q = *p	%q = load %p	∀ o ∈ pts(p): pts(q) = pts(o) ∪ pts(q)
Store:	*p = q	store %q, %p	∀ o ∈ pts(p): pts(o) = pts(q) ∪ pts(o)



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

Compile C Code to LLVM IR

```c
void swap(char **p, char **q){
  char* t = *p;
       *p = *q;
       *q = t;
}
int main(){
      char a1, b1;
      char *a = &a1;
      char *b = &b1;
      swap(&a,&b);
}
```

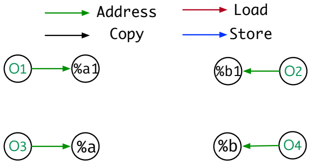
compile to

```llvm
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
}
```

*https://github.com/SVF-tools/SVF-Teaching/wiki/CodeGraph#2-llvm-ir-generation

Construct a Constraint Graph from LLVM IR

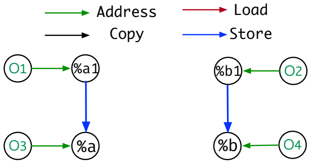
```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define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



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

Construct a Constraint Graph from LLVM IR

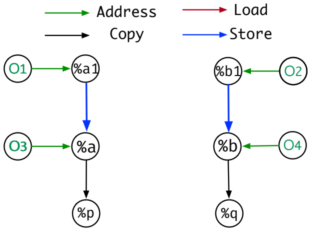
```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define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



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

Construct a Constraint Graph from LLVM IR

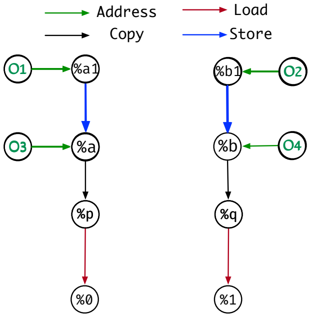
```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define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



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

Construct a Constraint Graph from LLVM IR

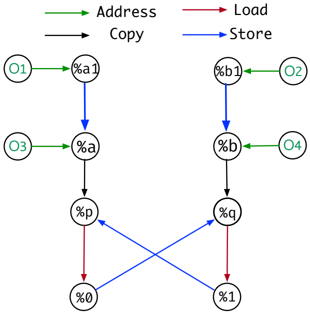
```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define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



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

Construct a Constraint Graph from LLVM IR

```
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
}
```

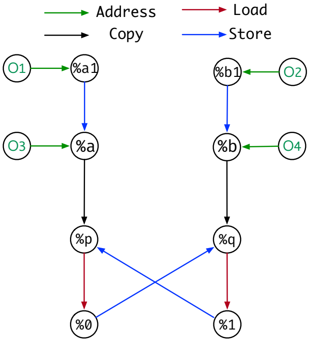


https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

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
}
```



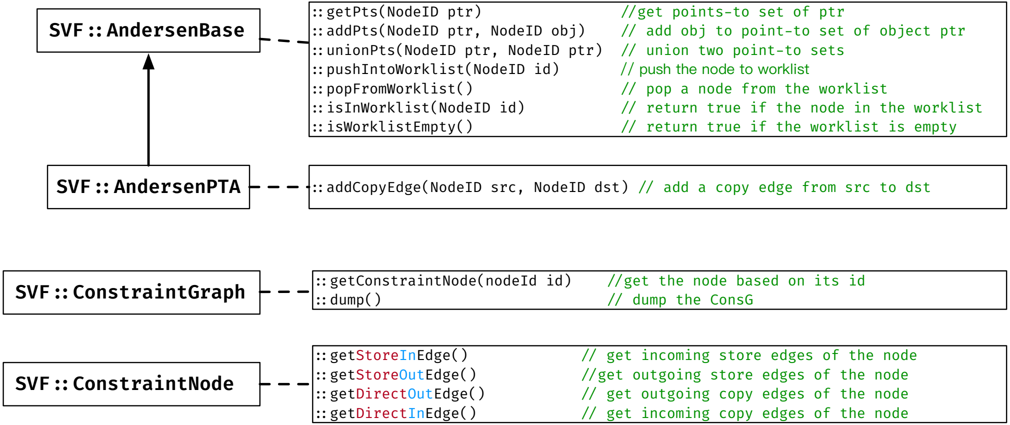

```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
foreach address p = &o do  // Address rule
      pts(p) = {o}
      pushIntoWorklist(p)
while WorkList ≠ ∅ do
   p ← popFromWorklist()
   foreach o ∈ pts(p) do
      foreach store *p = q do// Store rule
         if q→o ∉ E then
            E ← E ∪ {q→o}  // Add copy edge
            pushIntoWorklist(q)
      foreach load r = *p do  // Load rule
         if o→r ∉ E then
            E ← E ∪ {o→r}  // Add copy edge
            pushIntoWorklist(o)
   foreach p→x ∈ E do         // Copy rule
      pts(x) ← pts(x) ∪ pts(p)
      if pts(x) changed then
         pushIntoWorklist(x)
```

Andersen's Pointer Analysis

Constraint solving Algorithm

- A worklist holds a set of constraint graph nodes for processing
- Pop a node `p` from the worklist.
- Handle each incoming `store` edge and each outgoing `load` edge of node `p` by adding `copy` edges.
- Handle each outgoing `copy` edge of `p` by propagating points-to information.
- The constraint solving stops when no points-to set of a pointer is changed.

APIs for Implementing Andersen's analysis

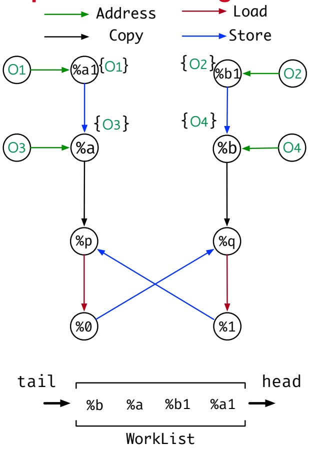


https://github.com/SVF-tools/SVF-Teaching/wiki/SVF-CPP-API#worklist-operations
https://github.com/SVF-tools/SVF-Teaching/wiki/SVF-CPP-API#points-to-set-operations
https://github.com/SVF-tools/SVF-Teaching/wiki/SVF-CPP-API#constraintgraph-constraintnode-and-constraintedge

Andersen's Pointer Analysis

Constraint graph before the while loop worklist solving

```
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
}
```

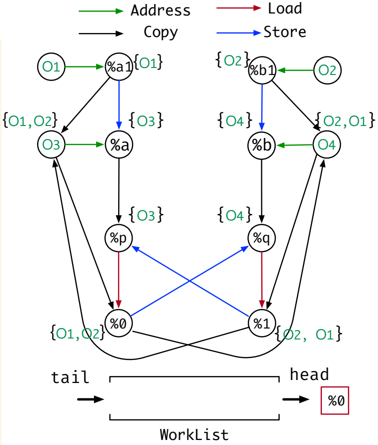



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
foreach address p = &o do  // Address rule
    pts(p) = {o}
    pushIntoWorklist(p)
while WorkList ≠ ∅ do
   p ← popFromWorklist()
   foreach o ∈ pts(p) do
      foreach store *p = q do// Store rule
         if q→o ∉ E then
            E ← E ∪ {q→o}  // Add copy edge
            pushIntoWorklist(q)
      foreach load r = *p do  // Load rule
         if o→r ∉ E then
            E ← E ∪ {o→r}  // Add copy edge
            pushIntoWorklist(o)
   foreach p→x ∈ E do        // Copy rule
      pts(x) ← pts(x) ∪ pts(p)
      if pts(x) changed then
         pushIntoWorklist(x)
```

Andersen's Pointer Analysis

Constraint graph after the while loop worklist solving

```
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
foreach address p = &o do  // Address rule
    pts(p) = {o}
    pushIntoWorklist(p)
while WorkList ≠ ∅ do
  p ← popFromWorklist()
  foreach o ∈ pts(p) do
    foreach store *p = q do// Store rule
      if q→o ∉ E then
        E ← E ∪ {q→o}  // Add copy edge
        pushIntoWorklist(q)
    foreach load r = *p do  // Load rule
      if o→r ∉ E then
        E ← E ∪ {o→r}  // Add copy edge
        pushIntoWorklist(o)
  foreach p→x ∈ E do         // Copy rule
    pts(x) ← pts(x) ∪ pts(p)
    if pts(x) changed then
      pushIntoWorklist(x)
```

What's next?

- (1) Understand data-dependence in today's slides
- (2) Implement Andersen's pointer analysis, i.e., Task in Assignment 3
  - Refer to 'Assignment-3.pdf' on Canvas to know more about Assignment 3.