

Program Analysis with Set Constraints

Ravi Chugh

Set-constraint based analysis

- Another technique for computing information about program variables
- Phase 1: constraint generation *
- Create set variables corresponding to program
- Add inclusion constraints between these sets $S_1 \subseteq S_2$
- Usually a local, syntax-directed process (ASTs vs CFGs)
- Phase 2: constraint resolution
- Solve for values of all set variables
- Extends naturally to inter-procedural analysis

Constant propagation

```
int abs(int i) {
    if (...) { return i; }
    else { return -i; }
}
int id(int j) {
    return j;
}
void main() {
    int a = 1, b = 2;
    int x = abs(a);
    int y = id(b);
    ... use x ...
    ... use y ...
}
```

- Want to determine whether x and y are constant values when they are used
- We will build a flow-insensitive analysis

Set constraints

- Terms

$t := c, d, e$ (constant)
| x (set variable)
| $C(t_1, \dots, t_n)$ (constructed term)

1, 2, "hi"
int x, y
C(x, y, e)

- Constraints

t_1 \leq t_2 (set inclusion)

$S_1 \subseteq S_2$

- Constructors

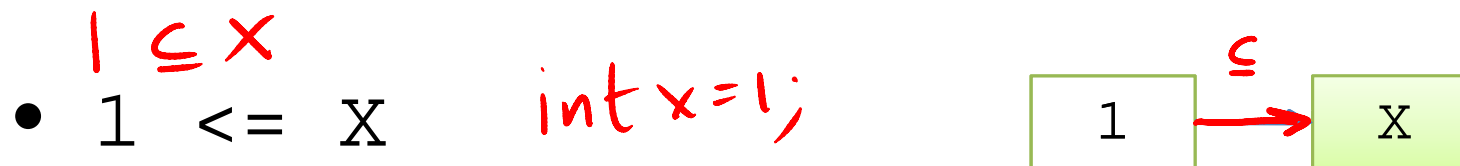
- $C(v_1, \dots, v_n)$ is an n-arg ctor C with variances v_i
- v_i is either + (covariant) or – (contravariant)
- Covariance corresponds to “forwards flow”
- Contravariance corresponds to “backwards flow”

Set constraints and graph reachability

- Tight correspondence between set-inclusion constraints and edges in a flow graph

Set constraints and graph reachability

- Tight correspondence between set-inclusion constraints and edges in a flow graph

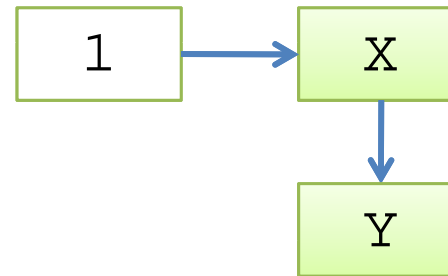


Set constraints and graph reachability

- Tight correspondence between set-inclusion constraints and edges in a flow graph

- $1 \leq X$

- $X \leq Y$ *int y=x*

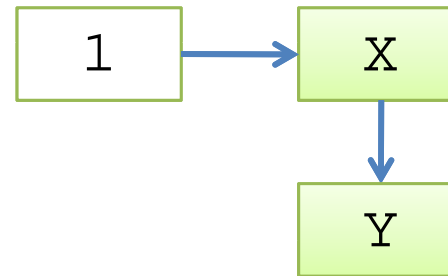


Set constraints and graph reachability

- Tight correspondence between set-inclusion constraints and edges in a flow graph

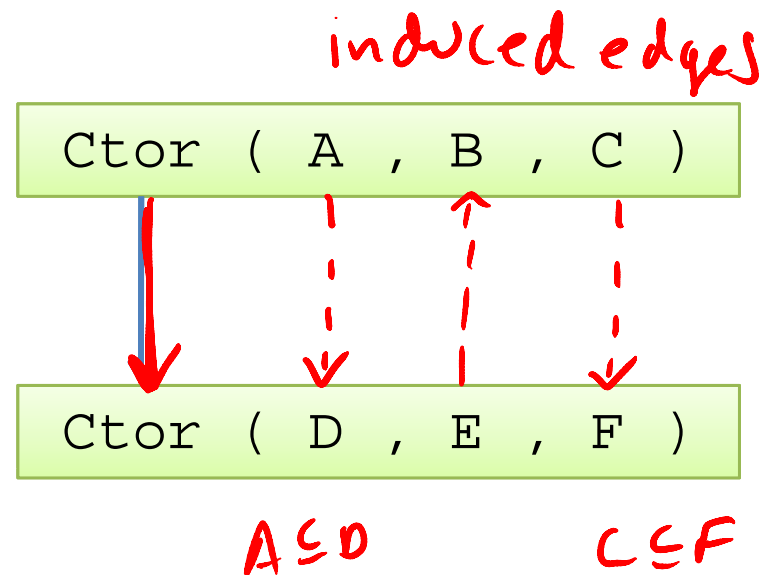
- $1 \leq X$

- $X \leq Y$



- $\text{Ctor}(\underline{A}, \underline{B}, \underline{C}) \leq \text{Ctor}(\underline{D}, \underline{E}, \underline{F})$

where $\text{Ctor}(\underline{+}, \underline{-}, \underline{+})$

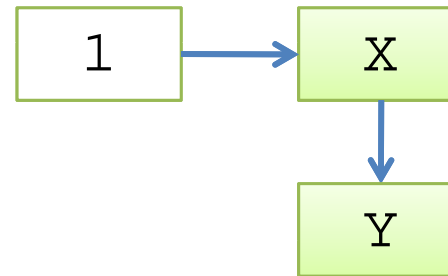


Set constraints and graph reachability

- Tight correspondence between set-inclusion constraints and edges in a flow graph

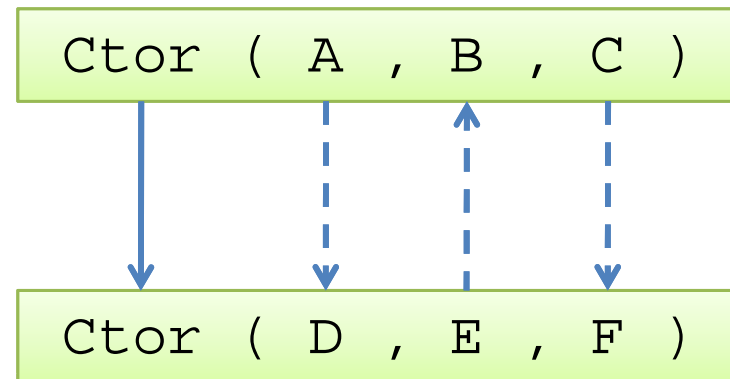
- $1 \leq X$

- $X \leq Y$



- $\text{Ctor}(A, B, C) \leq \text{Ctor}(D, E, F)$

where $\text{Ctor}(+, -, +)$



Constraint resolution

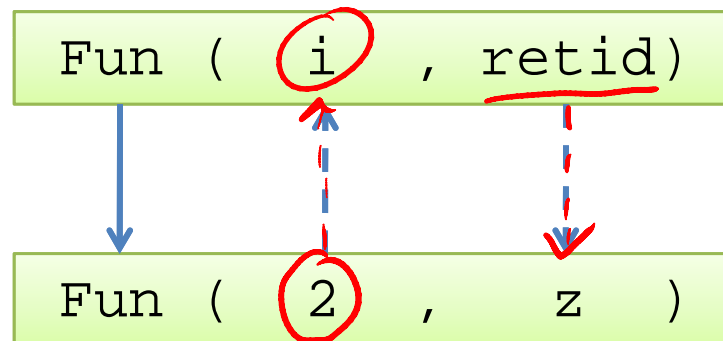
- System of constraints **Con** \rightarrow explicit constraints
- Additional constraints added by following rules
- 1) Transitivity of sets
 - $C(1,2,3) \subseteq X$
 - $X \subseteq D(4,5)$
 - **Con** with $\underline{x} \leq \underline{y} \wedge \underline{y} \leq \underline{z} \rightarrow \text{Con} \wedge \underline{x} \leq \underline{z}$
- 2) Constructed terms
 - **Con** with $\underline{C(\dots, x_i, \dots)} \leq \underline{C(\dots, y_i, \dots)}$
 - $\rightarrow \text{Con} \wedge_i \text{straint}_i$
 - straint_i is $x_i \leq y_i$ if C covariant in \underline{i} $x_i \leq y_i$
 - straint_i is $y_i \leq x_i$ if C contravariant in \underline{i} $x_i \geq y_i$
- 3) Inconsistent terms
 - **Con** with $\underline{C(\dots)} \leq \underline{D(\dots)} \rightarrow \text{Inconsistent}$

Fun constructor

- For simplicity, assume all functions take one arg
- Define constructor $\text{Fun}(-, +)$
- Places for function input and output 1) func defs
2) func calls
- Encoding a function call: `int z = id(2);`
① `Fun(i, retid)` ② `<=` `Fun(2, z)`

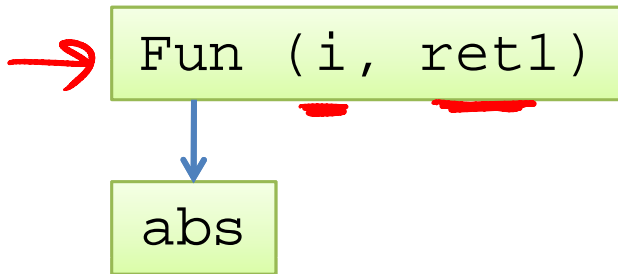
Fun constructor

- For simplicity, assume all functions take one arg
- Define constructor `Fun (- , +)`
- Places for function input and output
- Encoding a function call: `int z = id(2) + 5;`
`Fun(i, retid) <= Fun(2, z)`
- By contravariance, the actual 2 flows to i
- By covariance, the return value of id flows to z



```
int abs(int i) {
    if (...) { return i; }
    else { return -i; }
}
int id(int j) {
    return j;
}
void main() {
    int a = 1, b = 2;
    int x = abs(a);
    int y = id(b);
    ... use x ...
    ... use y ...
}
```

```
int abs(int i) { abs = function Fun(i,ret1) <= abs
  if (...) { return i; }
  else { return -i; }
}
int id(int j) {
  return j;
}
void main() {
  int a = 1, b = 2;
  int x = abs(a);
  int y = id(b);
  ... use x ...
  ... use y ...
}
```

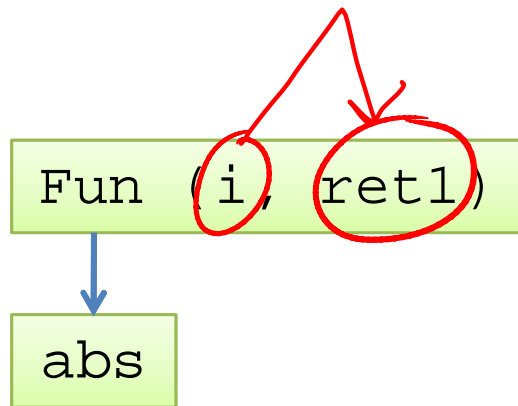


```
int abs(int i) { ret1 = i;  
  if (...) { return i; }  
  else { return -i; }  
}
```

```
int id(int j) {  
  return j;  
}
```

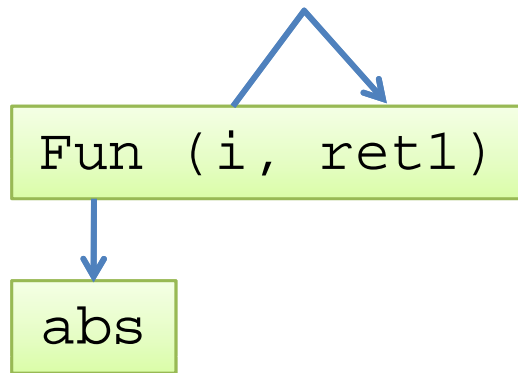
```
void main() {  
  int a = 1, b = 2;  
  int x = abs(a);  
  int y = id(b);  
  ... use x ...  
  ... use y ...  
}
```

```
Fun(i,ret1) <= abs  
i = ret1
```



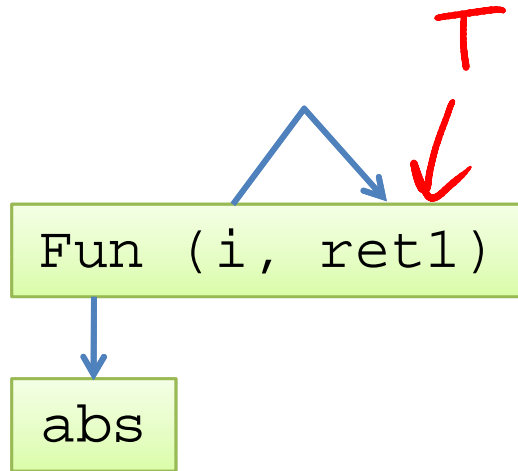
```
int abs(int i) {  
    if (...) { return i; }  
    else { return -i; }  
}  
int id(int j) {  
    return j;  
}  
void main() {  
    int a = 1, b = 2;  
    int x = abs(a);  
    int y = id(b);  
    ... use x ...  
    ... use y ...  
}
```

```
Fun(i,ret1) <= abs  
            i <= ret1
```



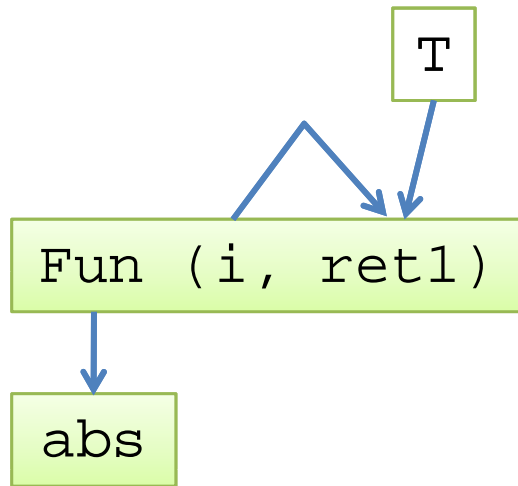
```
int abs(int i) {  
    if (...) { return i; }  
    else { return -i; }  
}  
int id(int j) {  
    return j;  
}  
void main() {  
    int a = 1, b = 2;  
    int x = abs(a);  
    int y = id(b);  
    ... use x ...  
    ... use y ...  
}
```

```
Fun(i,ret1) <= abs  
            i <= ret1  
T <= ret1
```



```
int abs(int i) {  
    if (...) { return i; }  
    else { return -i; }  
}  
int id(int j) {  
    return j;  
}  
void main() {  
    int a = 1, b = 2;  
    int x = abs(a);  
    int y = id(b);  
    ... use x ...  
    ... use y ...  
}
```

```
Fun(i,ret1) <= abs  
            i <= ret1  
            T <= ret1
```

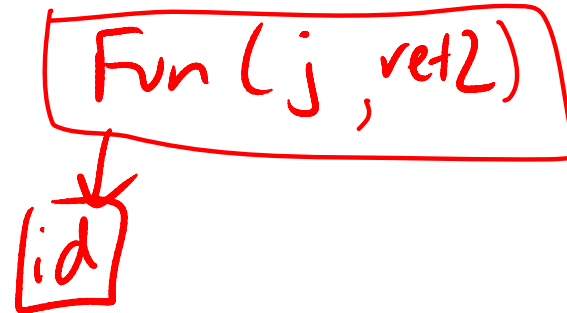
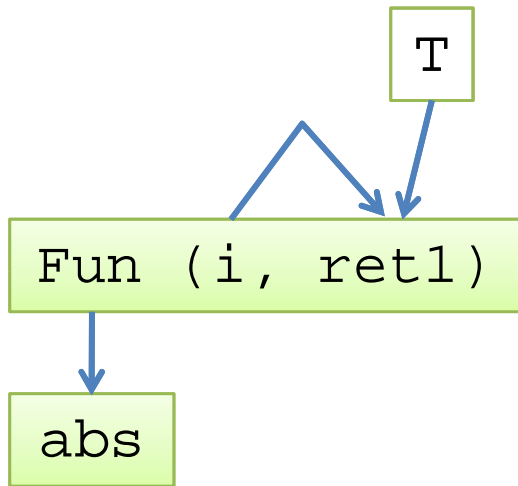


```
int abs(int i) {  
    if (...) { return i; }  
    else { return -i; }  
}
```

```
int id(int j) {  
    return j;  
}
```

```
void main() {  
    int a = 1, b = 2;  
    int x = abs(a);  
    int y = id(b);  
    ... use x ...  
    ... use y ...  
}
```

```
Fun(i,ret1) <= abs  
            i <= ret1  
            T <= ret1
```



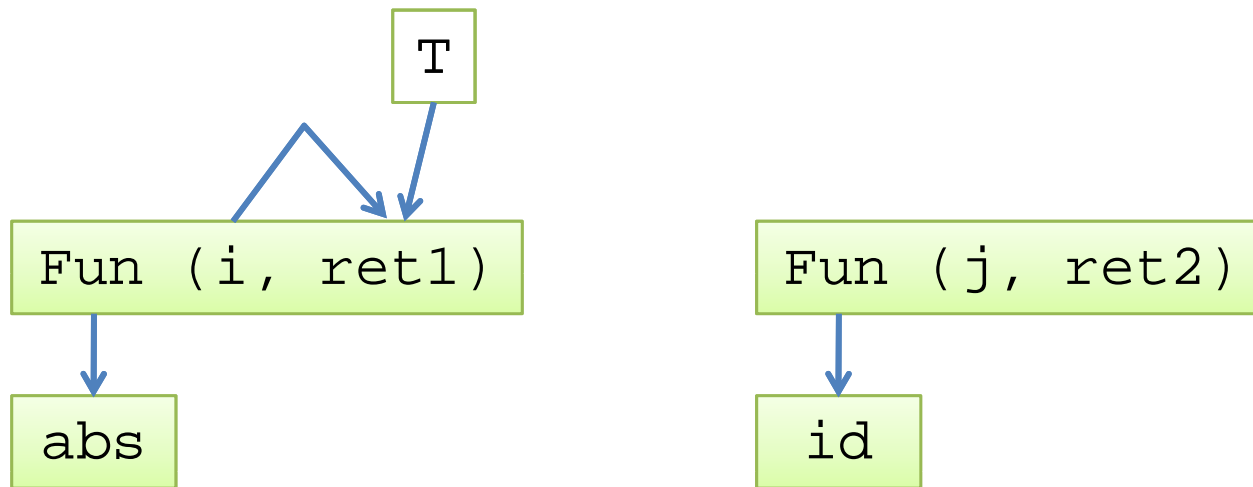
```
int abs(int i) {  
    if (...) { return i; }  
    else { return -i; }  
}
```

```
int id(int j) {  
    return j;  
}
```

```
void main() {  
    int a = 1, b = 2;  
    int x = abs(a);  
    int y = id(b);  
    ... use x ...  
    ... use y ...  
}
```

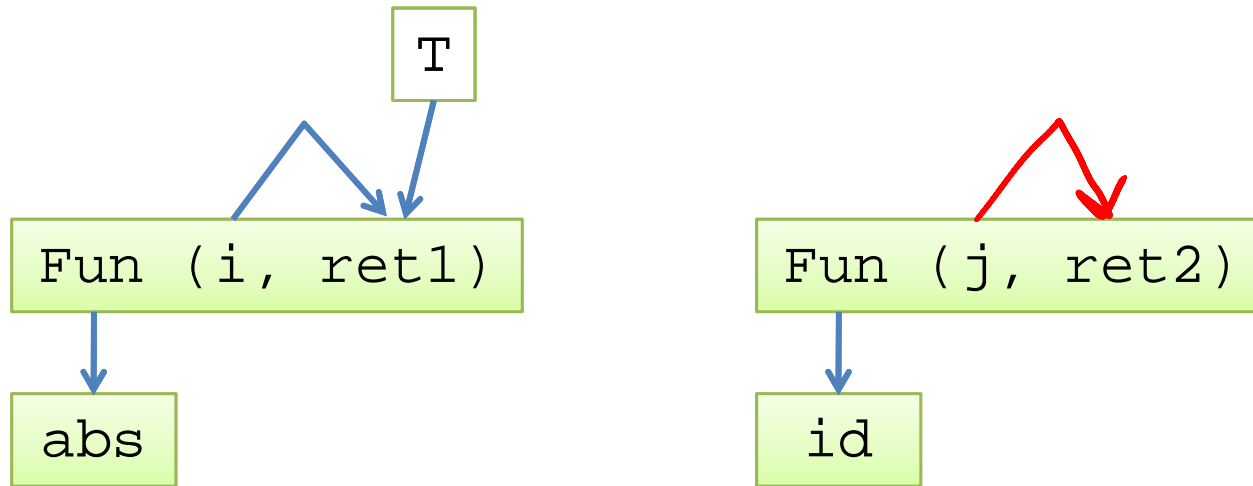
```
Fun(i,ret1) <= abs  
    i <= ret1  
    T <= ret1
```

```
Fun(j,ret2) <= id
```



```
int abs(int i) {  
  if (...) { return i; }  
  else { return -i; }  
}  
int id(int j) {  
  return j;  
}  
void main() {  
  int a = 1, b = 2;  
  int x = abs(a);  
  int y = id(b);  
  ... use x ...  
  ... use y ...  
}
```

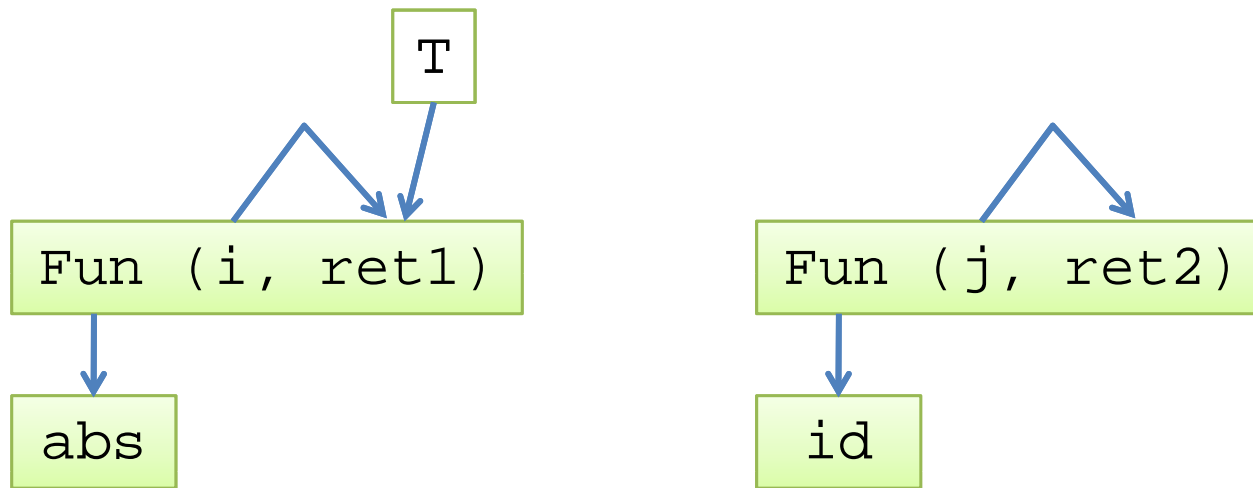
```
Fun(i,ret1) <= abs  
           i <= ret1  
           T <= ret1  
  
Fun(j,ret2) <= id
```



```
int abs(int i) {  
    if (...) { return i; }  
    else { return -i; }  
}  
int id(int j) {  
    return j;  
}  
void main() {  
    int a = 1, b = 2;  
    int x = abs(a);  
    int y = id(b);  
    ... use x ...  
    ... use y ...  
}
```

```
Fun(i,ret1) <= abs  
    i <= ret1  
    T <= ret1
```

```
Fun(j,ret2) <= id  
    j <= ret2
```



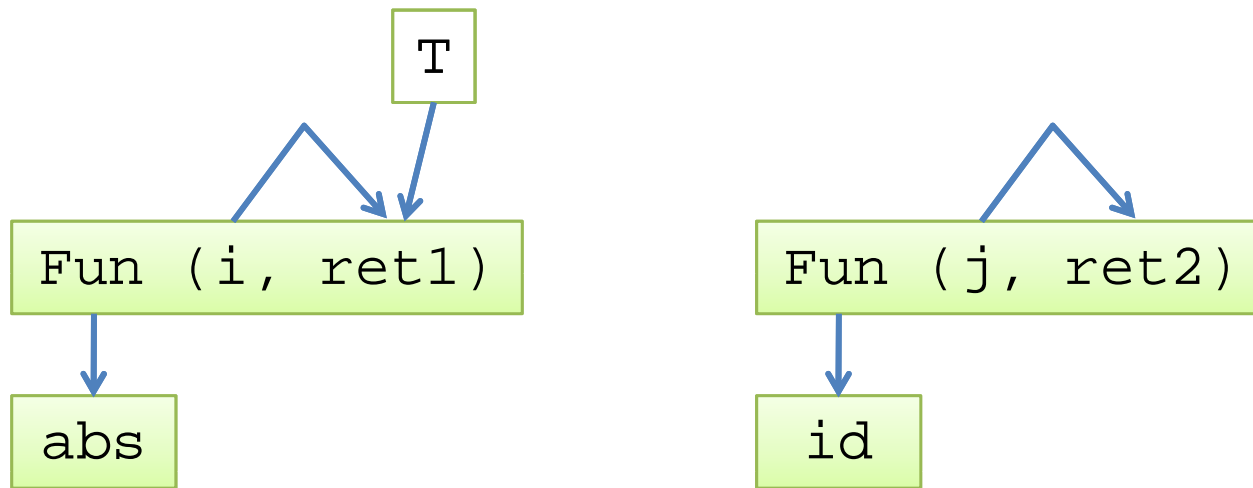
```
int abs(int i) {  
    if (...) { return i; }  
    else { return -i; }  
}
```

```
int id(int j) {  
    return j;  
}
```

```
→ void main() {  
    int a = 1, b = 2;  
    int x = abs(a);  
    int y = id(b);  
    ... use x ...  
    ... use y ...  
}
```

```
Fun(i,ret1) <= abs  
    i <= ret1  
    T <= ret1
```

```
Fun(j,ret2) <= id  
    j <= ret2
```



```

int abs(int i) {
    if (...) { return i; }
    else { return -i; }
}
int id(int j) {
    return j;
}
void main() {
    int a = 1, b = 2;
    int x = abs(a);
    int y = id(b);
    ... use x ...
    ... use y ...
}

```

```

Fun(i,ret1) <= abs
    i <= ret1
    T <= ret1

```

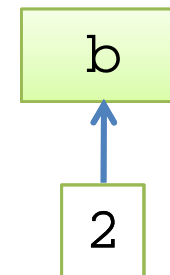
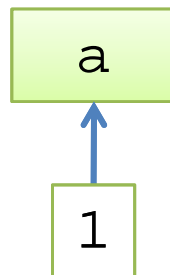
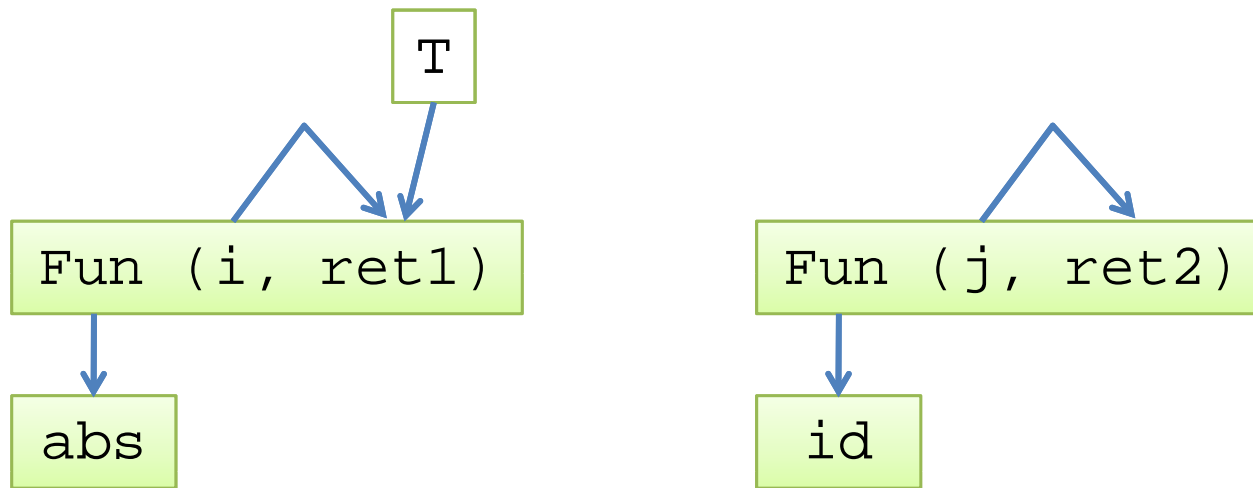
```

Fun(j,ret2) <= id
    j <= ret2

```

1 <= a

2 <= b

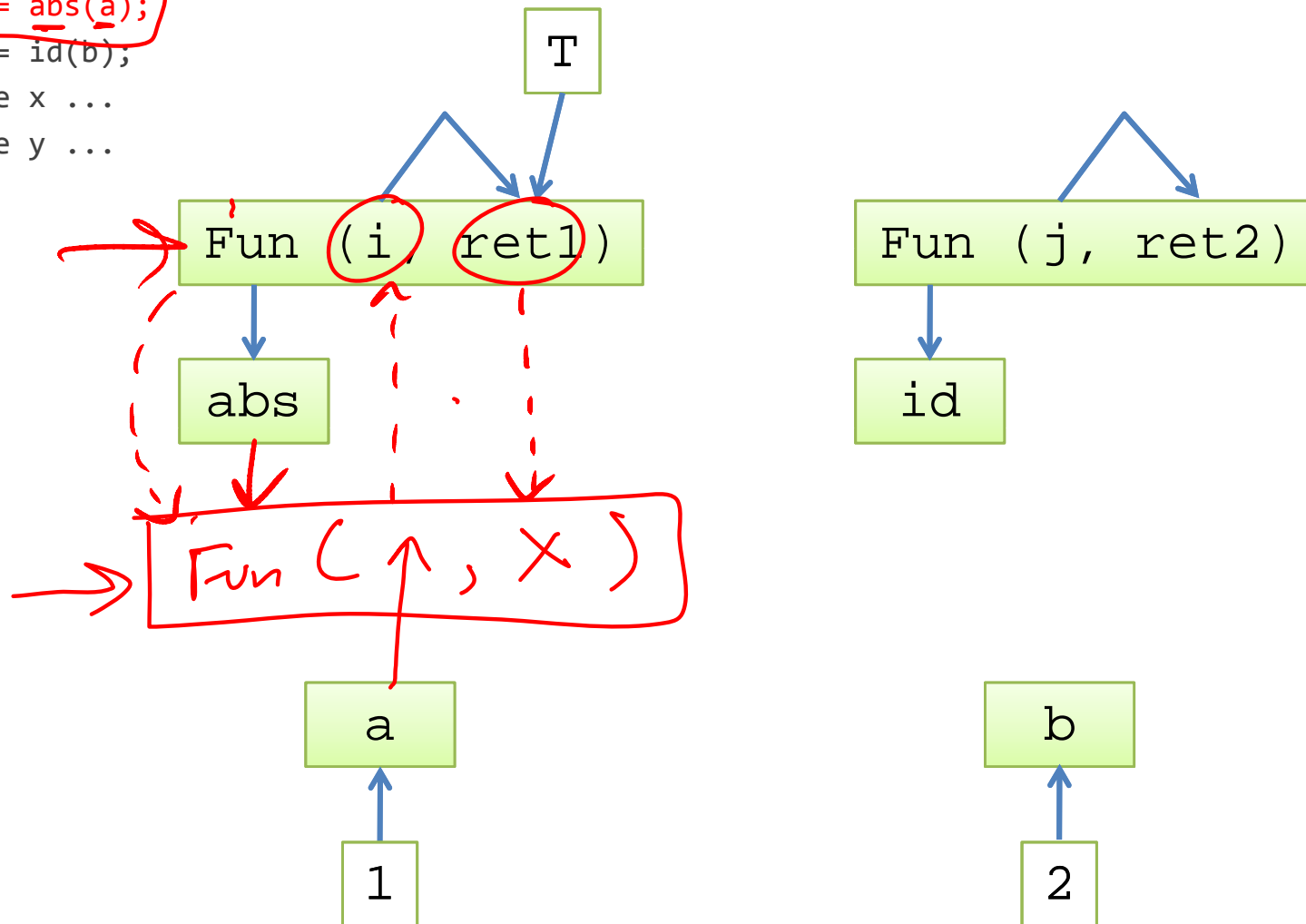


```
int abs(int i) {
  if (...) { return i; }
  else { return -i; }
}
int id(int j) {
  return j;
}
void main() {
  int a = 1, b = 2;
  int x = abs(a);
  int y = id(b);
  ... use x ...
  ... use y ...
}
```

```
Fun(i,ret1) <= abs
  i <= ret1
  T <= ret1

Fun(j,ret2) <= id
  j <= ret2

1 <= a
2 <= b
```



```

int abs(int i) {
    if (...) { return i; }
    else { return -i; }
}
int id(int j) {
    return j;
}
void main() {
    int a = 1, b = 2;
    int x = abs(a);
    int y = id(b);
    ... use x ...
    ... use y ...
}

```

```

Fun(i,ret1) <= abs
    i <= ret1
    T <= ret1

```

```

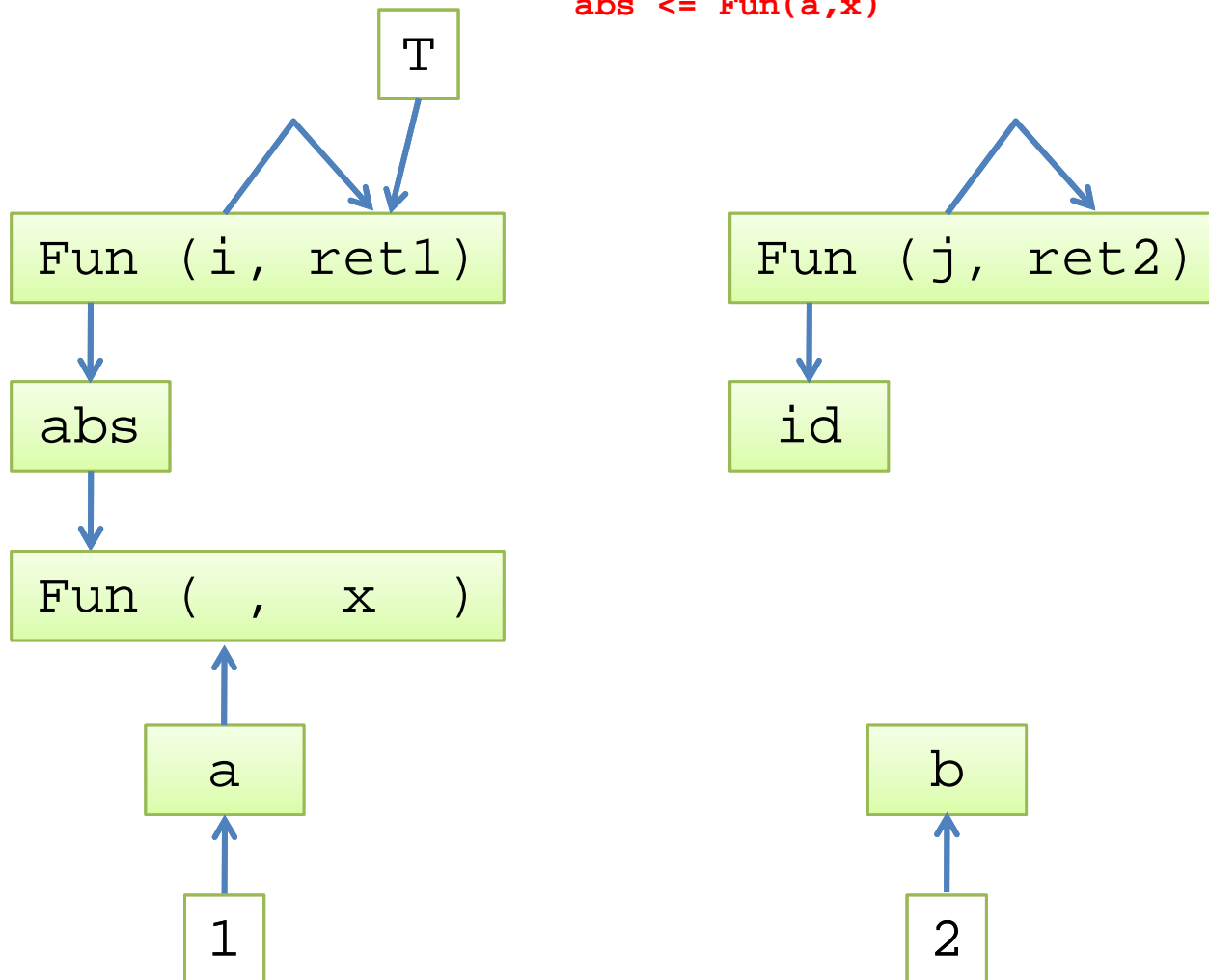
Fun(j,ret2) <= id
    j <= ret2

```

```
1 <= a
```

```
2 <= b
```

```
abs <= Fun(a,x)
```



```

int abs(int i) {
  if (...) { return i; }
  else { return -i; }
}
int id(int j) {
  return j;
}
void main() {
  int a = 1, b = 2;
  int x = abs(a);
  int y = id(b);
  ... use x ...
  ... use y ...
}

```

```

Fun(i,ret1) <= abs
  i <= ret1
  T <= ret1

```

```

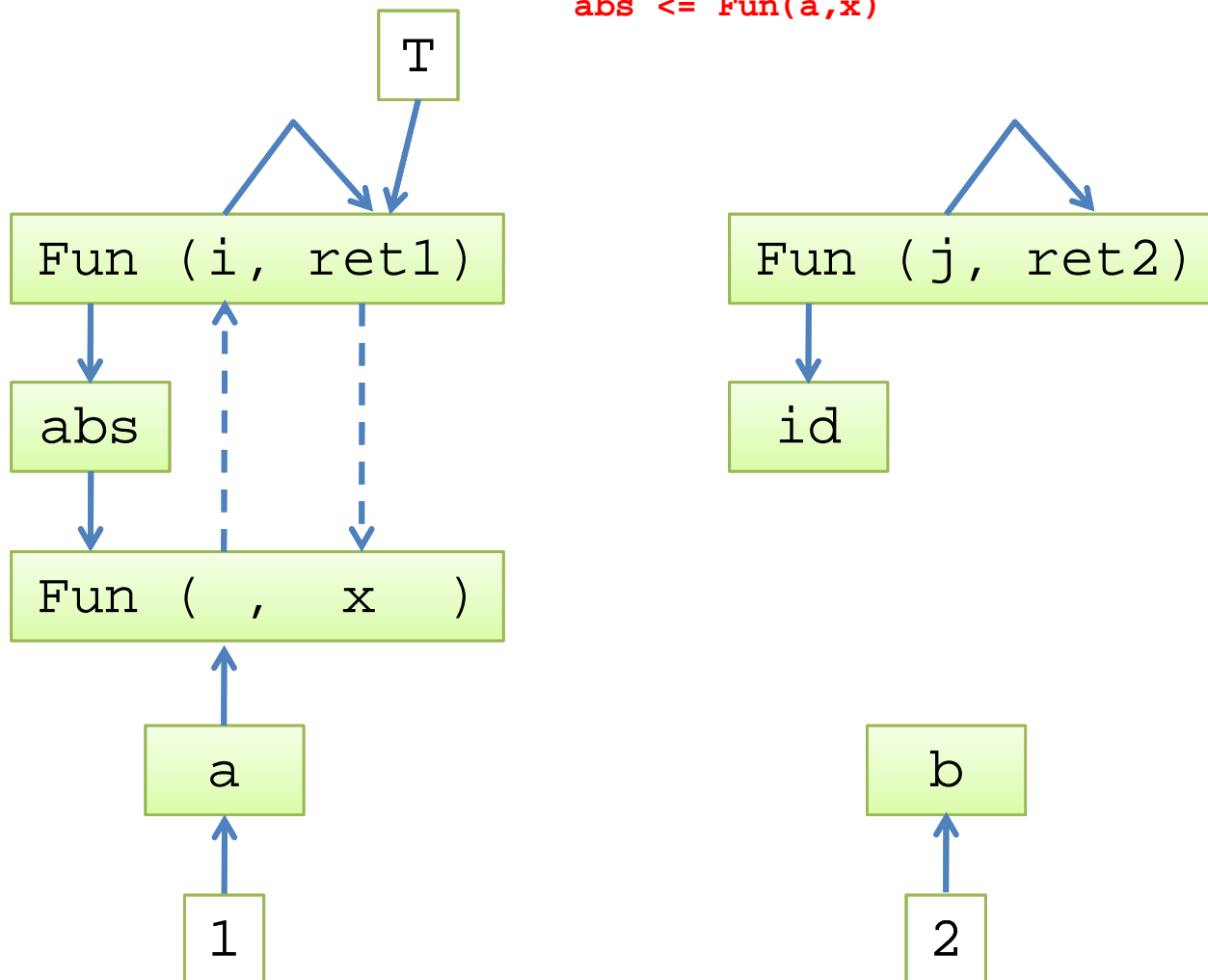
Fun(j,ret2) <= id
  j <= ret2

```

```
1 <= a
```

```
2 <= b
```

```
abs <= Fun(a,x)
```



```

int abs(int i) {
  if (...) { return i; }
  else { return -i; }
}
int id(int j) {
  return j;
}
void main() {
  int a = 1, b = 2;
  int x = abs(a);
  int y = id(b);
  ... use x ...
  ... use y ...
}

```

```

Fun(i,ret1) <= abs
  i <= ret1
  T <= ret1

```

```

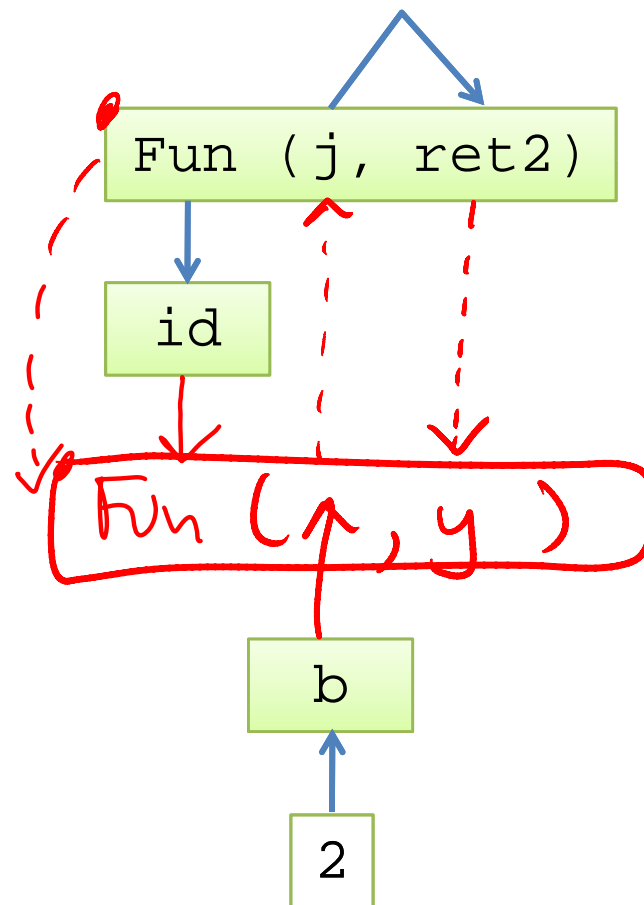
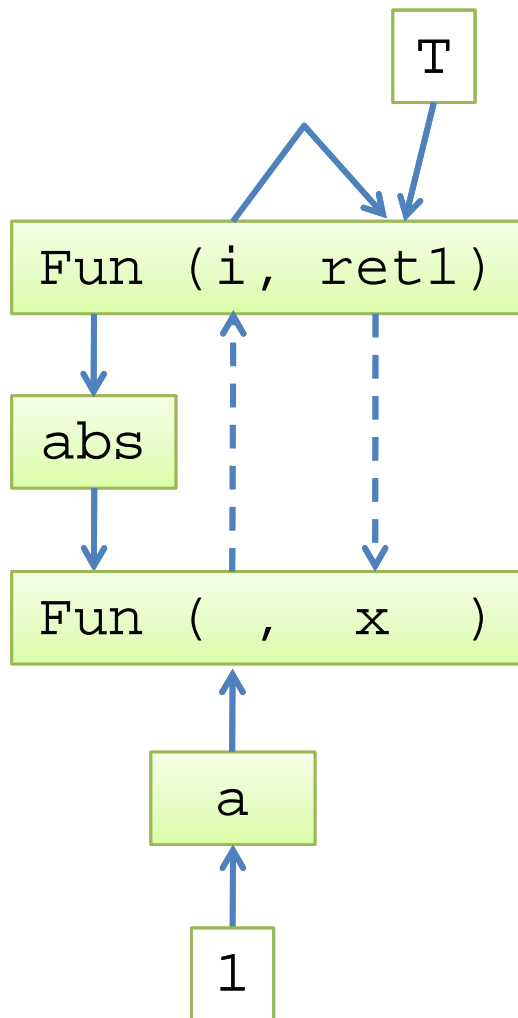
Fun(j,ret2) <= id
  j <= ret2

```

```

1 <= a
2 <= b
abs <= Fun(a,x)

```



```

int abs(int i) {
  if (...) { return i; }
  else { return -i; }
}
int id(int j) {
  return j;
}
void main() {
  int a = 1, b = 2;
  int x = abs(a);
  int y = id(b);
  ... use x ...
  ... use y ...
}

```

```

Fun(i,ret1) <= abs
  i <= ret1
  T <= ret1

```

```

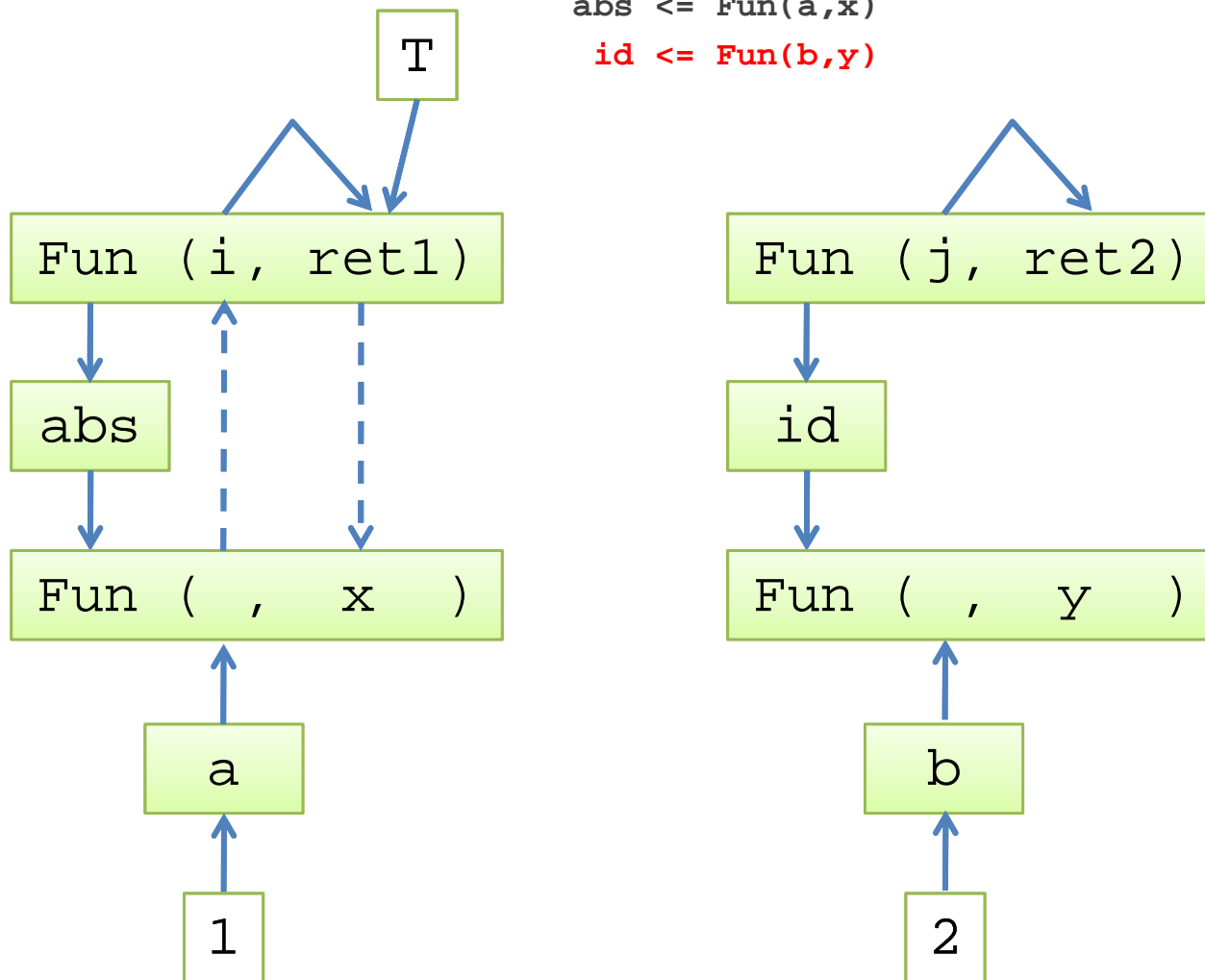
Fun(j,ret2) <= id
  j <= ret2

```

```

1 <= a
2 <= b
abs <= Fun(a,x)
id <= Fun(b,y)

```



```

int abs(int i) {
  if (...) { return i; }
  else { return -i; }
}
int id(int j) {
  return j;
}
void main() {
  int a = 1, b = 2;
  int x = abs(a);
  int y = id(b);
  ... use x ...
  ... use y ...
}

```

```

Fun(i,ret1) <= abs
  i <= ret1
  T <= ret1

```

```

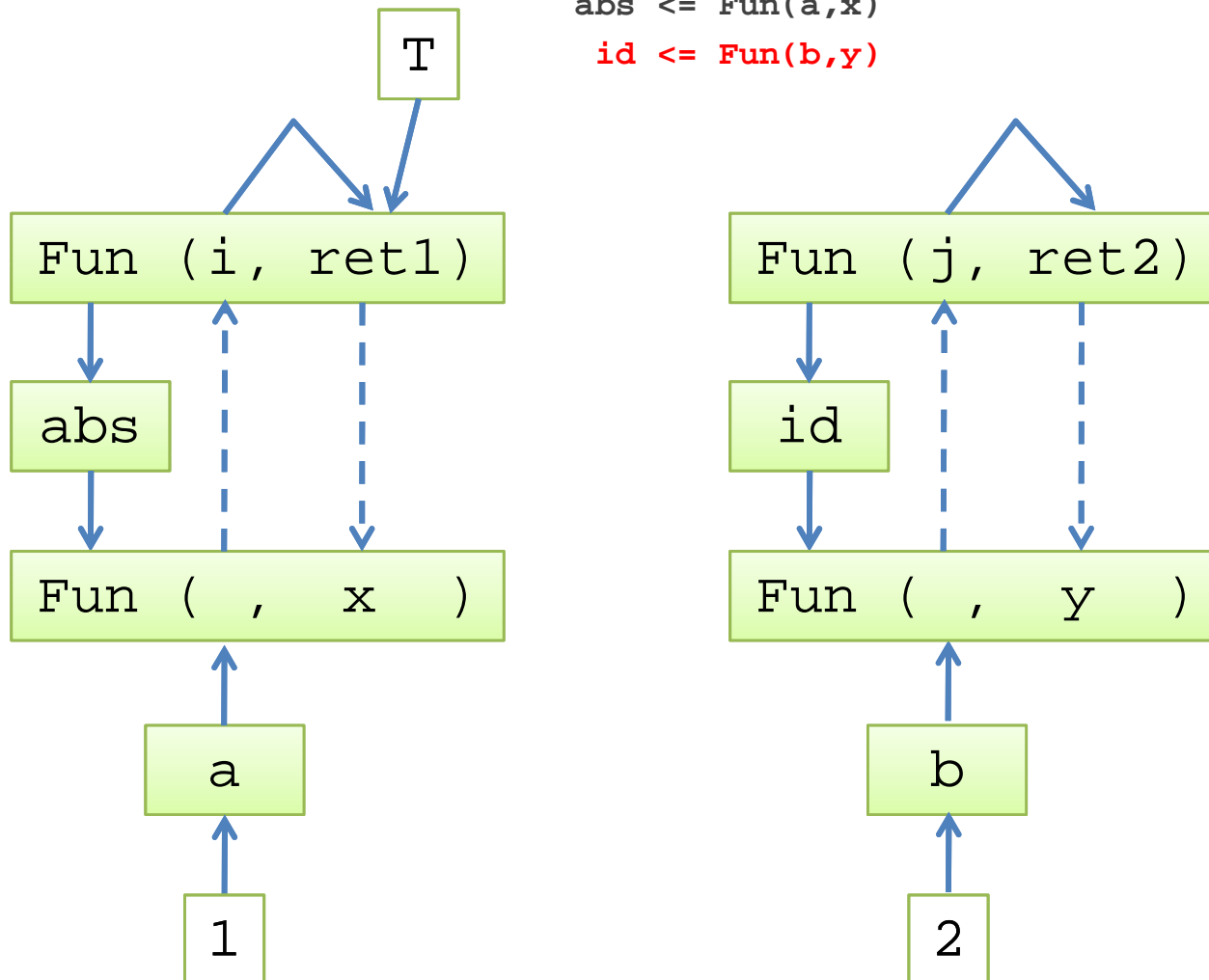
Fun(j,ret2) <= id
  j <= ret2

```

```

1 <= a
2 <= b
abs <= Fun(a,x)
id <= Fun(b,y)

```



```

int abs(int i) {
  if (...) { return i; }
  else { return -i; }
}
int id(int j) {
  return j;
}
void main() {
  int a = 1, b = 2;
  int x = abs(a);
  int y = id(b);
  ... use x ...
  ... use y ...
}

```

```

Fun(i,ret1) <= abs
  i <= ret1
  T <= ret1

```

```

Fun(j,ret2) <= id
  j <= ret2

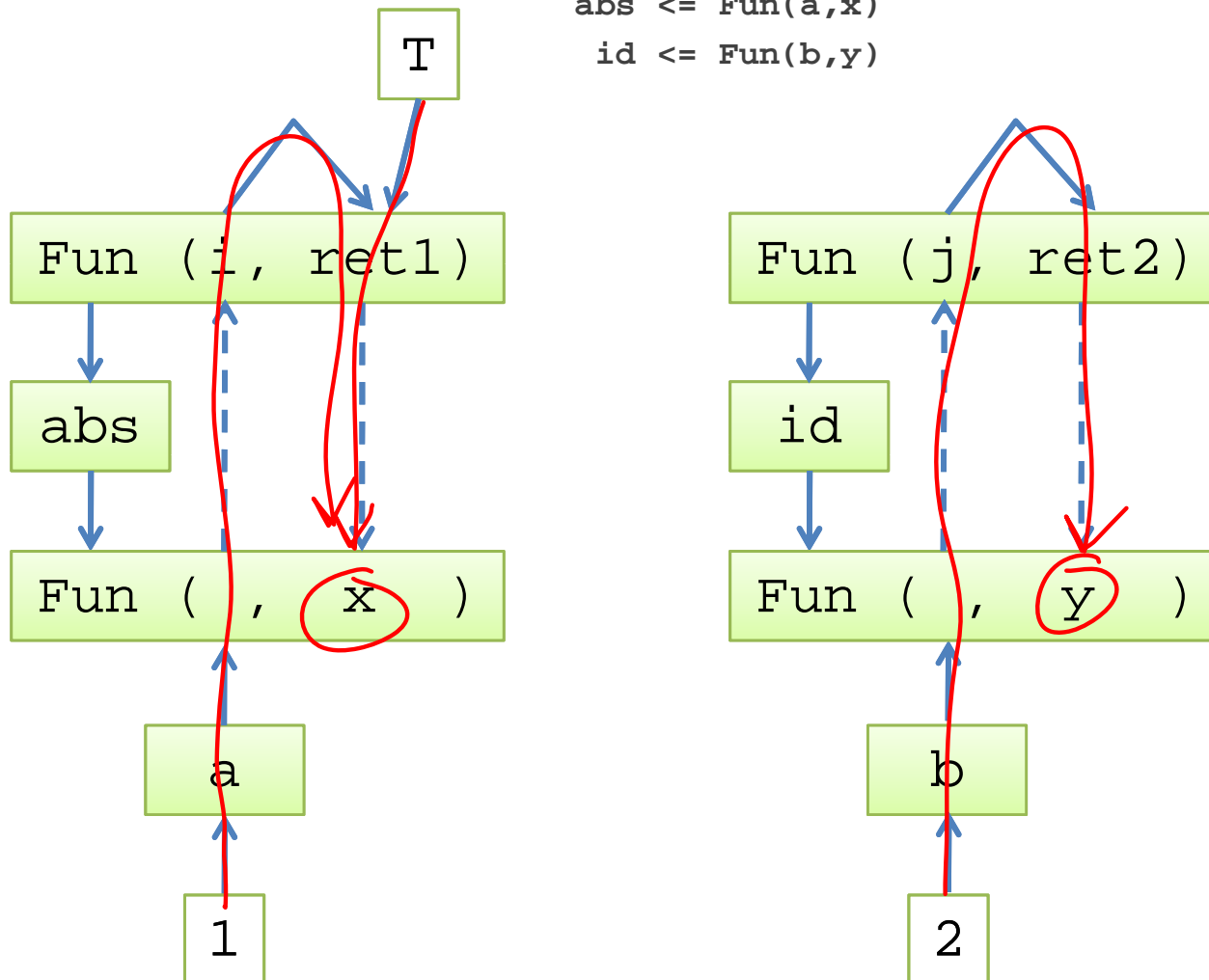
```

```

1 <= a
2 <= b
abs <= Fun(a,x)
id <= Fun(b,y)

```

[[x]] = {1, T}
 [[y]] = {2}



```

int abs(int i) {
  if (...) { return i; }
  else { return -i; }
}
int id(int j) {
  return j;
}
void main() {
  int a = 1, b = 2;
  int x = abs(a);
  int y = id(b);
  ... use x ...
  ... use y ...
}

```

```

Fun(i,ret1) <= abs
  i <= ret1
  T <= ret1

```

```

Fun(j,ret2) <= id
  j <= ret2

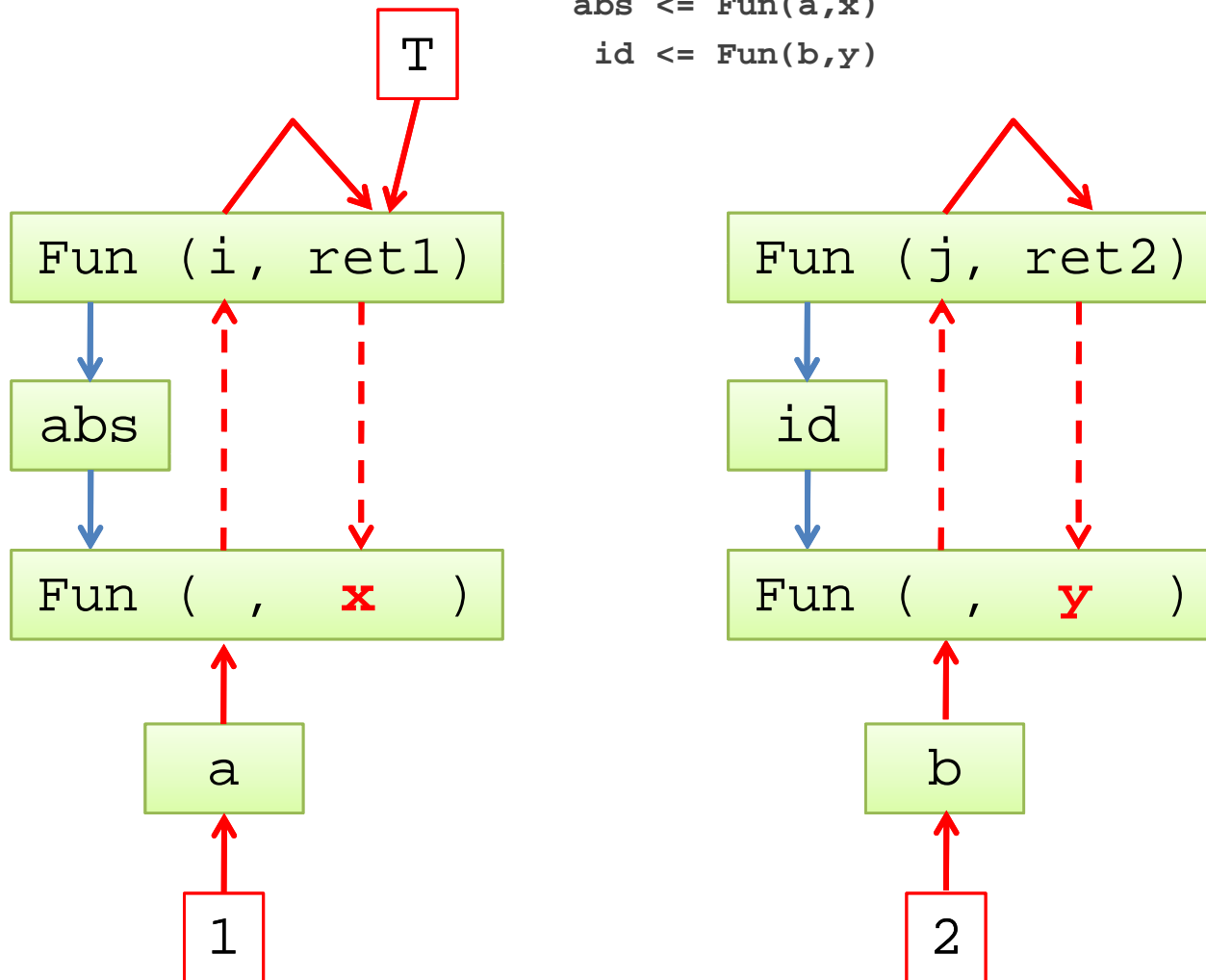
```

```

1 <= a
2 <= b
abs <= Fun(a,x)
id <= Fun(b,y)

```

[[x]] = {1, T} ✗
 [[y]] = {2} ✓



Pointers

- assignments
- func calls/defs
- return
- unary/binary expressions

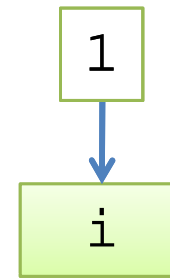
- Handle pointers with a Ref (-, +) constructor
- Two args correspond to set and get operations

```
int i = 1;
int *p = &i;
*p = 2;
int j = *p;
```

Pointers

- Handle pointers with a Ref (- , +) constructor
- Two args correspond to set and get operations

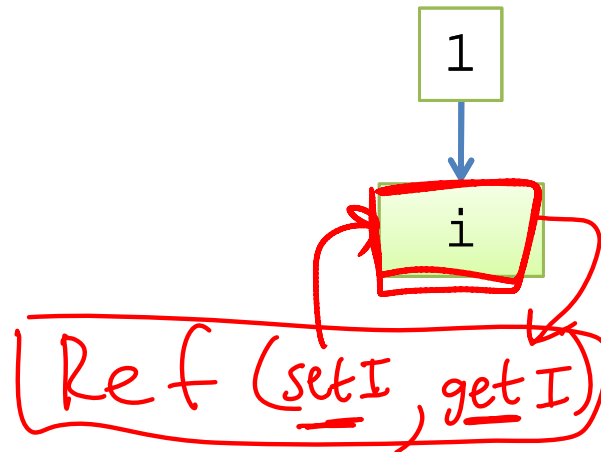
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a Ref (- , +) constructor
- Two args correspond to set and get operations

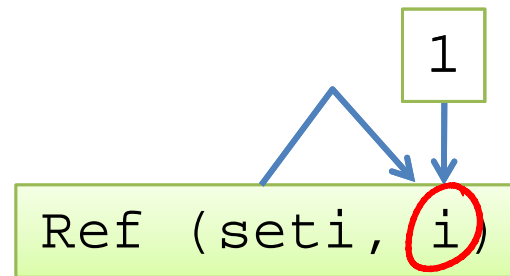
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a Ref (- , +) constructor
- Two args correspond to set and get operations

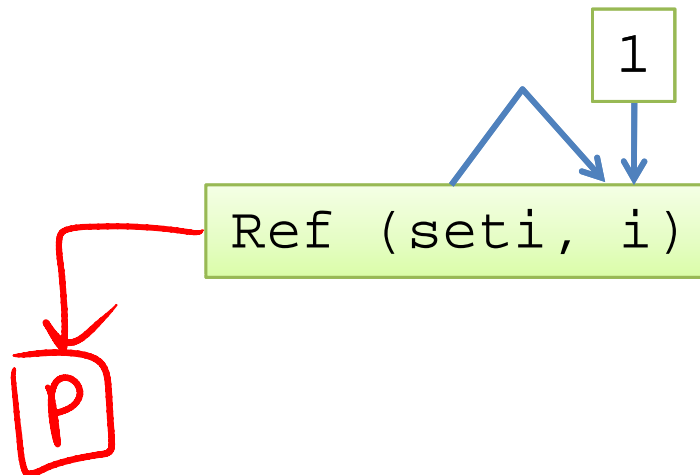
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a `Ref (-, +)` constructor
- Two args correspond to set and get operations

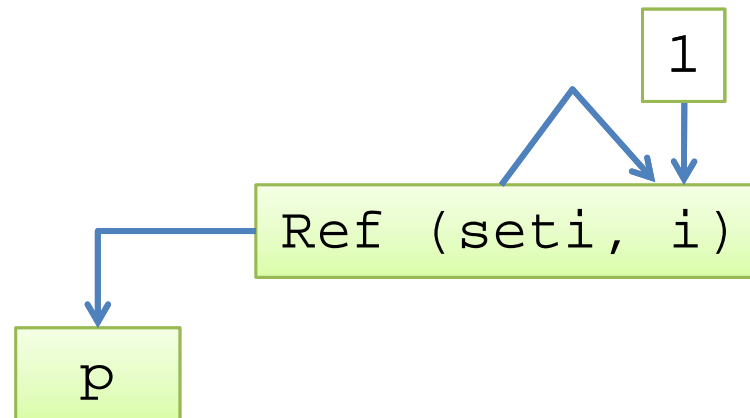
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a `Ref (-, +)` constructor
- Two args correspond to set and get operations

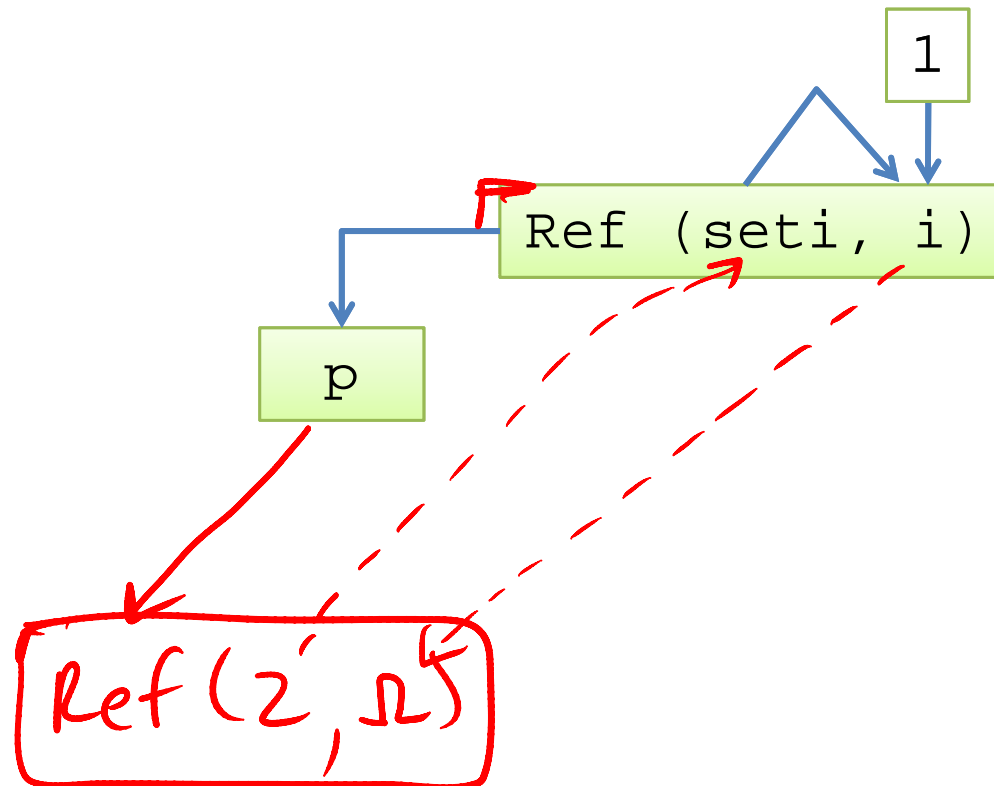
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a `Ref(-, +)` constructor
- Two args correspond to set and get operations

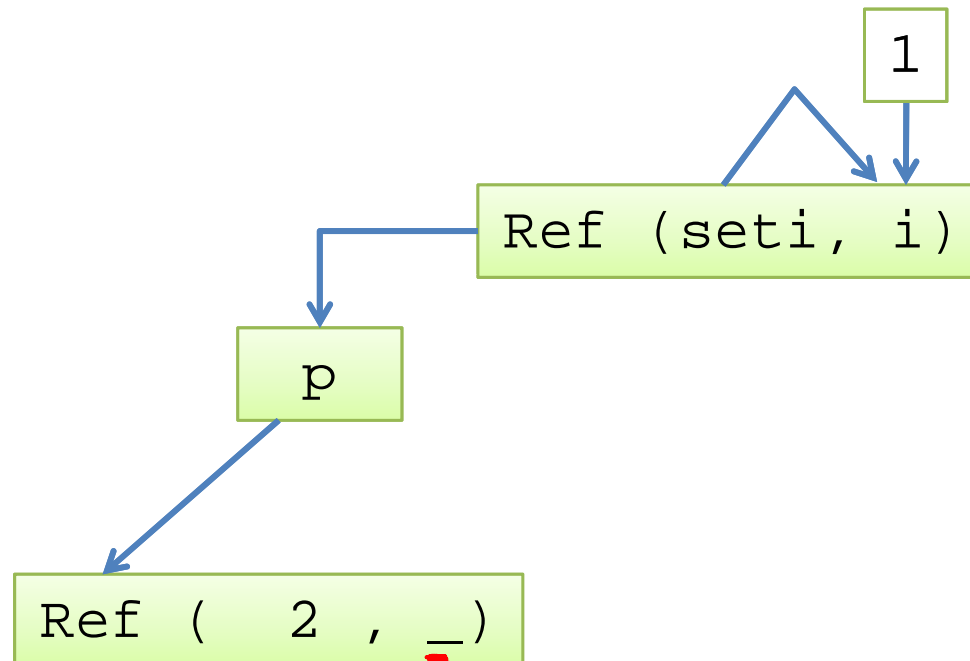
```
int i = 1;  
int *p = &i;  
→ *p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a `Ref (-, +)` constructor
- Two args correspond to set and get operations

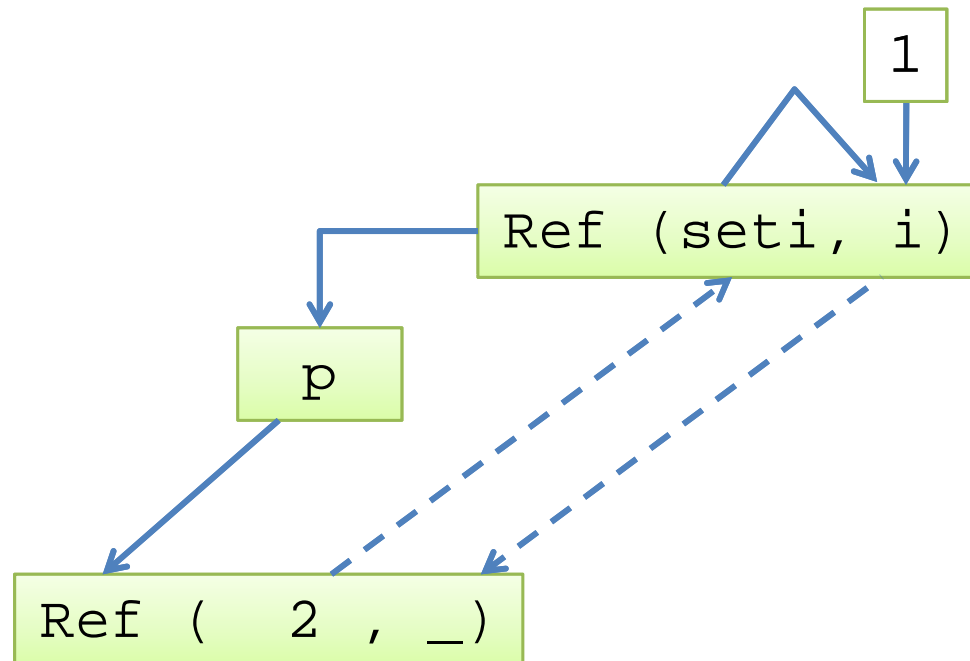
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a `Ref (-, +)` constructor
- Two args correspond to set and get operations

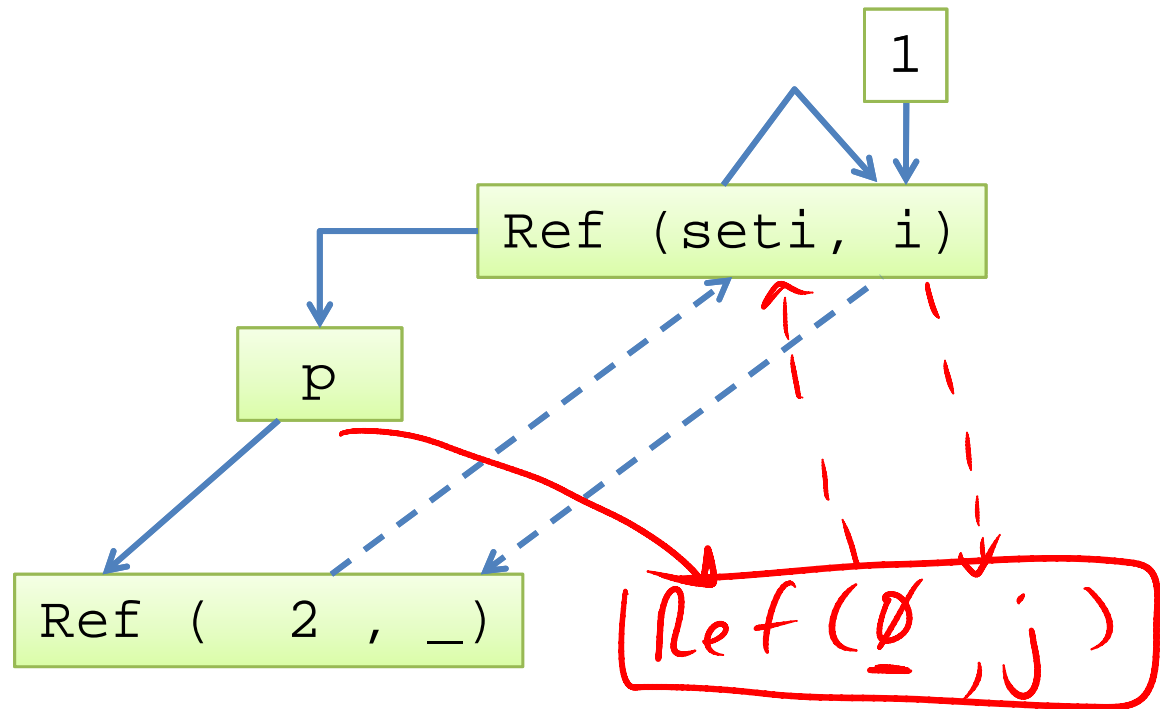
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a `Ref (-, +)` constructor
- Two args correspond to set and get operations

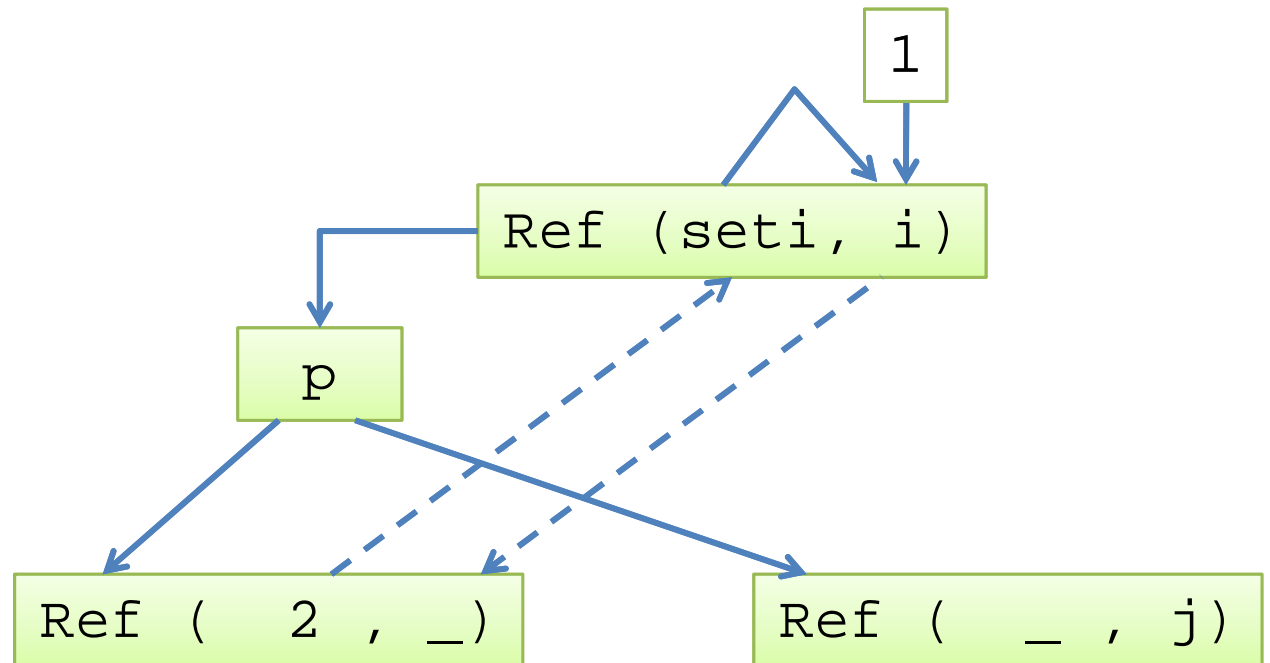
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a `Ref (-, +)` constructor
- Two args correspond to set and get operations

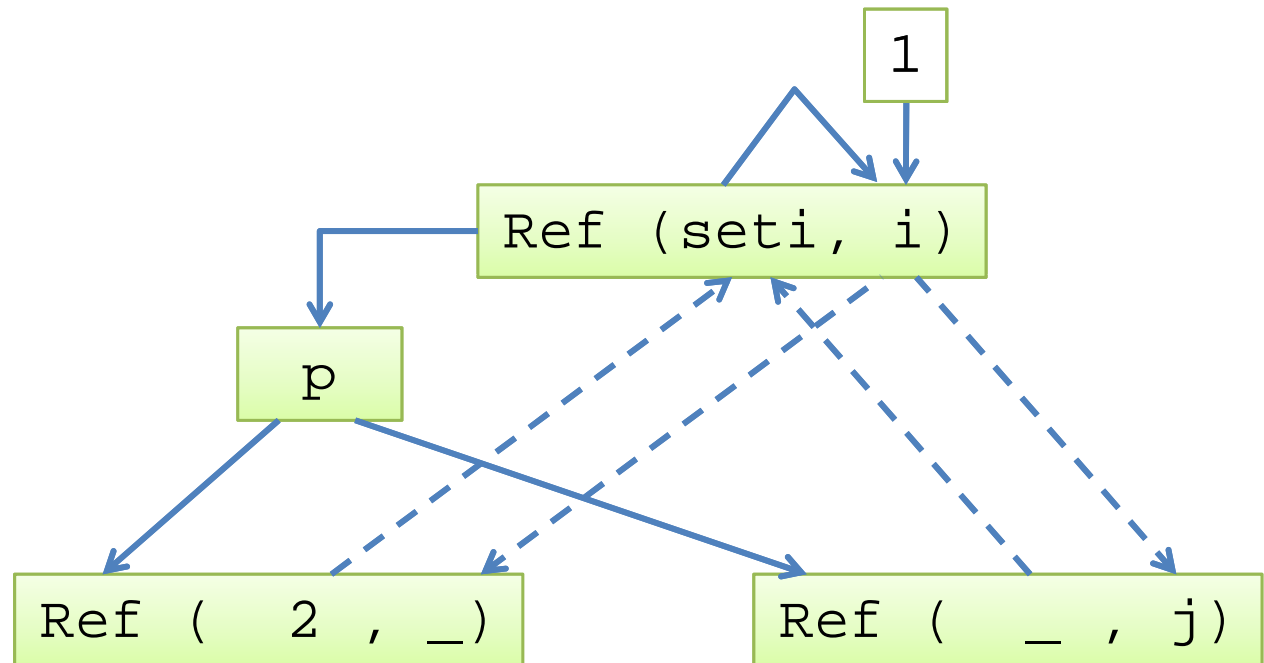
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a `Ref (-, +)` constructor
- Two args correspond to set and get operations

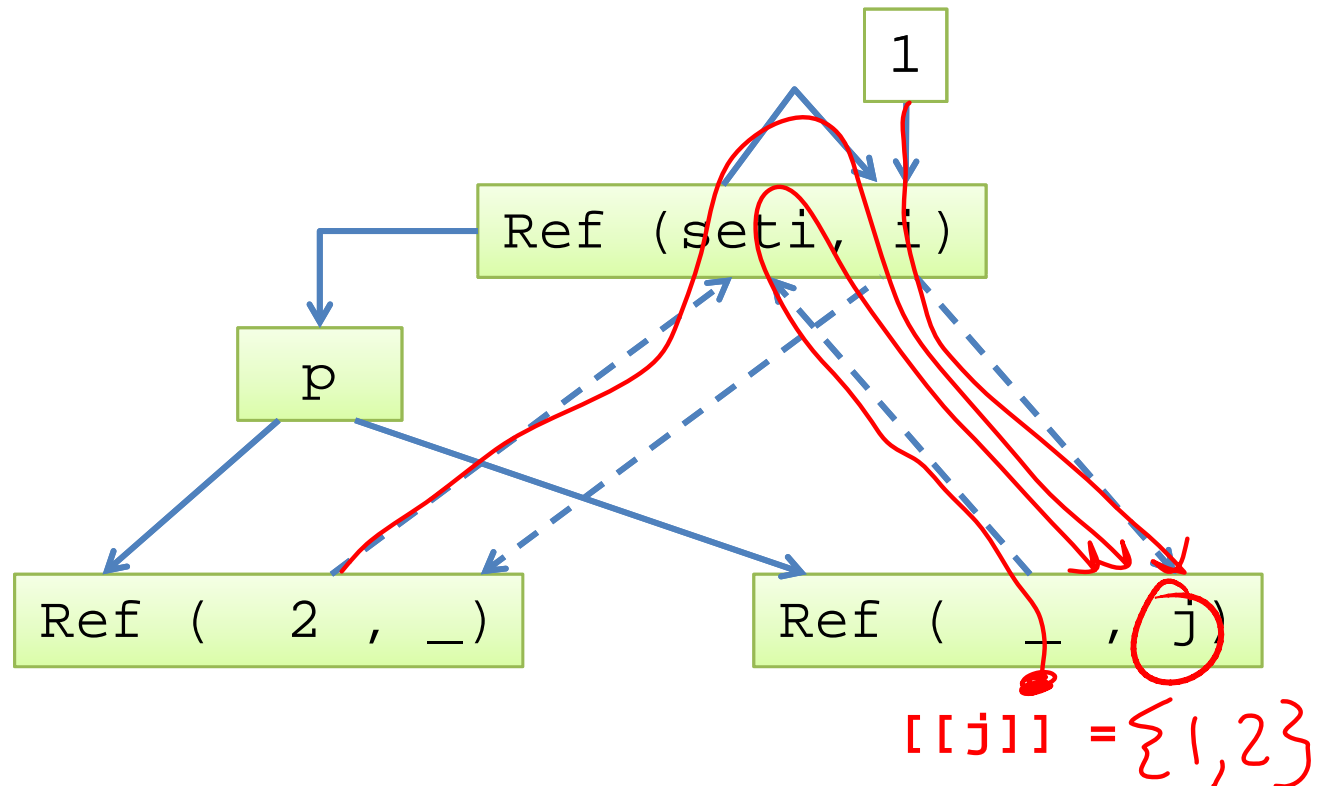
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a `Ref (-, +)` constructor
- Two args correspond to set and get operations

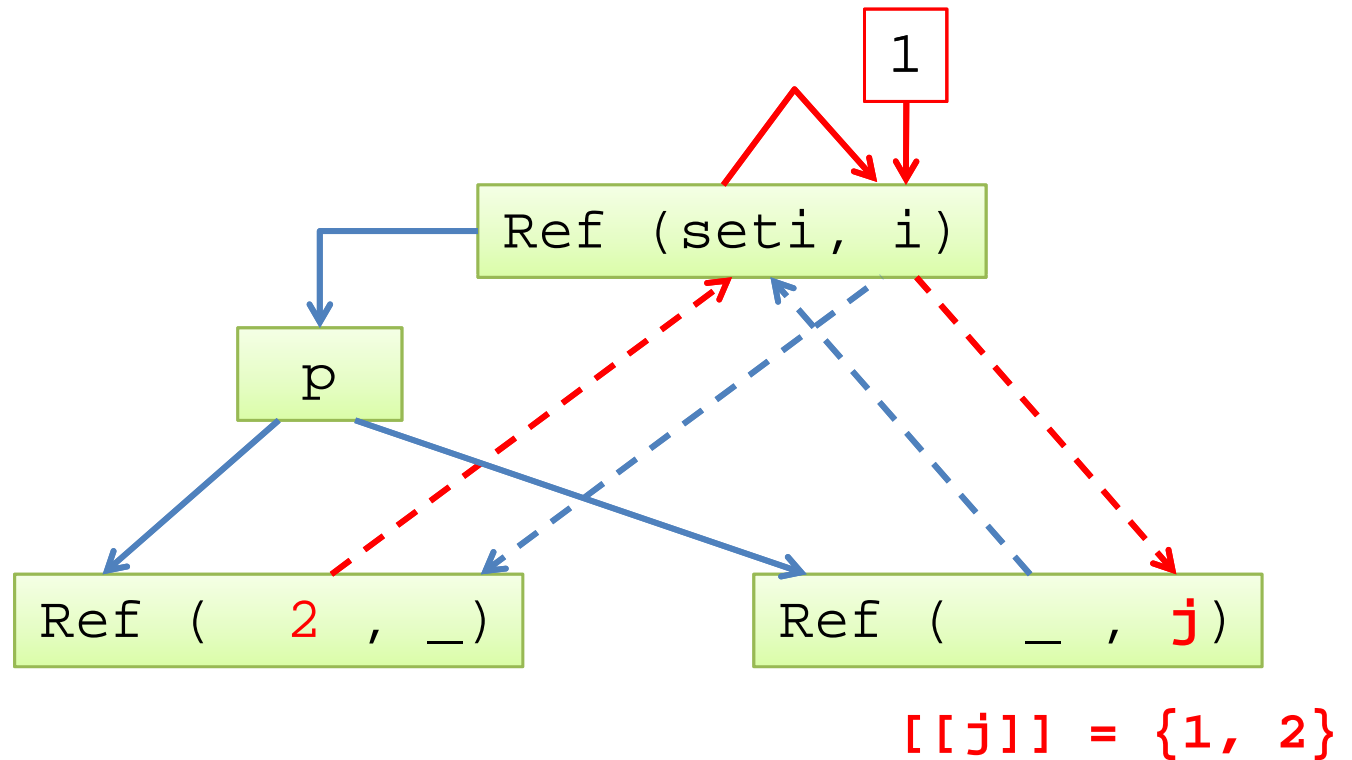
```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



Pointers

- Handle pointers with a `Ref (-, +)` constructor
- Two args correspond to set and get operations

```
int i = 1;  
int *p = &i;  
*p = 2;  
int j = *p;
```



More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

```
→ int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```

More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

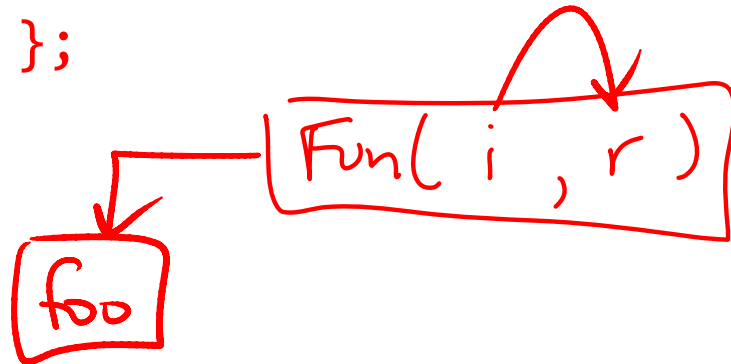
```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```

funcPtr

More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```

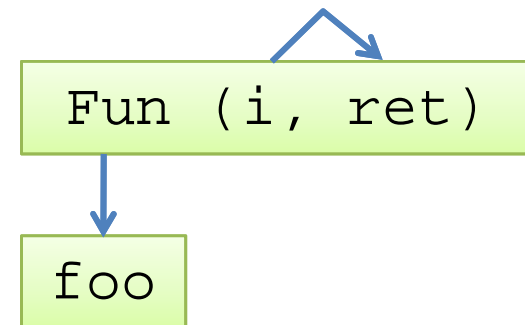


funcPtr

More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```

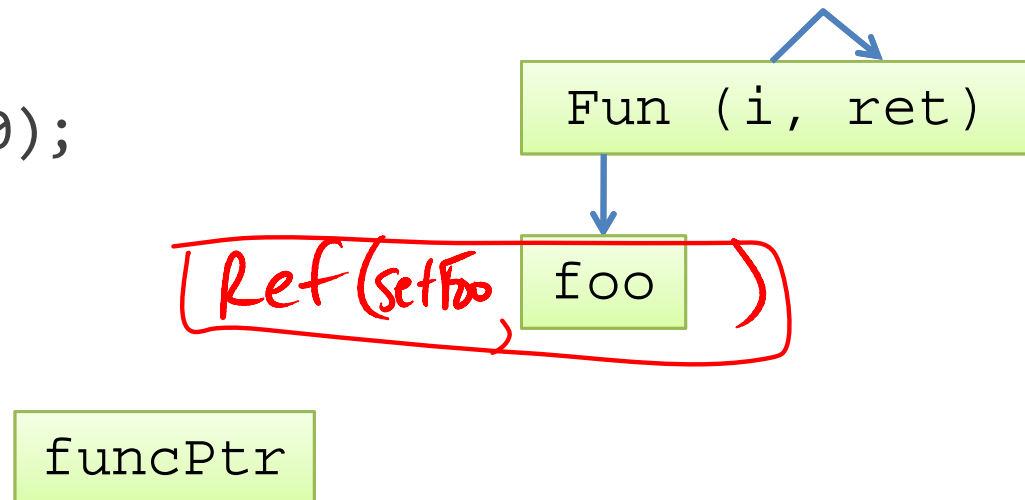


funcPtr

More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

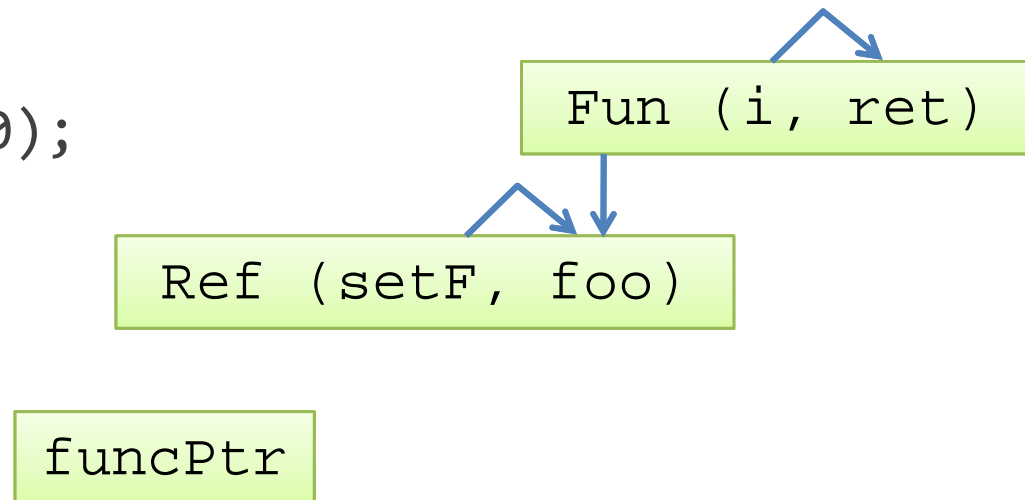
```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```



More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

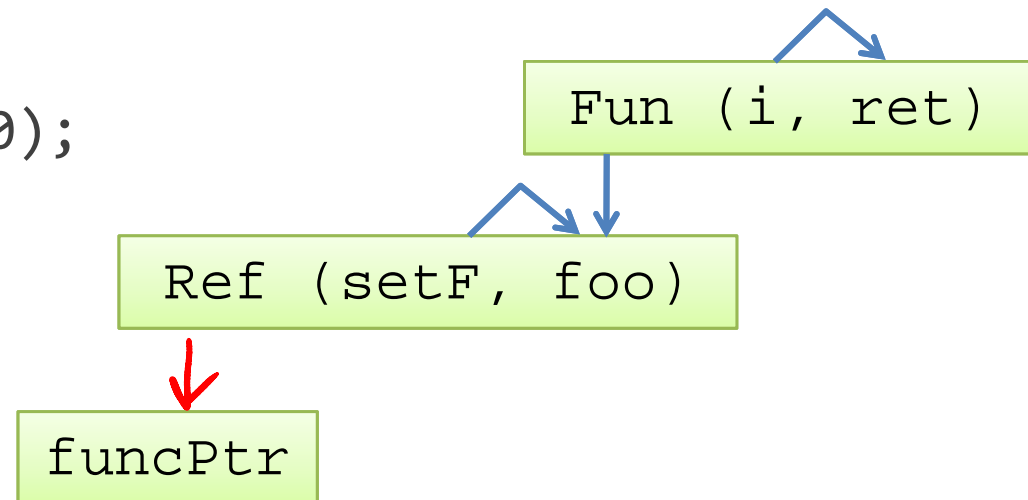
```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```



More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

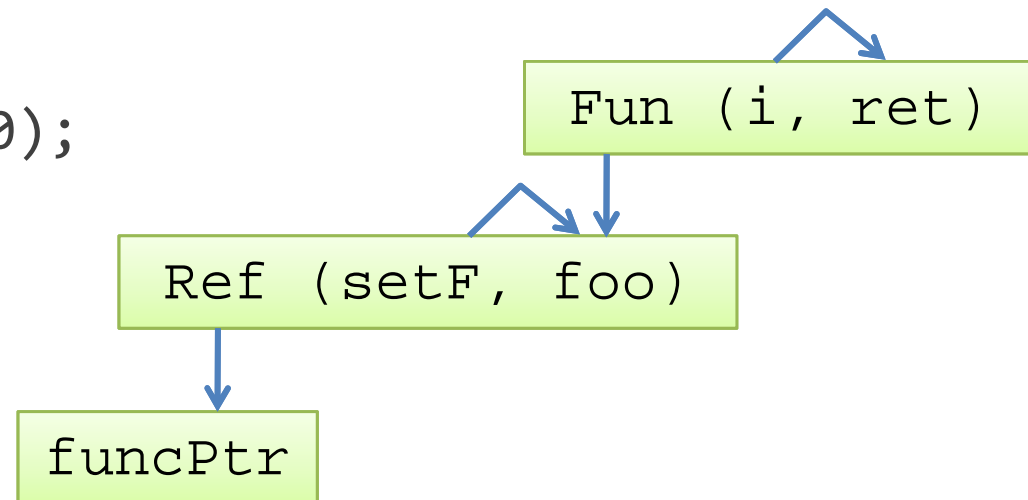
```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```



More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

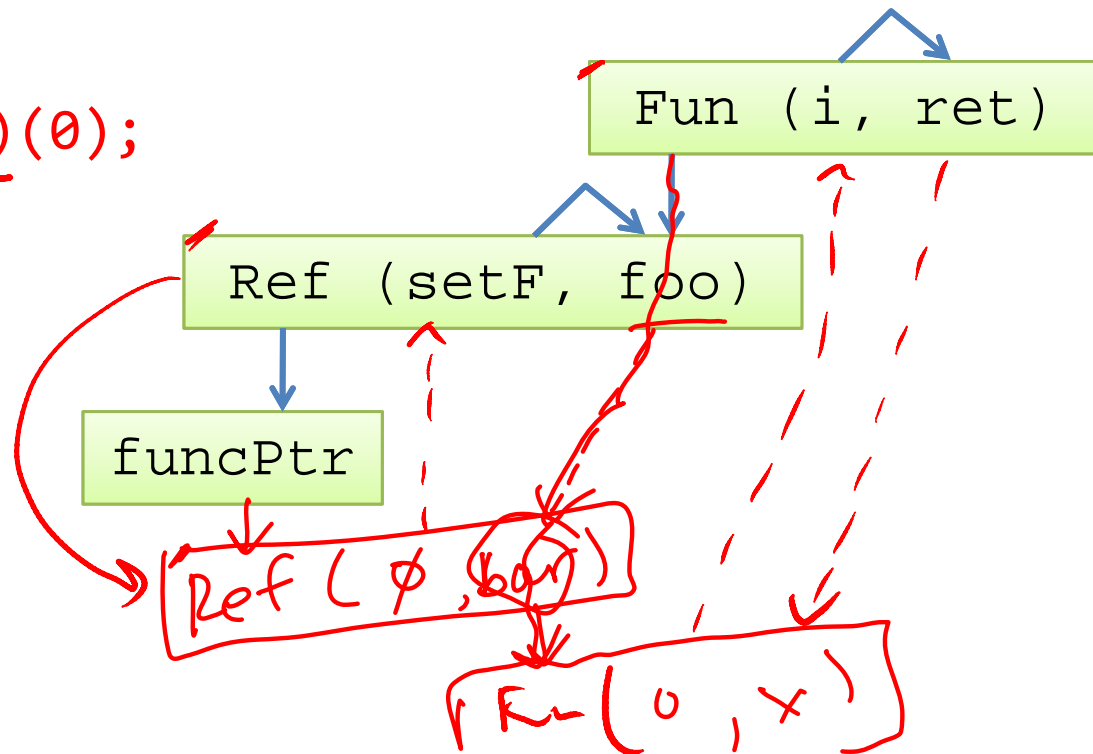
```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```



More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

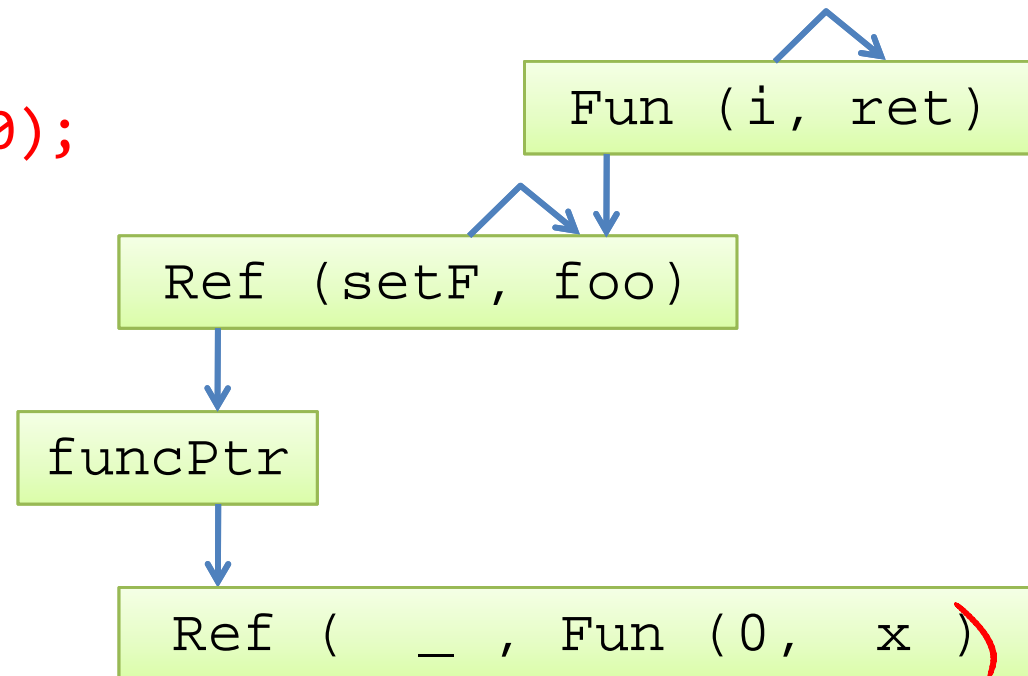
```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```



More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

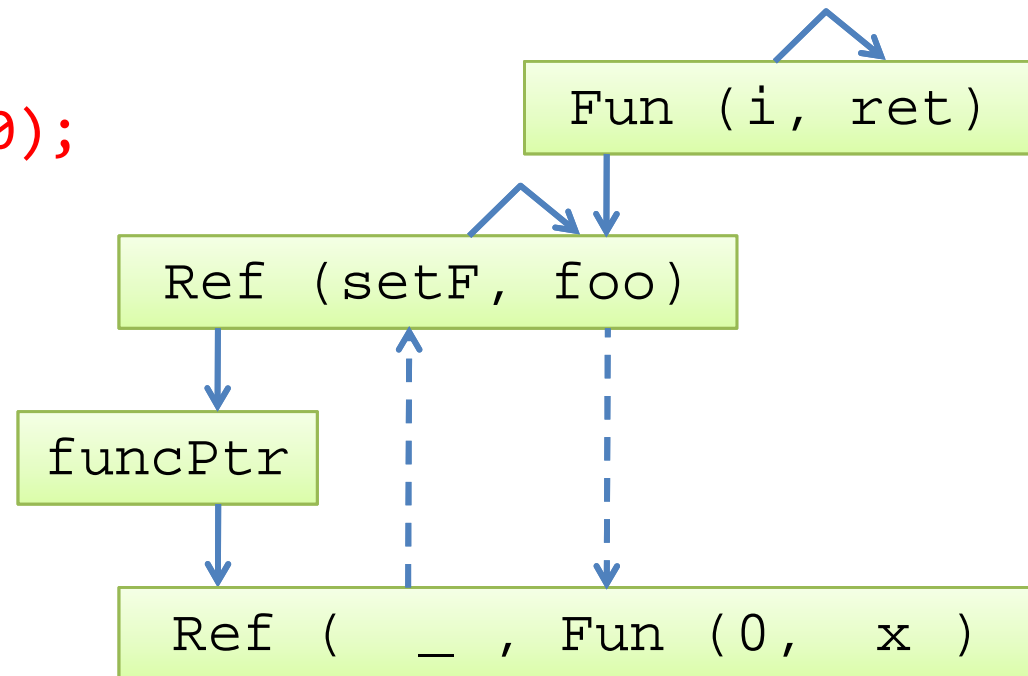
```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```



More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

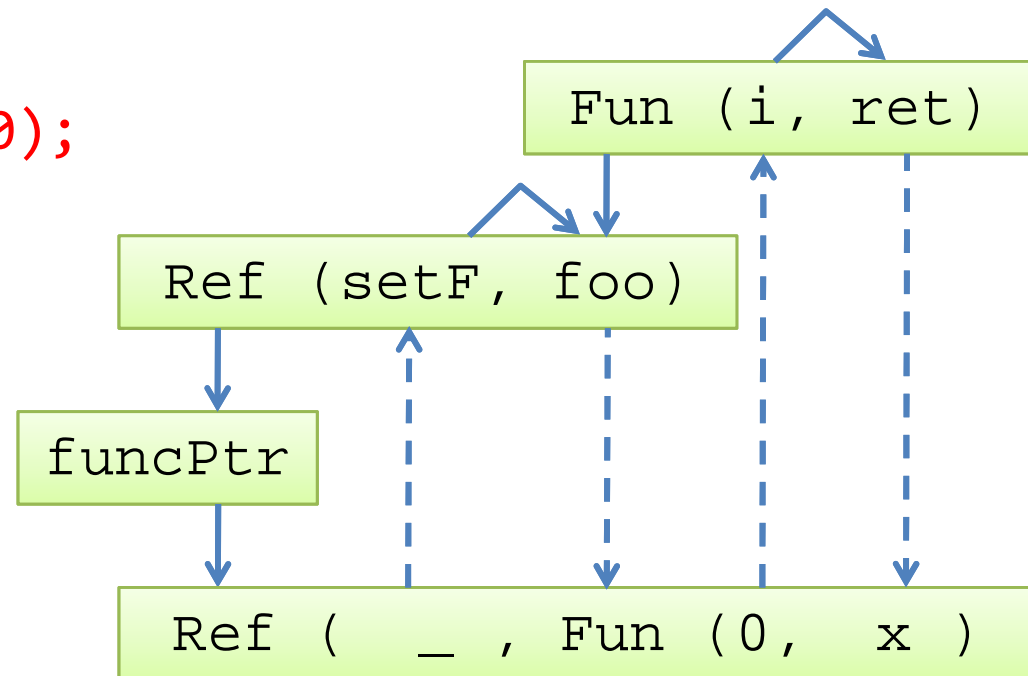
```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```



More on functions

- Our encoding supports higher-order functions
 - Passing around `Fun` terms just like constants
- Function pointers also just work

```
int (*funcPtr)(int);  
int foo(int i) { return i };  
funcPtr = &foo;  
int x = (*funcPtr)(0);
```



Context sensitivity

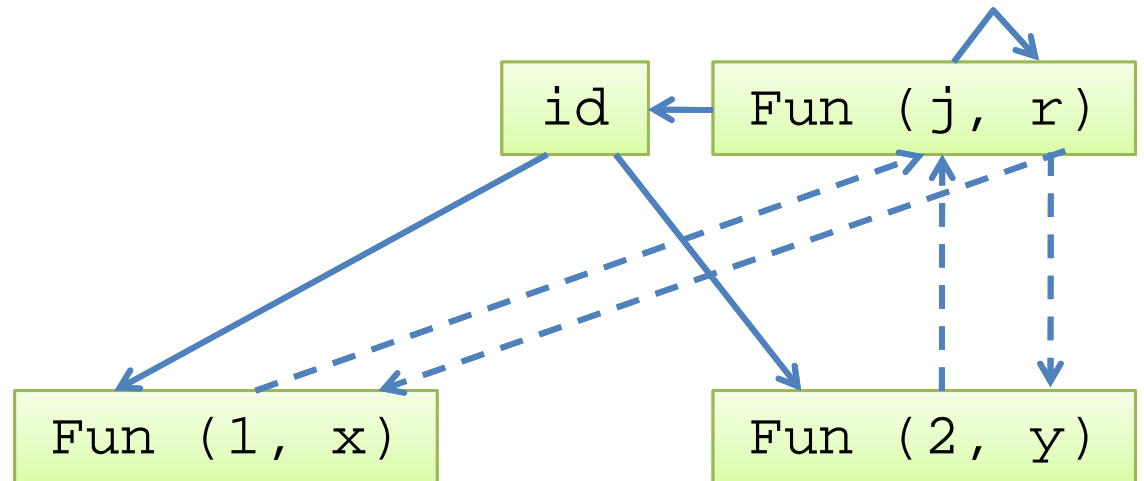
- Smearing call sites

```
int x = id(1);
```

```
int y = id(2);
```

```
[[x]] = {1, 2}
```

```
[[y]] = {1, 2}
```

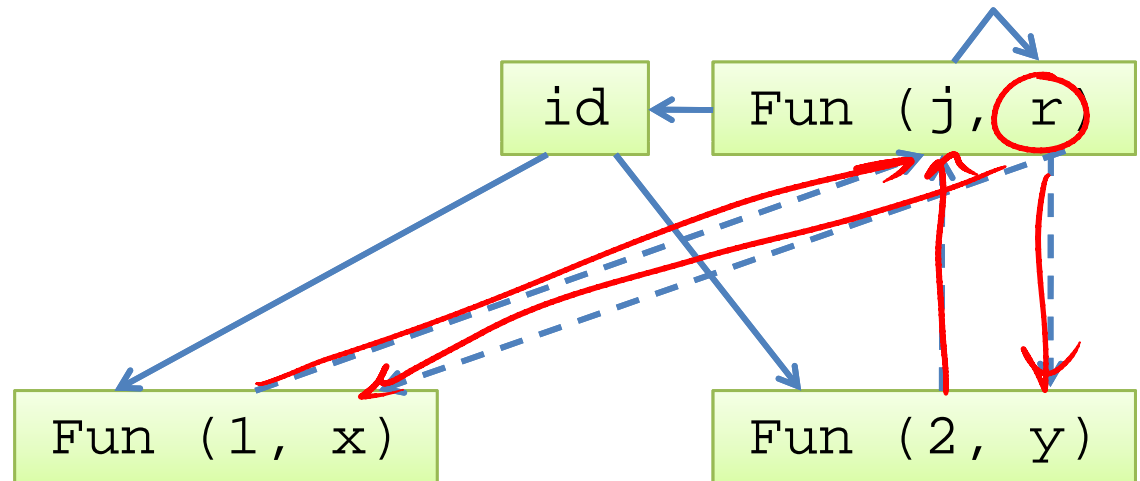


Context sensitivity

- Smearing call sites

```
int x = id(1);  
int y = id(2);
```

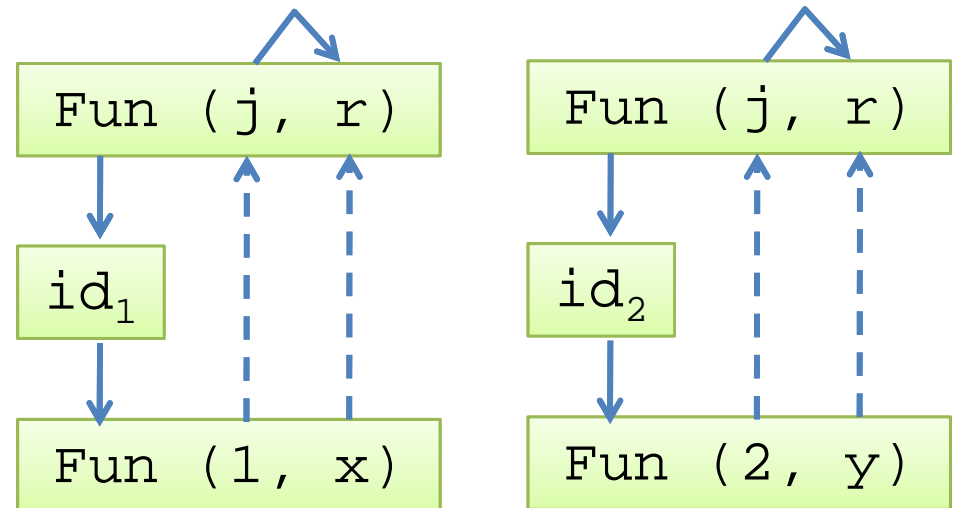
```
[[x]] = {1, 2}  
[[y]] = {1, 2}
```



- Option 1:

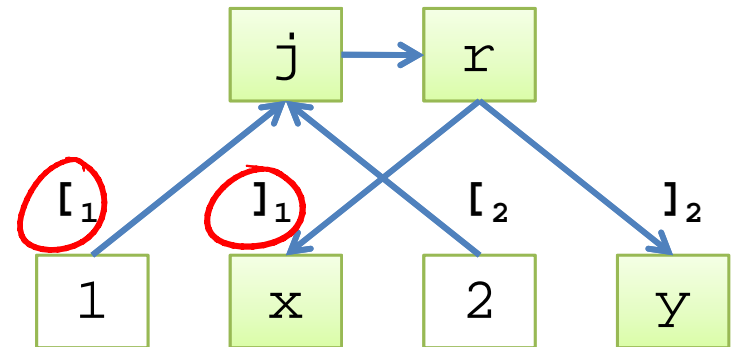
Specialization

- Each call id_i gets a new copy of id
- Eliminates smearing, but graph size increases



Context sensitivity

- Option 2: Unique labeled edges for each call site
- Not using Fun constructor

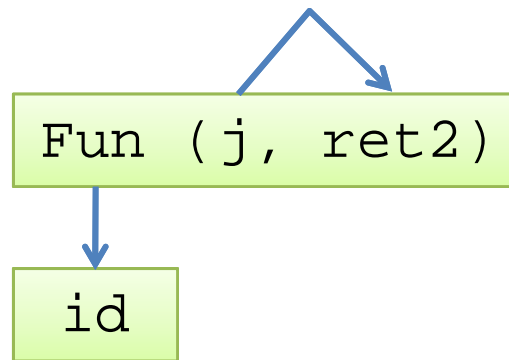


- There is flow only if there is a path that spells a substring of a well-bracketed string
 - $[_a[_b]_b]_a$ and $[_a]_a[_b]$ are valid; $[_a[_b]_a]_b$ is not
- For both options, if there are higher-order functions or function pointers, need a first pass to compute pointer targets

Field sensitivity

- For each field f , define $\text{Fld}_{\underline{f}}(-, +)$ constructor

```
int readG(obj p) {  
    return p.g;  
}  
obj o;  
o.f = 3;  
o.g = 4;  
int w = id(o.f);  
int z = readG(o);
```

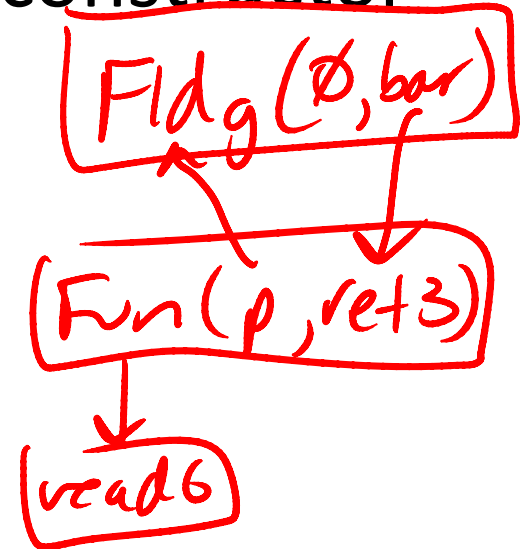
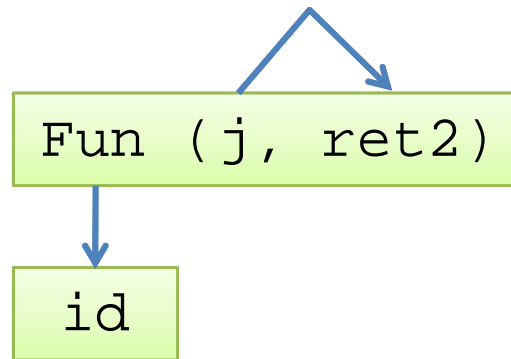


Field sensitivity

- For each field f , define $\text{Fld}_f(-, +)$ constructor

```
Struct obj { f g }  
int readG(obj p) {  
    return p.g;  
}
```

```
obj o;  
o.f = 3;  
o.g = 4;  
int w = id(o.f);  
int z = readG(o);
```

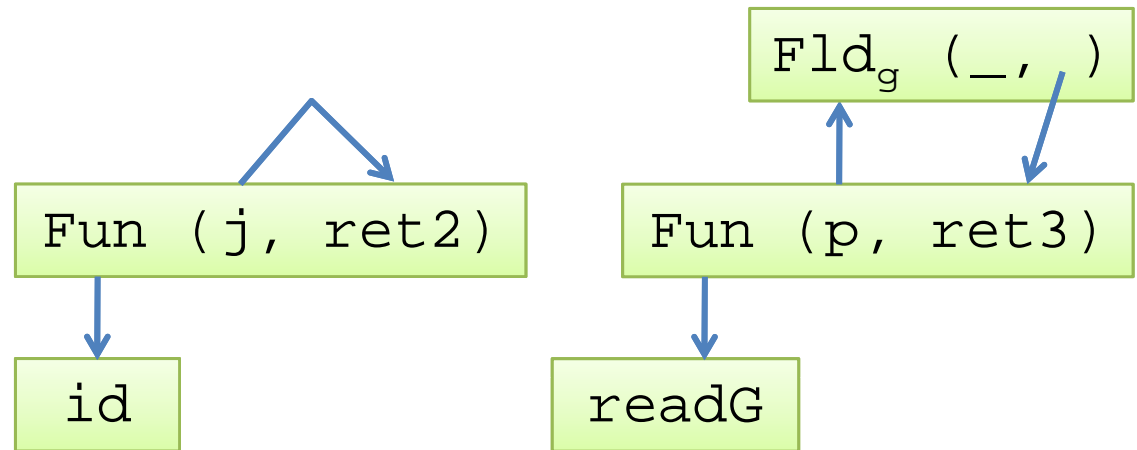


Field sensitivity

- For each field f , define $\text{Fld}_f(-, +)$ constructor

```
int readG(obj p) {  
    return p.g;  
}
```

```
obj o;  
o.f = 3;  
o.g = 4;  
int w = id(o.f);  
int z = readG(o);
```



Field sensitivity

- For each field f , define $\text{Fld}_f(-, +)$ constructor

```
int readG(obj p) {  
    return p.g;  
}
```

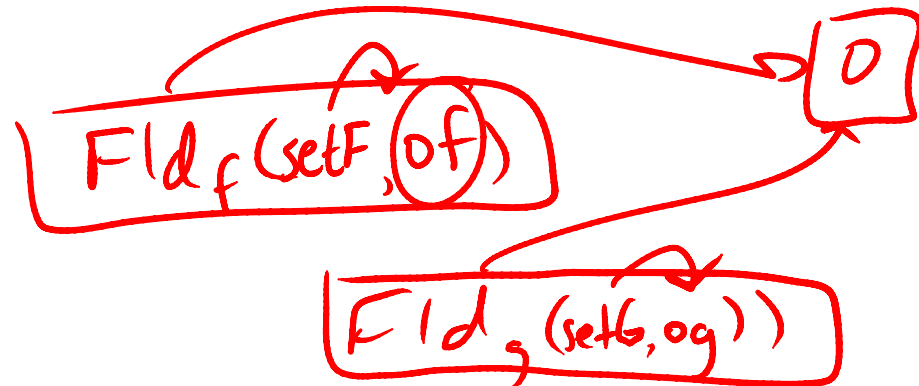
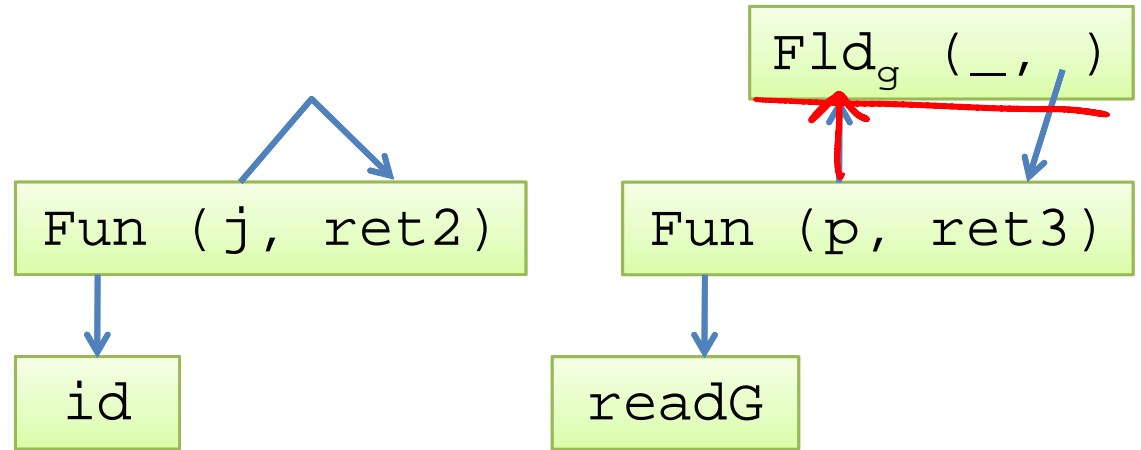
obj o;

o.f = 3;

o.g = 4;

int w = id(o.f);

int z = readG(o);



Field sensitivity

- For each field f , define $\text{Fld}_f(-, +)$ constructor

```
int readG(obj p) {  
    return p.g;  
}
```

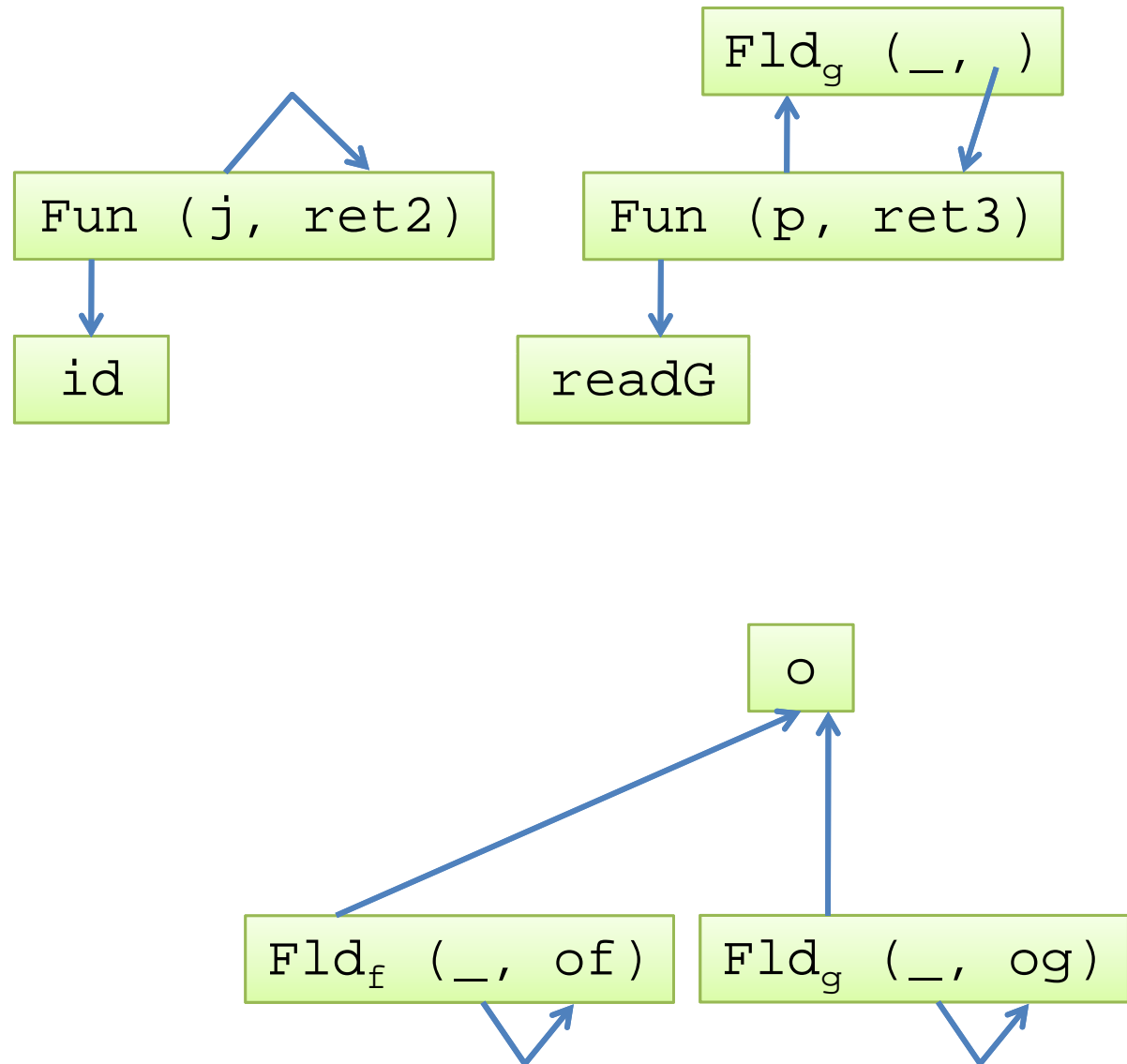
```
obj o;
```

```
o.f = 3;
```

```
o.g = 4;
```

```
int w = id(o.f);
```

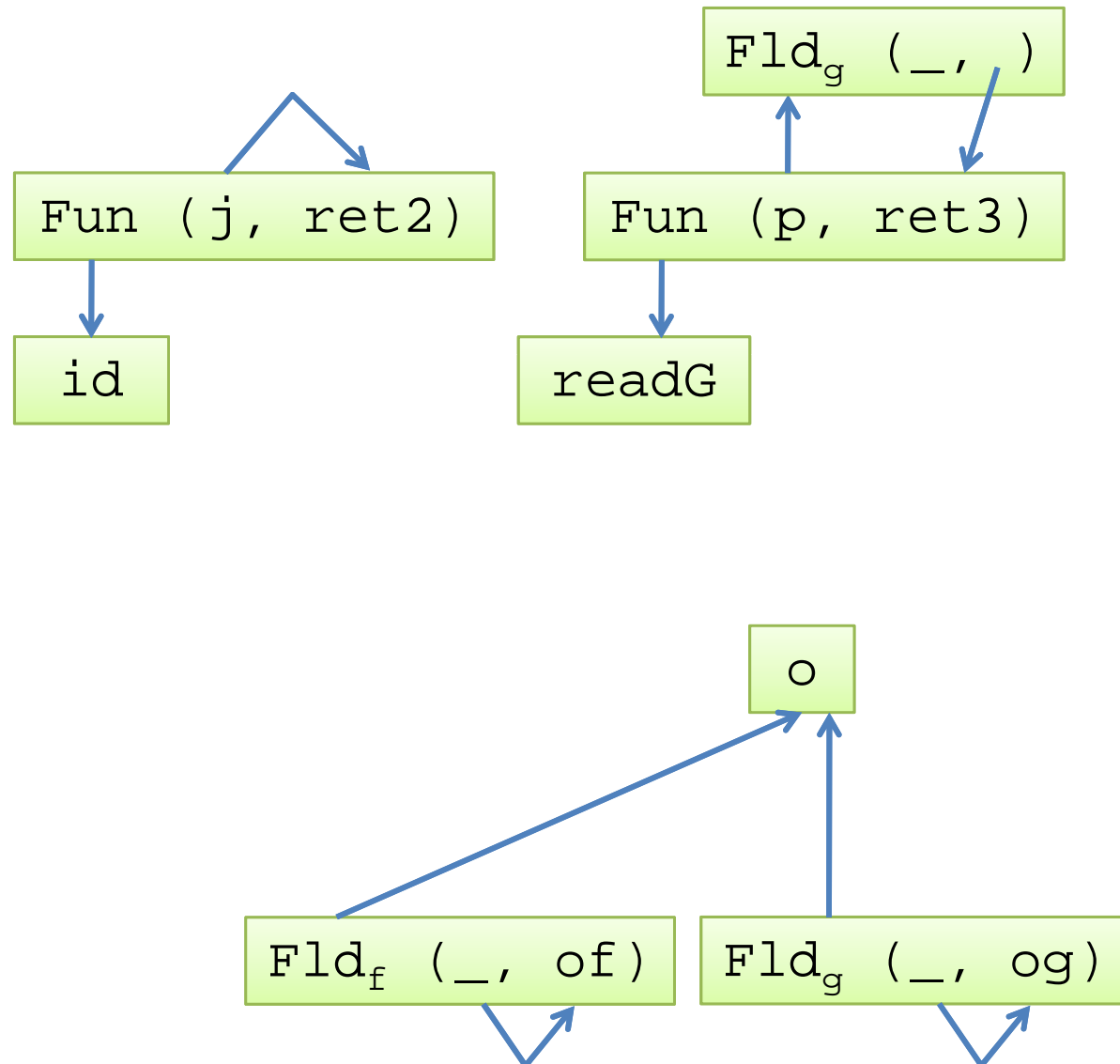
```
int z = readG(o);
```



Field sensitivity

- For each field f , define $\text{Fld}_f(-, +)$ constructor

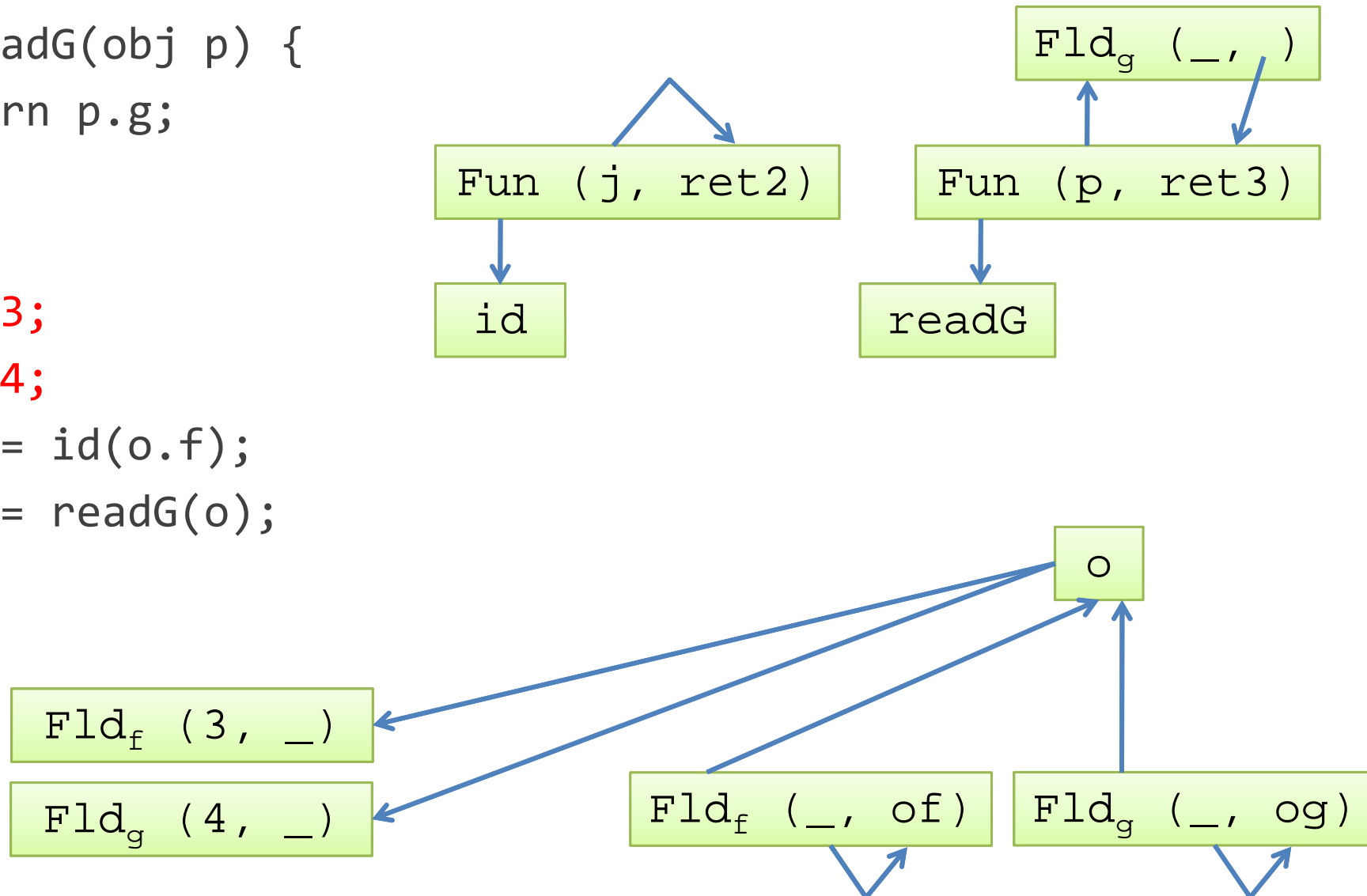
```
int readG(obj p) {  
    return p.g;  
}  
obj o;  
o.f = 3;  
o.g = 4;  
int w = id(o.f);  
int z = readG(o);
```



Field sensitivity

- For each field f , define $\text{Fld}_f(-, +)$ constructor

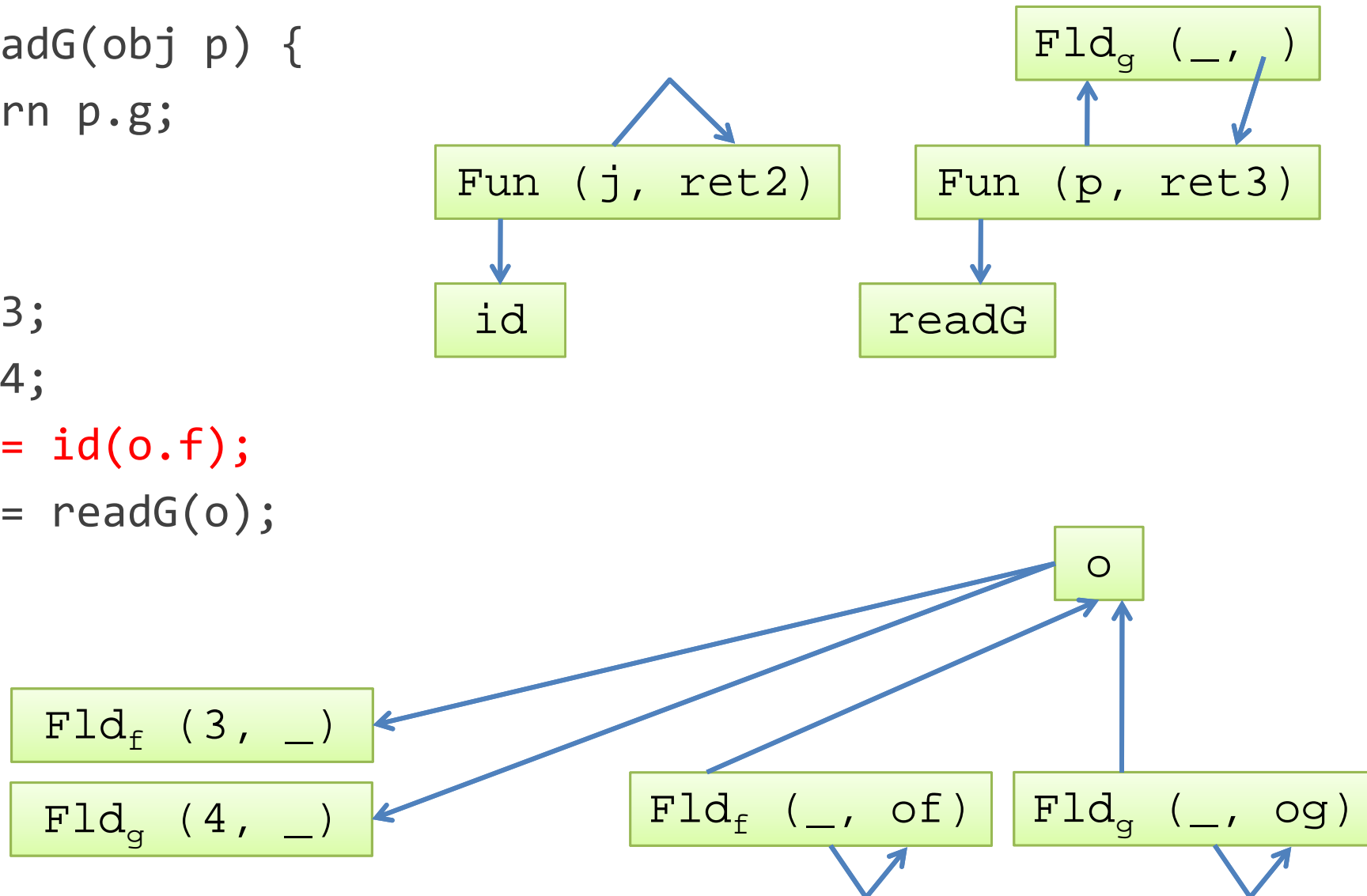
```
int readG(obj p) {  
    return p.g;  
}  
obj o;  
o.f = 3;  
o.g = 4;  
int w = id(o.f);  
int z = readG(o);
```



Field sensitivity

- For each field f , define $\text{Fld}_f(-, +)$ constructor

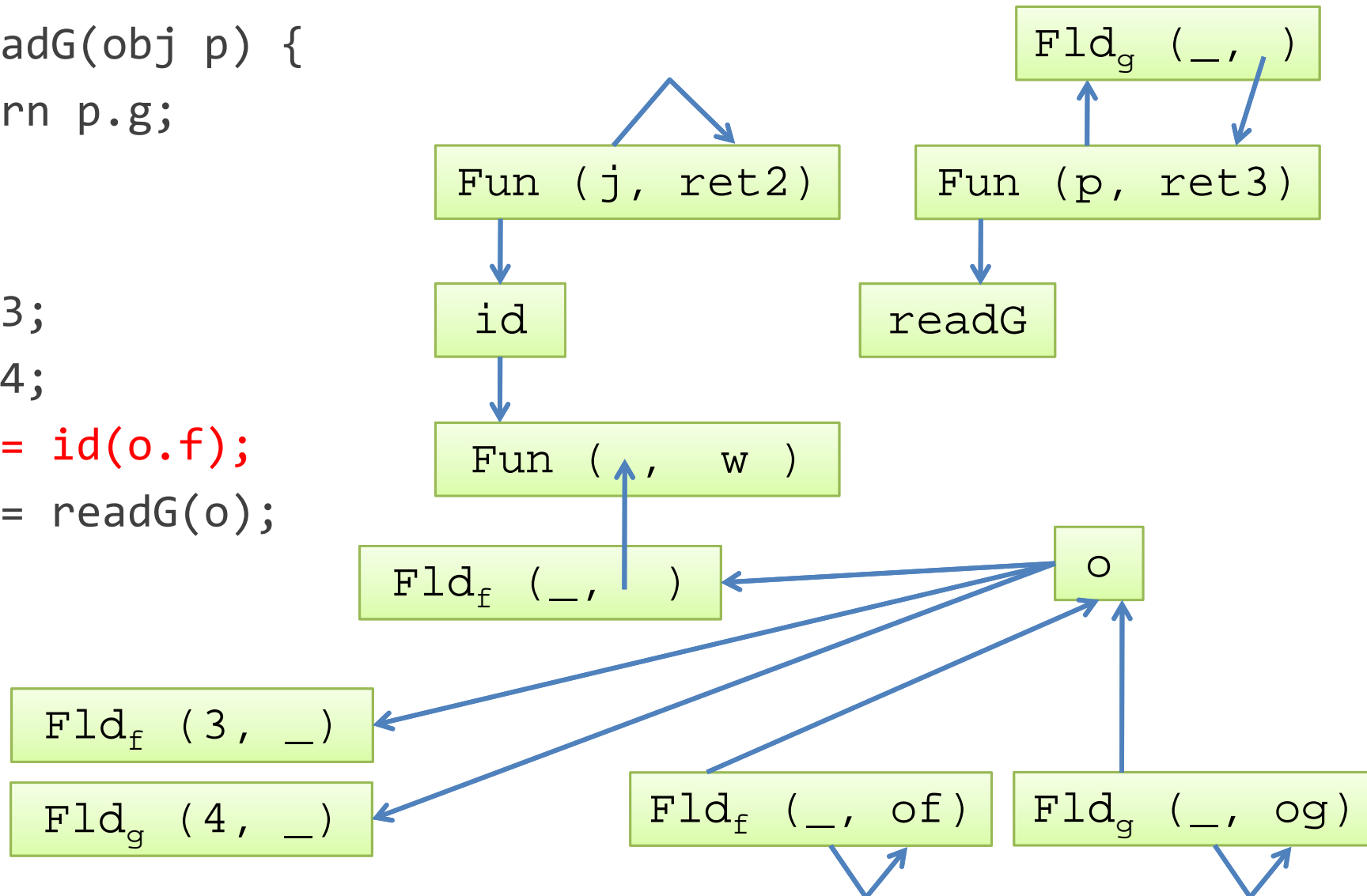
```
int readG(obj p) {  
    return p.g;  
}  
obj o;  
o.f = 3;  
o.g = 4;  
int w = id(o.f);  
int z = readG(o);
```



Field sensitivity

- For each field f , define $\text{Fld}_f(-, +)$ constructor

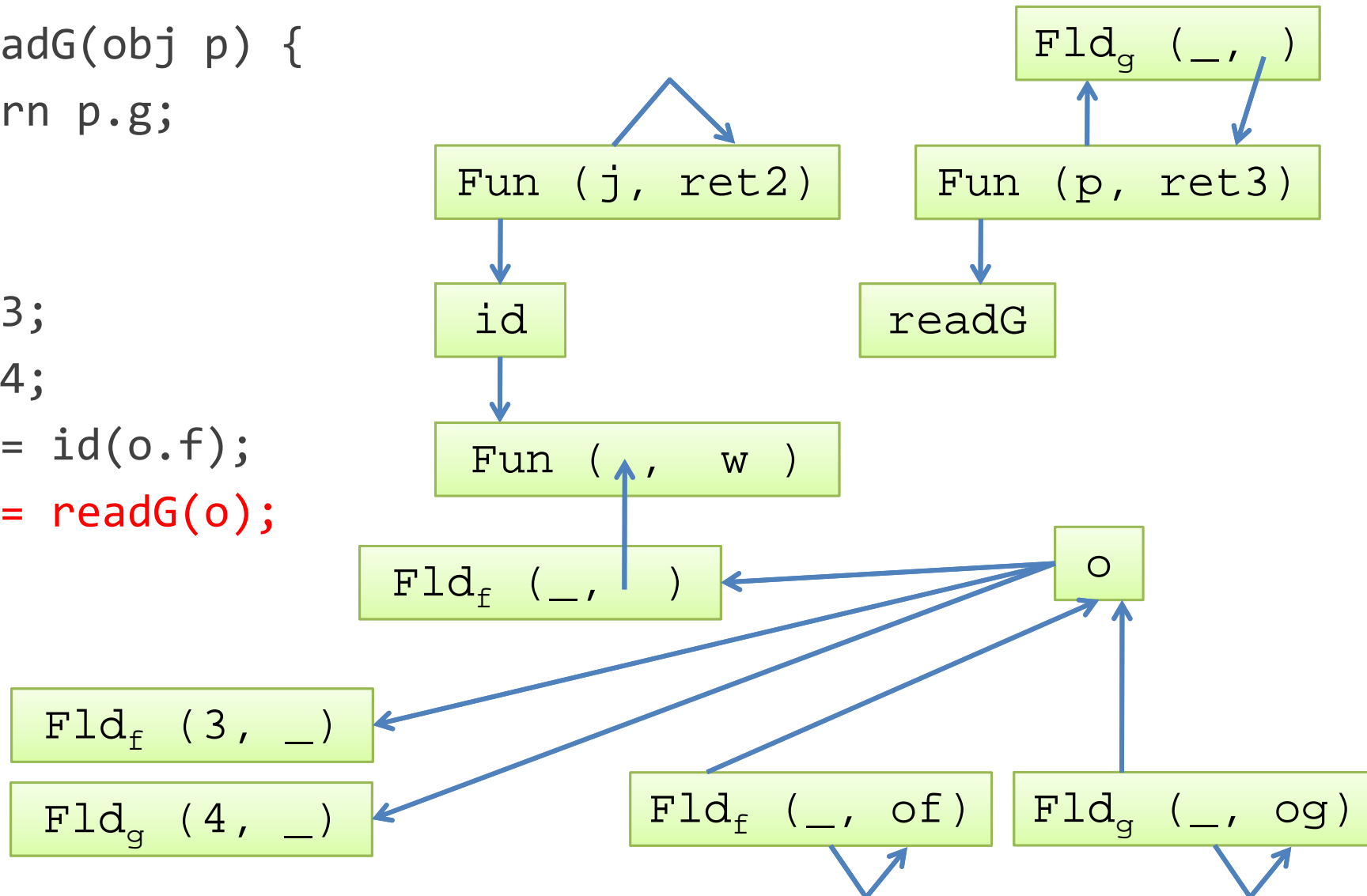
```
int readG(obj p) {  
    return p.g;  
}  
obj o;  
o.f = 3;  
o.g = 4;  
int w = id(o.f);  
int z = readG(o);
```



Field sensitivity

- For each field f , define $\text{Fld}_f(-, +)$ constructor

```
int readG(obj p) {  
    return p.g;  
}  
obj o;  
o.f = 3;  
o.g = 4;  
int w = id(o.f);  
int z = readG(o);
```



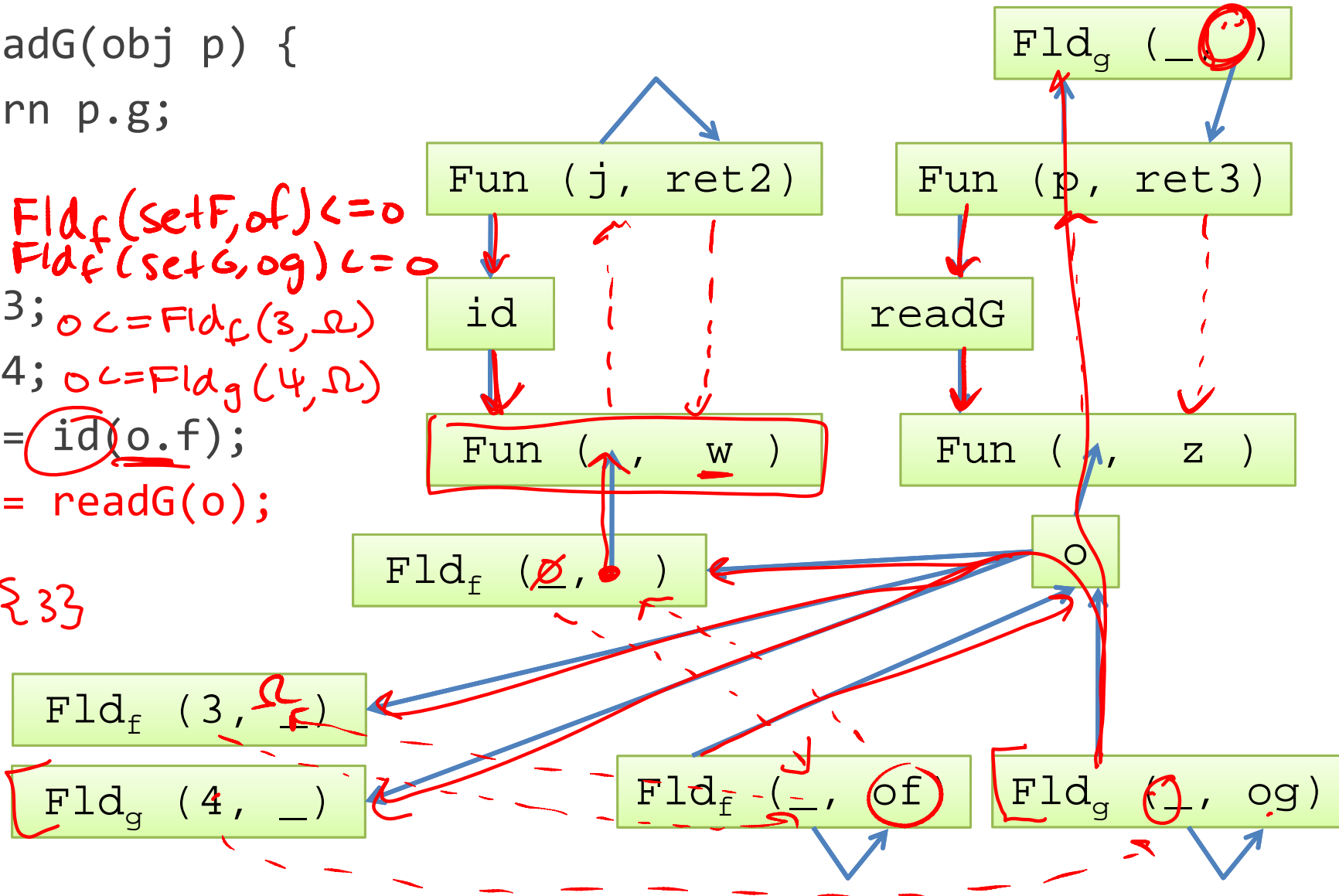
Field sensitivity

- For each field f , define $\text{Fld}_f(-, +)$ constructor

```
int readG(obj p) {
    return p.g;
}
```

```
obj o;
o.f = 3;
o.g = 4;
int w = id(o.f);
int z = readG(o);
```

$\text{Fld}_f(\text{set } F, \text{of}) \leq 0$
 $\text{Fld}_f(\text{set } G, \text{og}) \leq 0$
 $0 \leq \text{Fld}_f(3, \Omega)$
 $0 \leq \text{Fld}_g(4, \Omega)$
 $\llbracket \text{of} \rrbracket = \{3\}$



Scalability

- Constraint graph for entire program is in memory
- Even for flow-insensitive analyses, this can become a bottleneck
- Even worse for flow-sensitive analyses
- Techniques for analyzing parts of program in isolation and storing summaries of their observable effects

Summary

- Set constraints are often natural for expressing various program analyses
 - Constant propagation, pointer analysis
 - Closure analysis
 - Receiver class analysis
 - * Information flow
- Rich literature on solving systems of constraints
- Non-trivial to extend to flow-sensitive or summary-based analyses
- Interference between functions and references