Alias Analysis and Load/Store Elimination

15-411/15-611 Compiler Design

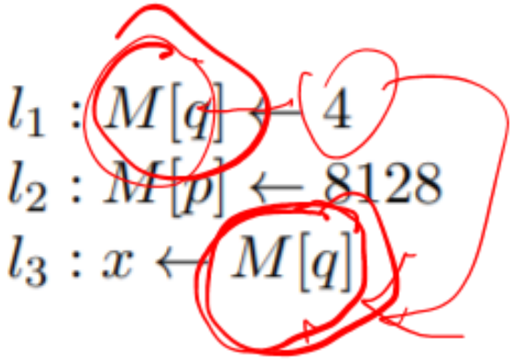
Ben L. Titzer and Seth Copen Goldstein

March 26, 2026

Today

- Alias analysis
- Load elimination
- Load-store forwarding
- Store elimination
- Lame alias analysis with SSA

Constant Propagation on Memory



- Can we replace l_3 with **x ← 4**?

Constant Propagation on Memory

$$l_1 : M[q] \leftarrow 4$$
$$l_2 : M[p] \leftarrow 8128$$
$$l_3 : x \leftarrow M[q]$$

- Can we replace l_3 with **x ← 4**?
- Only if p ≠ q
- That is, only if p and q do not alias

Alias Analysis

- Determines if two memory accesses are
  - Definitely different
  - Definitely the same
  - Maybe the same
- So far, optimizations (e.g., const prop, copy prop, …) work on temps/registers
- SSA exposes explicit dataflow
- Memory/Pointers are less explicit dataflow
- Alias analysis helps uncover dataflow between memory references

Alias Analysis Unlocks Optimizations

```
*a = 5
*b = 7
t = *a;
```

What if I know a ≠ b?

Available Expressions

Stmt s	Gen	Kill
t <- x op y	{x op y}-kill[s]	{exprs containing t}
t <- M[a]	{M[a]}-kill[s]	{exprs containing t}
M[a] <- b	{}	{for all x, M[x]}
f(a, …)	{}	{for all x, M[x]}
t <- f(a,…)	{}	{exprs containing t for all x, M[x]}

Alias Analysis Unlocks Optimizations

```
*a = 5
*b = 7
t = *a;
```

What if I know a ≠ b?

Available Expressions

Stmt s	Gen	Kill
t <- x op y	{x op y}-kill[s]	{exprs containing t}
t <- M[a]	{M[a]}-kill[s]	{exprs containing t}
M[a] <- b	{}	{for all x, M[x]}
f(a, …)	{}	{for all x, M[x]}
t <- f(a,…)	{}	{exprs containing t for all x, M[x]}

```
foo() {
        int a,k;
        extern int *q;
        ...
        // maybe &a or &k
        ...
        k = a+6;
        f(a,&k);
        *q = 13;
        k = a+6
        ...
}
```

CSE on Memory

```
struct point {
  int x;
  int y;
};
typedef struct point pt;

void mult(int[] A, pt* p, pt* q) {
  q->x = A[0] * p->x + A[1] * p->y;
  q->y = A[2] * p->x + A[3] * p->y;
  return;
}
```

mult(A, p, q) :

$t_0 \leftarrow M[A + 0]$
$t_1 \leftarrow M[p + 0]$
$t_2 \leftarrow t_0 * t_1$
$t_3 \leftarrow M[A + 4]$
$t_4 \leftarrow M[p + 4]$
$t_5 \leftarrow t_3 * t_4$
$t_6 \leftarrow t_2 + t_5$
$M[q + 0] \leftarrow t_6$
$t_8 \leftarrow M[A + 8]$
$t_9 \leftarrow M[p + 0]$
$t_{10} \leftarrow t_8 * t_9$
$t_{11} \leftarrow M[A + 12]$
$t_{12} \leftarrow M[p + 4]$
$t_{13} \leftarrow t_{11} * t_{12}$
$t_{14} \leftarrow t_{10} + t_{13}$
$M[q + 4] \leftarrow t_{14}$
return

Alias Analysis Unlocks Optimizations

- Load/Store Elimination
- Load/Store Forwarding
- Code Motion
- Constant Propagation through memory
- Etc.

Source of Aliases

- Pass by reference parameters
- Address of operator
- Address arithmetic
- Dereferencing pointers
- Array subscripting
- Non-local variables
- Assignment

Aliasing Possibilities

- Pascal
  - Only has pointers to objects and arrays
  - Objects and their fields are statically typed
  - No pointers to locals
  - Call-by-value and call by reference
  - Nested procedures

Aliasing Possibilities

- C
  - unions
  - pointers to structs and arrays
  - pointers to locals
  - pointers to fields
  - pointers to array elements
  - pointer arithmetic
  - type punning

Aliasing Possibilities

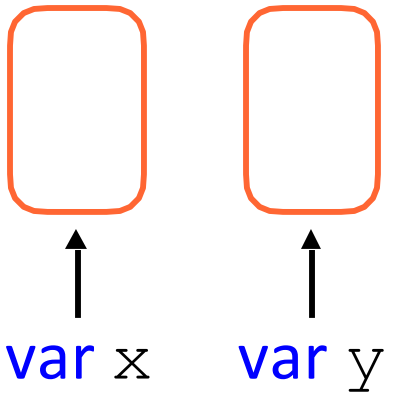
- C0
  - unions
  - pointers to structs and arrays
  - pointers to locals
  - pointers to fields
  - pointers to array elements
  - pointer arithmetic
  - type punning

Aliasing Possibilities

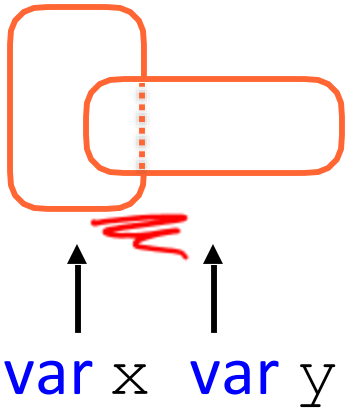
- C0
  - unions
  - pointers to structs and arrays
  - pointers to locals
  - pointers to fields
  - pointers to array elements
  - pointer arithmetic
  - type punning

Aliasing Relations

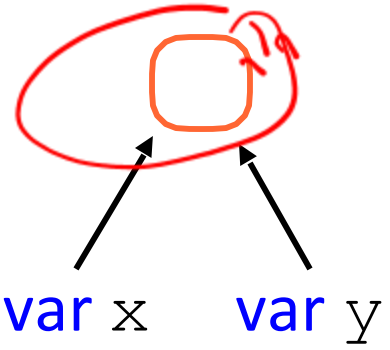
- Primarily interested in:
  - For any two pointers in the program, what set of objects could they point to?



no overlap
"must-not-alias"



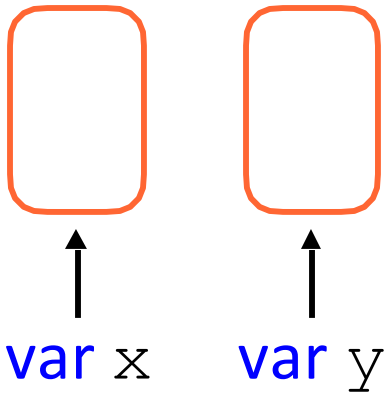
overlap
"may-alias"



same object
"must-alias"

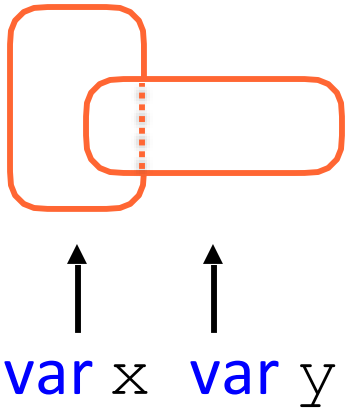
Aliasing Relations

- Primarily interested in:
  - For any two pointers in the program, what set of objects could they point to?



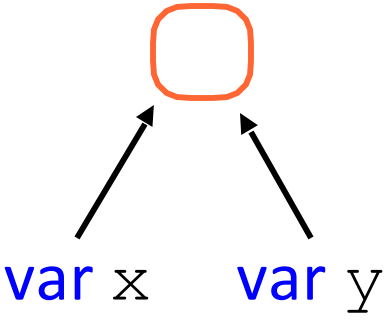
no overlap
"must-not-alias"

reorder at will



overlap
"may-alias"

treat conservatively

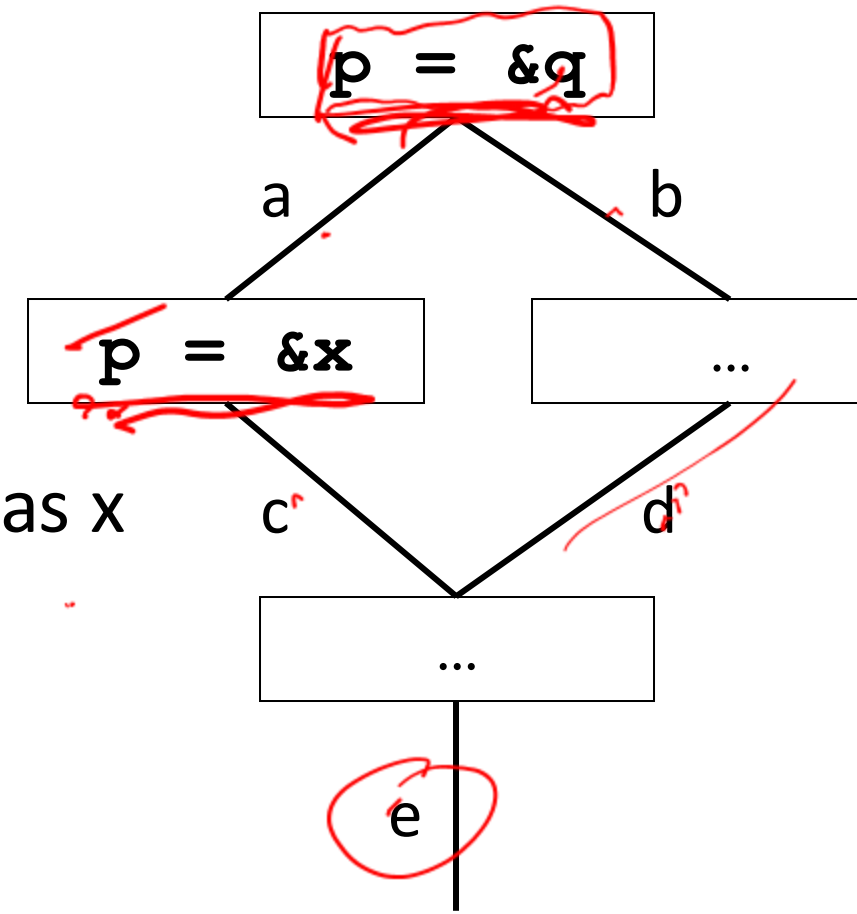


same object
"must-alias"

cache reads
propagate writes

Alias Information

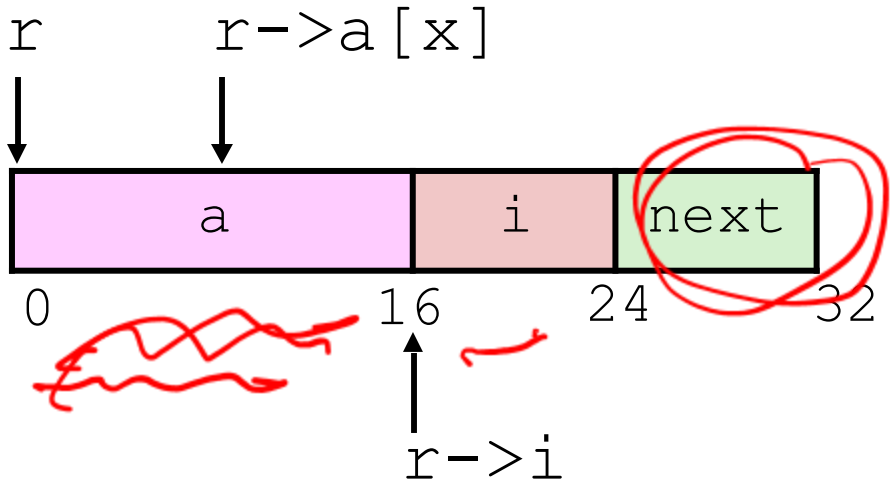
- must-alias and may-alias
  - a: p must-alias q
  - b: p must-alias q
  - c: p must-alias x
  - d: p must-alias q
  - e: p may-alias q and p may-alias x
- Analysis:
  - flow-insensitive
  - flow-sensitive



Type-based Alias Analysis

- Types severely restrict aliasing in C0, Java
- Preserve enough type information so alias analysis can distinguish types of pointer variables and field accesses.

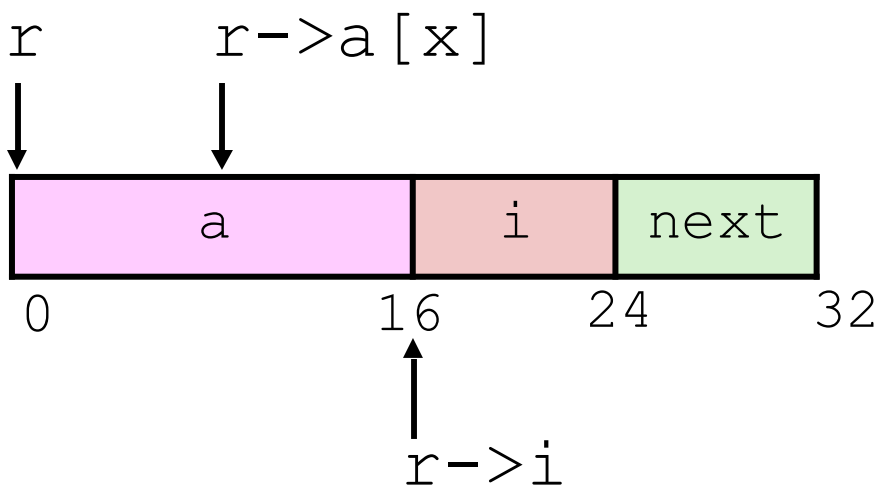
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
} *r;
```



Type-based Alias Analysis

- Types severely restrict aliasing in C0, Java
- Preserve enough type information so alias analysis can distinguish types of pointer variables and field accesses.

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
} *r;
```

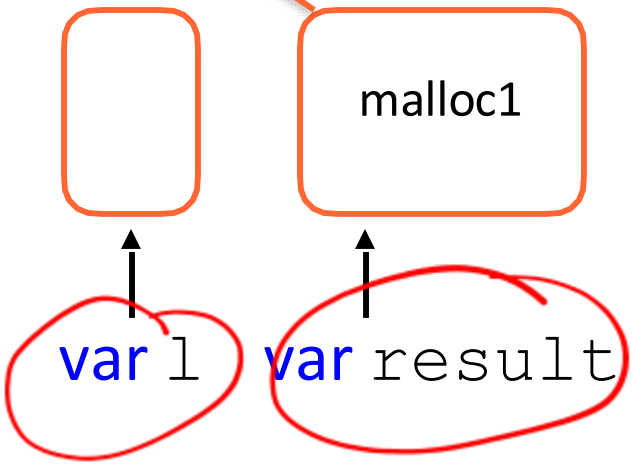


Most modern compilers make at least some use of types in alias analysis.

Allocation Sites

- Allocation sites can distinguish new pointers from old pointers.

```
List* add(List* l, int a) {
  List* result = malloc(…);
  result->next = l;
  result->val = a;
  return result;
}
```

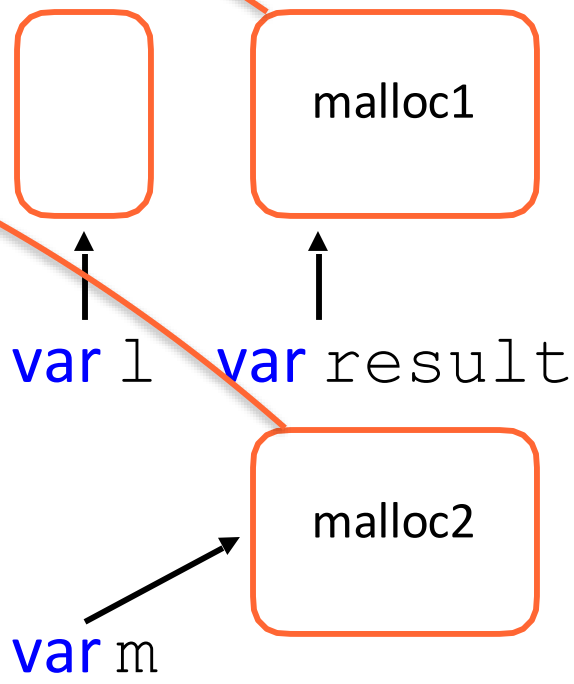


Most modern compilers make use of allocation sites for alias analysis.

Allocation Sites

- Allocation sites can distinguish new pointers from other new pointers.

```
List* add2(List* l, int a) {
  List* result = malloc(…);
  result->next = l;
  result->val = a;
  List* m = malloc(…);
  . . .
  return result;
}
```

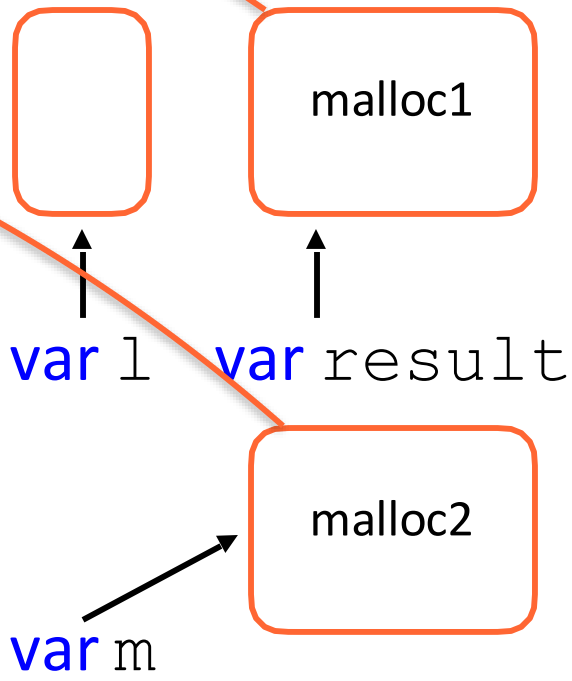


Most modern compilers make use of allocation sites for alias analysis.

Allocation Sites

- Allocation sites can distinguish new pointers from other new pointers.

```
List* add2(List* l, int a) {
  List* result = malloc(…);
  result->next = l;
  result->val = a;
  List* m = malloc(…);
  . . .
  return result;
}
```



Flow-Sensitive Alias Analysis

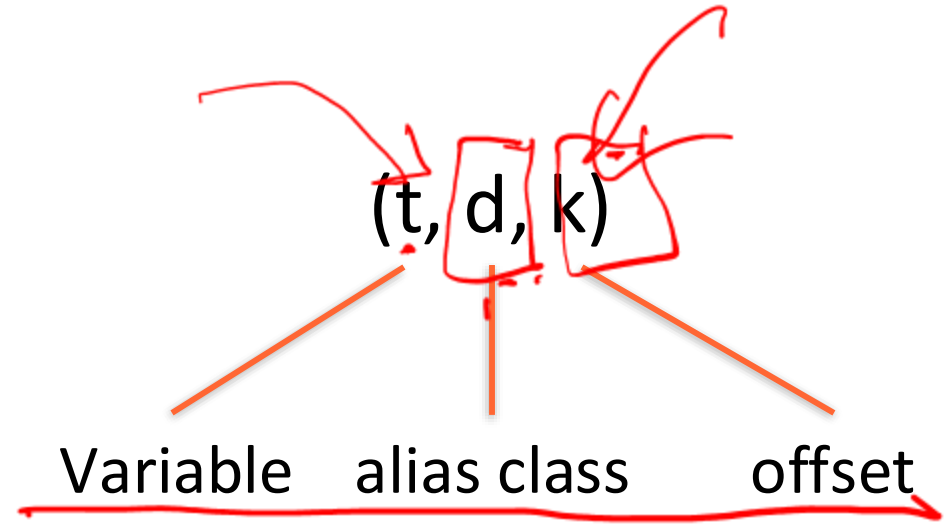
- Aliasing relationships between variables change as the program executes.
- Being accurate in the general case of pointers to pointers requires a flow-sensitive analysis.

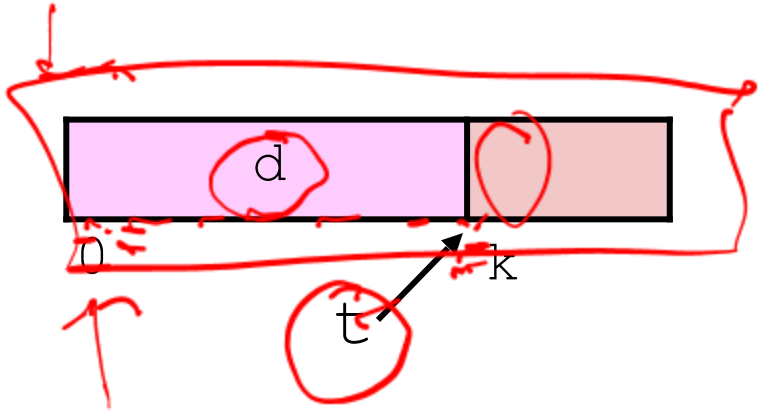
Flow-Sensitivity

- A flow-sensitive analysis distinguishes information about variables at different program locations.
- A flow-insensitive analysis merges information about variables across the whole program (function).

Simple Flow-Sensitive Alias Analysis

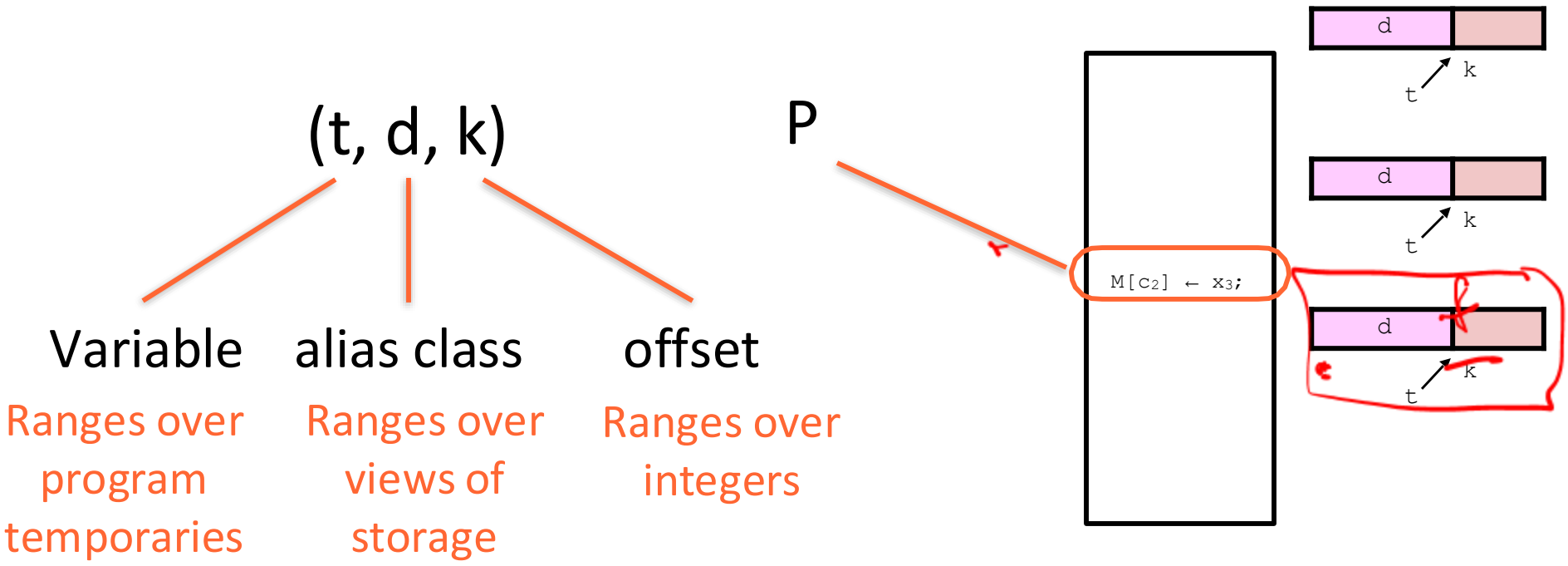
- At every program point P, compute the set of tuples that represent known aliasing relationships after executing the statement at P.





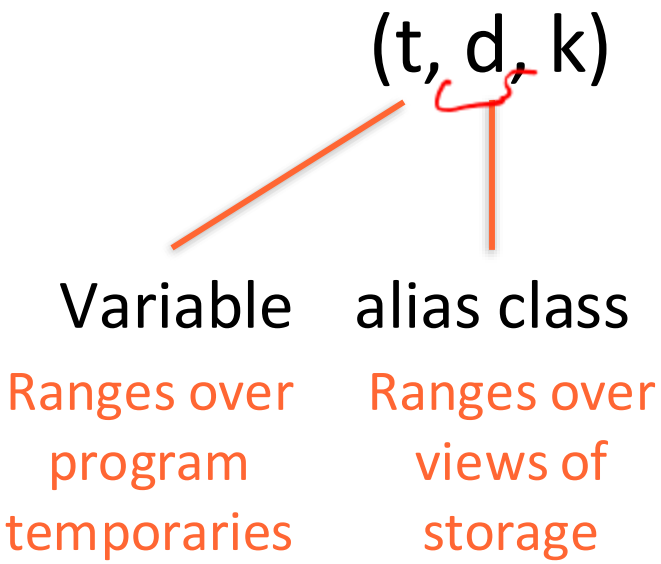
Simple Flow-Sensitive Alias Analysis

- At every program point P, compute the set of tuples that represent known aliasing relationships after executing the statement at P.



Abstract Memory Locations

- Alias classes (d) allow us to tune the precision of the analysis and deal with different languages differently.
- They represent possibly-overlapping views of storage.
- Add an offset to represent abstract memory locations.



- locals of type τ
- structs of type τ
- arrays of type τ
- struct fields S.f
- globals of type τ

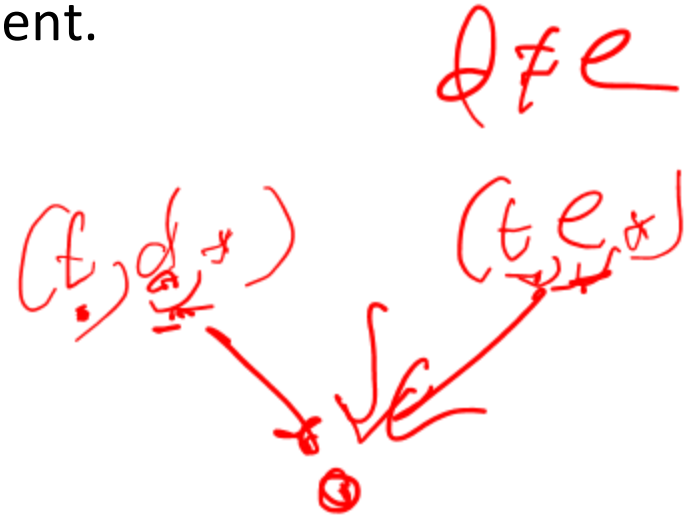
Dataflow Analysis

- After defining alias classes for our language, flow-sensitive analysis falls into the general category of forward dataflow analysis.
- Define a relation for each statement expressing aliasing after the statement in terms of aliasing before the statement.
- Iteratively solve the dataflow equations.

$$in[start] = A$$

$$in[P] = \bigcup_{q \in pred(P)} in[q]$$

$$out[P] = trans_P(in[P])$$



Dataflow Analysis

- After defining alias classes for our language, flow-sensitive analysis falls into the general category of forward dataflow analysis.
- Define a relation for each statement expressing aliasing after the statement in terms of aliasing before the statement.
- Iteratively solve the dataflow equations.

$$in[start] = A$$

$$in[P] = \bigcup_{q \in pred(P)} in[q]$$

$$out[P] = trans_P(in[P])$$

Dataflow Analysis

- After defining alias classes for our language, flow-sensitive analysis falls into the general category of forward dataflow analysis.
- Define a relation for each statement expressing aliasing after the statement in terms of aliasing before the statement.
- Iteratively solve the dataflow equations.

$$in[start] = A$$

$$in[P] = \bigcup_{q \in pred(P)} in[q]$$

$$out[P] = trans_P(in[P])$$

Initial relation A

(x, struct T, 0)

x points to some struct T

```
void compute(struct T* x){
  . . .
}
```

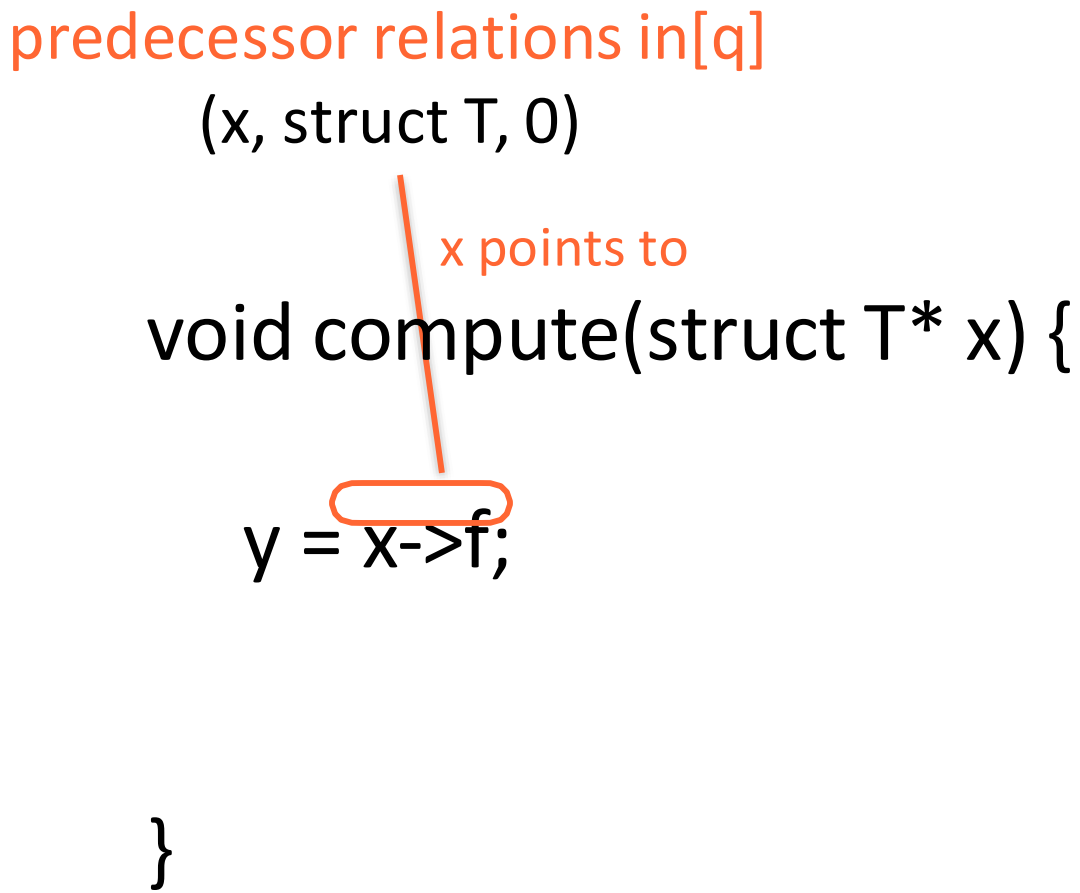
Dataflow Analysis

- After defining alias classes for our language, flow-sensitive analysis falls into the general category of forward dataflow analysis.
- Define a relation for each statement expressing aliasing after the statement in terms of aliasing before the statement.
- Iteratively solve the dataflow equations.

$$in[start] = A$$

$$in[P] = \bigcup_{q \in pred(P)} in[q]$$

$$out[P] = trans_P(in[P])$$



Dataflow Analysis

- After defining alias classes for our language, flow-sensitive analysis falls into the general category of forward dataflow analysis.
- Define a relation for each statement expressing aliasing after the statement in terms of aliasing before the statement.
- Iteratively solve the dataflow equations.

$$in[start] = A$$

$$in[P] = \bigcup_{q \in pred(P)} in[q]$$

$$out[P] = trans_P(in[P])$$

transfer function

what values could y become?

```
void compute(struct T* x) {

  y = x->f;

}
```

Transfer Functions

- Transfer function for simple alias analysis is a pretty straightforward case analysis on the instruction.

Statement	trans(A)
`t ← b`	(A - A(t)) ∪ A(b↦t)
`t ← b + k`	
`t ← b ⊕ c`	
`t ← M[b]`	
`M[a] ← b`	
`if a > b goto L`	
`goto L`	
`f(a … )`	
`t ← alloc(…)`	

Assume A(t) represents the set of all tuples mentioning the variable t.

Assume A(b↦t) represents the set of all tuples by substituting t for b.

Transfer Functions

- Transfer function for simple alias analysis is a pretty straightforward case analysis on the instruction.

Statement	trans(A)
`t ← b`	(A - A(t))
`t ← b + k`	∪
`t ← b ⊕ c`	{(t, d, i) | (b, d, i - k) ∈ A})
`t ← M[b]`	
`M[a] ← b`	
`if a > b goto L`	
`goto L`	
`f(a … )`	
`t ← alloc(…)`	

We can handle pointer arithmetic by making use of statically-known offsets

Transfer Functions

- Transfer function for simple alias analysis is a pretty straightforward case analysis on the instruction.

Statement	trans(A)
`t ← b`	(A - A(t))
`t ← b + k`	(A - A(t))
`t ← b ⊕ c`	(A - A(t))
`t ← M[b]`	(A - A(t)) ∪ unknown(t)
`M[a] ← b`	(A - A(t))
`if a > b goto L`	(A - A(t))
`goto L`	(A - A(t))
`f(a … )`	(A - A(t))
`t ← alloc(…)`	(A - A(t))

Transfer Functions

- Transfer function for simple alias analysis is a pretty straightforward case analysis on the instruction.

Statement	trans(A)
`t ← b`	(A - A(t)) ∪ unknown(t)
`t ← b + k`	
`t ← b ⊕ c`	
`t ← M[b]`	
`M[a] ← b`	
`if a > b goto L`	
`goto L`	
`f(a … )`	
`t ← alloc(…)`	

Depends on the type of t and represents the set of unknown locations for the type, e.g.

(t, struct S, 0)

Transfer Functions

- Transfer function for simple alias analysis is a pretty straightforward case analysis on the instruction.

Statement	trans(A)

`t ← b`
`t ← b + k`
`t ← b ⊕ c`
`t ← M[b]`
`M[a] ← b`
`if a > b goto L`
`goto L`
`f(a … )`
`t ← alloc(…)`
	A

"Ignoring stores" works because we conservatively treat loads as unknown

Transfer Functions

- Transfer function for simple alias analysis is a pretty straightforward case analysis on the instruction.

Statement	trans(A)
`t ← b`	(A - A(t)) ∪ (t, $alloc_{99}$, 0)
`t ← b + k`	
`t ← b ⊕ c`	
`t ← M[b]`	
`M[a] ← b`	
`if a > b goto L`	
`goto L`	
`f(a … )`	
`t ← alloc99(…)`	

We can track each allocation separately.

Dataflow solution to may-alias

- After solving dataflow equations, answering p may-alias q before a program point P is:

$$p \text{ may-alias } q \text{ at } P$$

$$\Leftrightarrow$$

$$(p, d, k) \in in[P] \wedge (q, d, k) \in in[P]$$

Assuming we set up alias classes d to be non-overlapping

Dataflow solution to may-alias

- After solving dataflow equations, answering p may-alias q before a program point P is:

p may-alias q at P

$\Leftrightarrow$

$(p, d_1, k_1) \in in[P] \wedge (q, d_2, k_2) \in in[P]$
$\wedge ((d_1, k_1), (d_2, k_2)) \in overlap$

Assuming we have additional overlap relation

```
struct intlist {
        int val;
        struct intlist* next;
};

struct int foo() {
      struct intlist* p;
      struct intlist* q;
      int a = 1;

      q = alloc(struct intlist);
      q->val = 0;
      q->next NULL;
      p = alloc(struct intlist);
      p->val = 0;
      p->next = q;
      q->val = 5;
      a = p->val;
      return a;
}
```

```
1:  MOVE   t2          <- 0                          p
2:  MOVE   t3          <- 0                          q
3:  MOVE   t4          <- 1                          a
4:  CALL   t5          <- malloc(8)    (t5,4,0)      q <- alloc
5:  PLUS   u1          <- t5,     0
6:  STORE  MEM[u1]     <-  0                         q->val = 0
7:  PLUS   u2          <- t5,     4
8:  STORE  MEM[u2]     <-  0                         q->next = 0
9:  MOVE   t3          <- t5                         q
10: CALL   t6          <- malloc(8)                  p <- alloc
11: PLUS   u3          <- t6,     0
12: STORE  MEM[u3]     <-  0                         p->val = 0
13: PLUS   u4          <- t6,     4
14: STORE  MEM[u4]     <- t3                         p->next = q
15: MOVE   t2          <- t6                         p
16: PLUS   u5          <- t3,     0
17: STORE  MEM[u5]     <-  5                         q->val = 5
18: PLUS   u6          <- t2,     0                  p
19: LOAD   t4          <- MEM[u6]                    a
20: RET                t4
```

```
1:  MOVE   t2         <- 0                              p
2:  MOVE   t3         <- 0                              q
3:  MOVE   t4         <- 1                              a
4:  CALL   t5         <- malloc(8)    (t5,4,0)      q <- alloc
5:  PLUS   u1         <- t5,    0     (u1,4,0)
6:  STORE  MEM[u1]    <-  0                           q->val = 0
7:  PLUS   u2         <- t5,    4
8:  STORE  MEM[u2]    <-  0                           q->next = 0
9:  MOVE   t3         <- t5                           q
10: CALL   t6         <- malloc(8)                    p <- alloc
11: PLUS   u3         <- t6,    0
12: STORE  MEM[u3]    <-  0                           p->val = 0
13: PLUS   u4         <- t6,    4
14: STORE  MEM[u4]    <- t3                           p->next = q
15: MOVE   t2         <- t6                           p
16: PLUS   u5         <- t3,    0
17: STORE  MEM[u5]    <-  5                           q->val = 5
18: PLUS   u6         <- t2,    0                     p
19: LOAD   t4         <- MEM[u6]                      a
20: RET               t4
```

```
1:  MOVE   t2         <- 0                              p
2:  MOVE   t3         <- 0                              q
3:  MOVE   t4         <- 1                              a
4:  CALL   t5         <- malloc(8)    (t5,4,0)      q <- alloc
5:  PLUS   u1         <- t5,    0     (u1,4,0)
6:  STORE  MEM[u1]    <-  0                           q->val = 0
7:  PLUS   u2         <- t5,    4     (u2,4,4)
8:  STORE  MEM[u2]    <-  0                           q->next = 0
9:  MOVE   t3         <- t5                           q
10: CALL   t6         <- malloc(8)                    p <- alloc
11: PLUS   u3         <- t6,    0
12: STORE  MEM[u3]    <-  0                           p->val = 0
13: PLUS   u4         <- t6,    4
14: STORE  MEM[u4]    <- t3                           p->next = q
15: MOVE   t2         <- t6                           p
16: PLUS   u5         <- t3,    0
17: STORE  MEM[u5]    <-  5                           q->val = 5
18: PLUS   u6         <- t2,    0                     p
19: LOAD   t4         <- MEM[u6]                      a
20: RET               t4
```

```
1:  MOVE   t2        <- 0                          p
2:  MOVE   t3        <- 0                          q
3:  MOVE   t4        <- 1                          a
4:  CALL   t5        <- malloc(8)    (t5,4,0)      q <- alloc
5:  PLUS   u1        <- t5,    0     (u1,4,0)
6:  STORE  MEM[u1]   <-  0                         q->val = 0
7:  PLUS   u2        <- t5,    4     (u2,4,4)
8:  STORE  MEM[u2]   <-  0                         q->next = 0
9:  MOVE   t3        <- t5           (t3,4,0)      q
10: CALL   t6        <- malloc(8)                  p <- alloc
11: PLUS   u3        <- t6,    0
12: STORE  MEM[u3]   <-  0                         p->val = 0
13: PLUS   u4        <- t6,    4
14: STORE  MEM[u4]   <- t3                         p->next = q
15: MOVE   t2        <- t6                         p
16: PLUS   u5        <- t3,    0
17: STORE  MEM[u5]   <-  5                         q->val = 5
18: PLUS   u6        <- t2,    0                   p
19: LOAD   t4        <- MEM[u6]                    a
20: RET              t4
```

```
1:  MOVE   t20       <- 0                          p
2:  MOVE   t30       <- 0                          q
3:  MOVE   t40       <- 1                          a
4:  CALL   t5        <- malloc(8)    (t5,4,0)      q <- alloc
5:  PLUS   u1        <- t5,    0     (u1,4,0)
6:  STORE  MEM[u1]   <-  0                         q->val = 0
7:  PLUS   u2        <- t5,    4     (u2,4,4)
8:  STORE  MEM[u2]   <-  0                         q->next = 0
9:  MOVE   t3        <- t5           (t3,4,0)      q
10: CALL   t6        <- malloc(8)    (t6,10,0)     p <- alloc
11: PLUS   u3        <- t6,    0
12: STORE  MEM[u3]   <-  0                         p->val = 0
13: PLUS   u4        <- t6,    4
14: STORE  MEM[u4]   <- t3                         p->next = q
15: MOVE   t2        <- t6                         p
16: PLUS   u5        <- t3,    0
17: STORE  MEM[u5]   <-  5                         q->val = 5
18: PLUS   u6        <- t2,    0                   p
19: LOAD   t4        <- MEM[u6]                    a
20: RET              t4
```

```
1:  MOVE   t2        <- 0                          p
2:  MOVE   t3        <- 0                          q
3:  MOVE   t4        <- 1                          a
4:  CALL   t5        <- malloc(8)    (t5,4,0)      q <- alloc
5:  PLUS   u1        <- t5,    0     (u1,4,0)
6:  STORE  MEM[u1]   <-  0                         q->val = 0
7:  PLUS   u2        <- t5,    4     (u2,4,4)
8:  STORE  MEM[u2]   <-  0                         q->next = 0
9:  MOVE   t3        <- t5           (t3,4,0)      q
10: CALL   t6        <- malloc(8)    (t6,10,0)     p <- alloc
11: PLUS   u3        <- t6,    0     (u3,10,0)
12: STORE  MEM[u3]   <-  0                         p->val = 0
13: PLUS   u4        <- t6,    4
14: STORE  MEM[u4]   <- t3                         p->next = q
15: MOVE   t2        <- t6                         p
16: PLUS   u5        <- t3,    0
17: STORE  MEM[u5]   <-  5                         q->val = 5
18: PLUS   u6        <- t2,    0                   p
19: LOAD   t4        <- MEM[u6]                    a
20: RET              t4
```

```
1:  MOVE   t2        <- 0                          p
2:  MOVE   t3        <- 0                          q
3:  MOVE   t4        <- 1                          a
4:  CALL   t5        <- malloc(8)    (t5,4,0)      q <- alloc
5:  PLUS   u1        <- t5,    0     (u1,4,0)
6:  STORE  MEM[u1]   <-  0                         q->val = 0
7:  PLUS   u2        <- t5,    4     (u2,4,4)
8:  STORE  MEM[u2]   <-  0                         q->next = 0
9:  MOVE   t3        <- t5           (t3,4,0)      q
10: CALL   t6        <- malloc(8)    (t6,10,0)     p <- alloc
11: PLUS   u3        <- t6,    0     (u3,10,0)
12: STORE  MEM[u3]   <-  0                         p->val = 0
13: PLUS   u4        <- t6,    4     (u4,10,4)
14: STORE  MEM[u4]   <- t3                         p->next = q
15: MOVE   t2        <- t6                         p
16: PLUS   u5        <- t3,    0
17: STORE  MEM[u5]   <-  5                         q->val = 5
18: PLUS   u6        <- t2,    0                   p
19: LOAD   t4        <- MEM[u6]                    a
20: RET              t4
```

```
1:  MOVE   t2        <- 0                          p
2:  MOVE   t3        <- 0                          q
3:  MOVE   t4        <- 1                          a
4:  CALL   t5        <- malloc(8)    (t5,4,0)      q <- alloc
5:  PLUS   u1        <- t5,    0     (u1,4,0)
6:  STORE  MEM[u1]   <-  0                         q->val = 0
7:  PLUS   u2        <- t5,    4     (u2,4,4)
8:  STORE  MEM[u2]   <-  0                         q->next = 0
9:  MOVE   t3        <- t5           (t3,4,0)      q
10: CALL   t6        <- malloc(8)    (t6,10,0)     p <- alloc
11: PLUS   u3        <- t6,    0     (u3,10,0)
12: STORE  MEM[u3]   <-  0                         p->val = 0
13: PLUS   u4        <- t6,    4     (u4,10,4)
14: STORE  MEM[u4]   <- t3                         p->next = q
15: MOVE   t2        <- t6           (t2,10,0)     p
16: PLUS   u5        <- t3,    0
17: STORE  MEM[u5]   <-  5                         q->val = 5
18: PLUS   u6        <- t2,    0                   p
19: LOAD   t4        <- MEM[u6]                    a
20: RET              t4
```

```
1:  MOVE   t2        <- 0                          p
2:  MOVE   t3        <- 0                          q
3:  MOVE   t4        <- 1                          a
4:  CALL   t5        <- malloc(8)    (t5,4,0)      q <- alloc
5:  PLUS   u1        <- t5,    0     (u1,4,0)
6:  STORE  MEM[u1]   <-  0                         q->val = 0
7:  PLUS   u2        <- t5,    4     (u2,4,4)
8:  STORE  MEM[u2]   <-  0                         q->next = 0
9:  MOVE   t3        <- t5           (t3,4,0)      q
10: CALL   t6        <- malloc(8)    (t6,10,0)     p <- alloc
11: PLUS   u3        <- t6,    0     (u3,10,0)
12: STORE  MEM[u3]   <-  0                         p->val = 0
13: PLUS   u4        <- t6,    4     (u4,10,4)
14: STORE  MEM[u4]   <- t3                         p->next = q
15: MOVE   t2        <- t6           (t2,10,0)     p
16: PLUS   u5        <- t3,    0     (u5,4,0)
17: STORE  MEM[u5]   <-  5                         q->val = 5
18: PLUS   u6        <- t2,    0                   p
19: LOAD   t4        <- MEM[u6]                    a
20: RET              t4
```

```
1:  MOVE   t2        <- 0                          p
2:  MOVE   t3        <- 0                          q
3:  MOVE   t4        <- 1                          a
4:  CALL   t5        <- malloc(8)    (t5,4,0)      q <- alloc
5:  PLUS   u1        <- t5,    0     (u1,4,0)
6:  STORE  MEM[u1]   <-  0                         q->val = 0
7:  PLUS   u2        <- t5,    4     (u2,4,4)
8:  STORE  MEM[u2]   <-  0                         q->next = 0
9:  MOVE   t3        <- t5           (t3,4,0)      q
10: CALL   t6        <- malloc(8)    (t6,10,0)     p <- alloc
11: PLUS   u3        <- t6,    0     (u3,10,0)
12: STORE  MEM[u3]   <-  0                         p->val = 0
13: PLUS   u4        <- t6,    4     (u4,10,4)
14: STORE  MEM[u4]   <- t3                         p->next = q
15: MOVE   t2        <- t6           (t2,10,0)     p
16: PLUS   u5        <- t3,    0     (u5,4,0)
17: STORE  MEM[u5]   <-  5                         q->val = 5
18: PLUS   u6        <- t2,    0     (u6,4,0)      p
19: LOAD   t4        <- MEM[u6]                    a
20: RET              t4
```

```
1:  MOVE   t2         <- 0                              p
2:  MOVE   t3         <- 0                              q
3:  MOVE   t4         <- 1                              a
4:  CALL   t5         <- malloc(8)     (t5,4,0)         q <- alloc
5:  PLUS   u1         <- t5,     0     (u1,4,0)
6:  STORE  MEM[u1]    <-  0                             q->val = 0
7:  PLUS   u2         <- t5,     4     (u2,4,4)
8:  STORE  MEM[u2]    <-  0                             q->next = 0
9:  MOVE   t3         <- t5            (t3,4,0)         q
10: CALL   t6         <- malloc(8)     (t6,10,0)        p <- alloc
11: PLUS   u3         <- t6,     0     (u3,10,0)
12: STORE  MEM[u3]    <-  0                             p->val = 0
13: PLUS   u4         <- t6,     4     (u4,10,4)
14: STORE  MEM[u4]    <- t3                             p->next = q
15: MOVE   t2         <- t6            (t2,10,0)        p
16: PLUS   u5         <- t3,     0     (u5,4,0)
17: STORE  MEM[u5]    <-  5                             q->val = 5
18: PLUS   u6         <- t2,     0     (u6,10,0)        p
19: LOAD   t4         <- MEM[u6]                        a
20: RET               t4
```

```
1:  MOVE   t2          <- 0                                  p
2:  MOVE   t3          <- 0                                  q
3:  MOVE   t4          <- 1                                  a
4:  CALL   t5          <- malloc(8)     (t5,4,0)     q <- alloc
5:  PLUS   u1          <- t5,     0     (u1,4,0)
6:  STORE  MEM[u1]     <-  0                           q->val = 0
7:  PLUS   u2          <- t5,     4     (u2,4,4)
8:  STORE  MEM[u2]     <-  0                           q->next = 0
9:  MOVE   t3          <- t5            (t3,4,0)     q
10: CALL   t6          <- malloc(8)     (t6,10,0)    p <- alloc
11: PLUS   u3          <- t6,     0     (u3,10,0)
12: STORE  MEM[u3]     <-  0                           p->val = 0
13: PLUS   u4          <- t6,     4     (u4,10,4)
14: STORE  MEM[u4]     <- t3                           p->next = q
15: MOVE   t2          <- t6            (t2,10,0)    p
16: PLUS   u5          <- t3,     0     (u5,4,0)
17: STORE  MEM[u5]     <-  5                           q->val = 5
18: PLUS   u6          <- t2,     0     (u6,10,0)    p
19: MOVE   t4          <- 0                            a
20: RET                t4
```

```
1:  MOVE   t2         <- 0                                  p
2:  MOVE   t3         <- 0                                  q
3:  MOVE   t4         <- 1                                  a
4:  CALL   t5         <- malloc(8)     (t5,4,0)     q <- alloc
5:  PLUS   u1         <- t5,     0     (u1,4,0)
6:  STORE  MEM[u1]    <-  0                           q->val = 0
7:  PLUS   u2         <- t5,     4     (u2,4,4)
8:  STORE  MEM[u2]    <-  0                           q->next = 0
9:  MOVE   t3         <- t5            (t3,4,0)     q
10: CALL   t6         <- malloc(8)     (t6,10,0)    p <- alloc
11: PLUS   u3         <- t6,     0     (u3,10,0)
12: STORE  MEM[u3]    <-  0                           p->val = 0
13: PLUS   u4         <- t6,     4     (u4,10,4)
14: STORE  MEM[u4]    <- t3                           p->next = q
15: MOVE   t2         <- t6            (t2,10,0)    p
16: PLUS   u5         <- t3,     0     (u5,4,0)
17: STORE  MEM[u5]    <-  5                           q->val = 5
18: PLUS   u6         <-  t2,    0     (u6,10,0)    p
19: MOVE   t4         <- 0                            a
20: RET               0
```

```
struct intlist* foo() {
        struct intlist* p;
        struct intlist* q;
        int a = 1;

        p = alloc(struct intlist);
        p->val = 0;
        p->next = 0;

        q = alloc(struct intlist);
        q->val = 6;
        q->next = p;

        if (a == 0) p = q;
        p->val = 4;
        return p;
}
```

```
1:  CALL   t2        <- malloc(8)      p
2:  PLUS   u1        <- t2,0
3:  STR    MEM[u1]   <-  0             p->val = 0
4:  PLUS   u2        <- t2, 4
5:  STR    MEM[u2]   <- 0              p->next = 0
6:  MOVE   t3        <- t2             x=p
7:  CALL   t4        <- malloc(8)      q
8:  PLUS   u3        <- t4,0
9:  STR    MEM[u3]   <- 6              q->val = 6
10: PLUS   u4        <- t4,0
11: STR    MEM[u4]   <- t3             q->next = p
12: MOVE   t5        <- t4             q
13: MOVE   t6        <- 1              a
14: CJUMP  t6 == 0, ifT0,ifF1          a == 0 ?
ifT0:
16: MOVE   t3'       <- t5             x=q
ifF1:
18: PHI    t3''      <- ϕ(t3,t3')
19: PLUS   u5        <- t3'', 0
20: STR    MEM[u5]   <- 4
21: RET    t2
```

```
1:  CALL   t2       <- malloc(8) (t2,1,0)      p
2:  PLUS   u1       <- t2,0
3:  STR    MEM[u1] <-  0                       p->val = 0
4:  PLUS   u2       <- t2, 4
5:  STR    MEM[u2] <- 0                        p->next = 0
6:  MOVE   t3       <- t2                      x=p
7:  CALL   t4       <- malloc(8)               q
8:  PLUS   u3       <- t4,0
9:  STR    MEM[u3] <- 6                        q->val = 6
10: PLUS   u4       <- t4,0
11: STR    MEM[u4] <- t3                       q->next = p
12: MOVE   t5       <- t4                      q
13: MOVE   t6       <- 1                       a
14: CJUMP  t6 == 0, ifT0,ifF1                  a == 0 ?
ifT0:
16: MOVE   t3'      <- t5                      x=q
ifF1:
18: PHI    t3''     <- ϕ(t3,t3')
19: PLUS   u5       <- t3'', 0
20: STR    MEM[u5] <- 4
21: RET    t2
```

```
1:  CALL   t2       <- malloc(8) (t2,1,0)      p
2:  PLUS   u1       <- t2,0      (u1,1,0)
3:  STR    MEM[u1] <-  0                       p->val = 0
4:  PLUS   u2       <- t2, 4     (u2,1,4)
5:  STR    MEM[u2] <- 0                        p->next = 0
6:  MOVE   t3       <- t2                      x=p
7:  CALL   t4       <- malloc(8)               q
8:  PLUS   u3       <- t4,0
9:  STR    MEM[u3] <- 6                        q->val = 6
10: PLUS   u4       <- t4,0
11: STR    MEM[u4] <- t3                       q->next = p
12: MOVE   t5       <- t4                      q
13: MOVE   t6       <- 1                       a
14: CJUMP  t6 == 0, ifT0,ifF1                  a == 0 ?
ifT0:
16: MOVE   t3'      <- t5                      x=q
ifF1:
18: PHI    t3''     <- ϕ(t3,t3')
19: PLUS   u5       <- t3'', 0
20: STR    MEM[u5] <- 4
21: RET    t2
```

```
1:  CALL   t2       <- malloc(8) (t2,1,0)      p
2:  PLUS   u1       <- t2,0      (u1,1,0)
3:  STR    MEM[u1]  <-  0                      p->val = 0
4:  PLUS   u2       <- t2, 4     (u2,1,4)
5:  STR    MEM[u2]  <- 0                       p->next = 0
6:  MOVE   t3       <- t2        (t3,1,0)      x=p
7:  CALL   t4       <- malloc(8) (t4,7,0)      q
8:  PLUS   u3       <- t4,0      (u3,7,0)
9:  STR    MEM[u3]  <- 6                       q->val = 6
10: PLUS   u4       <- t4,0      (u4,7,0)
11: STR    MEM[u4]  <- t3                      q->next = p
12: MOVE   t5       <- t4        (t5,7,0)      q
13: MOVE   t6       <- 1                       a
14: CJUMP  t6 == 0, ifT0,ifF1                  a == 0 ?
ifT0:
16: MOVE   t3'      <- t5                      x=q
ifF1:
18: PHI    t3''     <- ϕ(t3,t3')
19: PLUS   u5       <- t3'', 0
20: STR    MEM[u5]  <- 4
21: RET    t2
```

```
1:  CALL   t2       <- malloc(8) (t2,1,0)      p
2:  PLUS   u1       <- t2,0      (u1,1,0)
3:  STR    MEM[u1]  <-  0                      p->val = 0
4:  PLUS   u2       <- t2, 4     (u2,1,4)
5:  STR    MEM[u2]  <- 0                       p->next = 0
6:  MOVE   t3       <- t2        (t3,1,0)      x=p
7:  CALL   t4       <- malloc(8) (t4,7,0)      q
8:  PLUS   u3       <- t4,0      (u3,7,0)
9:  STR    MEM[u3]  <- 6                       q->val = 6
10: PLUS   u4       <- t4,0      (u4,7,0)
11: STR    MEM[u4]  <- t3                      q->next = p
12: MOVE   t5       <- t4        (t5,7,0)      q
13: MOVE   t6       <- 1                       a
14: CJUMP  t6 == 0, ifT0,ifF1                  a == 0 ?
ifT0:
16: MOVE   t3'      <- t5        (t3',7,0)     x=q
ifF1:
18: PHI    t3''     <- ϕ(t3,t3')
19: PLUS   u5       <- t3'', 0
20: STR    MEM[u5]  <- 4
21: RET    t2
```

```
1:  CALL   t2       <- malloc(8) (t2,1,0)      p
2:  PLUS   u1       <- t2,0      (u1,1,0)
3:  STR    MEM[u1]  <-  0                      p->val = 0
4:  PLUS   u2       <- t2, 4     (u2,1,4)
5:  STR    MEM[u2]  <- 0                       p->next = 0
6:  MOVE   t3       <- t2        (t3,1,0)      x=p
7:  CALL   t4       <- malloc(8) (t4,7,0)      q
8:  PLUS   u3       <- t4,0      (u3,7,0)
9:  STR    MEM[u3]  <- 6                       q->val = 6
10: PLUS   u4       <- t4,0      (u4,7,0)
11: STR    MEM[u4]  <- t3                      q->next = p
12: MOVE   t5       <- t4        (t5,7,0)      q
13: MOVE   t6       <- 1                       a
14: CJUMP  t6 == 0, ifT0,ifF1                  a == 0 ?
ifT0:
16: MOVE   t3'      <- t5        (t3',7,0)     x=q
ifF1:
18: PHI    t3''     <- ϕ(t3,t3') (t3'',7,0),(t3'',1,0)
19: PLUS   u5       <- t3'', 0
20: STR    MEM[u5]  <- 4
21: RET    t2
```

```
1:  CALL   t2       <- malloc(8) (t2,1,0)      p
2:  PLUS   u1       <- t2,0      (u1,1,0)
3:  STR    MEM[u1]  <-  0                      p->val = 0
4:  PLUS   u2       <- t2, 4     (u2,1,4)
5:  STR    MEM[u2]  <- 0                       p->next = 0
6:  MOVE   t3       <- t2        (t3,1,0)      x=p
7:  CALL   t4       <- malloc(8) (t4,7,0)      q
8:  PLUS   u3       <- t4,0      (u3,7,0)
9:  STR    MEM[u3]  <- 6                       q->val = 6
10: PLUS   u4       <- t4,0      (u4,7,0)
11: STR    MEM[u4]  <- t3                      q->next = p
12: MOVE   t5       <- t4        (t5,7,0)      q
13: MOVE   t6       <- 1                       a
14: CJUMP  t6 == 0, ifT0,ifF1                  a == 0 ?
ifT0:
16: MOVE   t3'      <- t5        (t3',7,0)     x=q
ifF1:
18: PHI    t3''     <- ϕ(t3,t3') (t3'',7,0),(t3'',1,0)
19: PLUS   u5       <- t3'', 0   (u5,7,0),(u5,1,0)
20: STR    MEM[u5]  <- 4
21: RET    t2
```

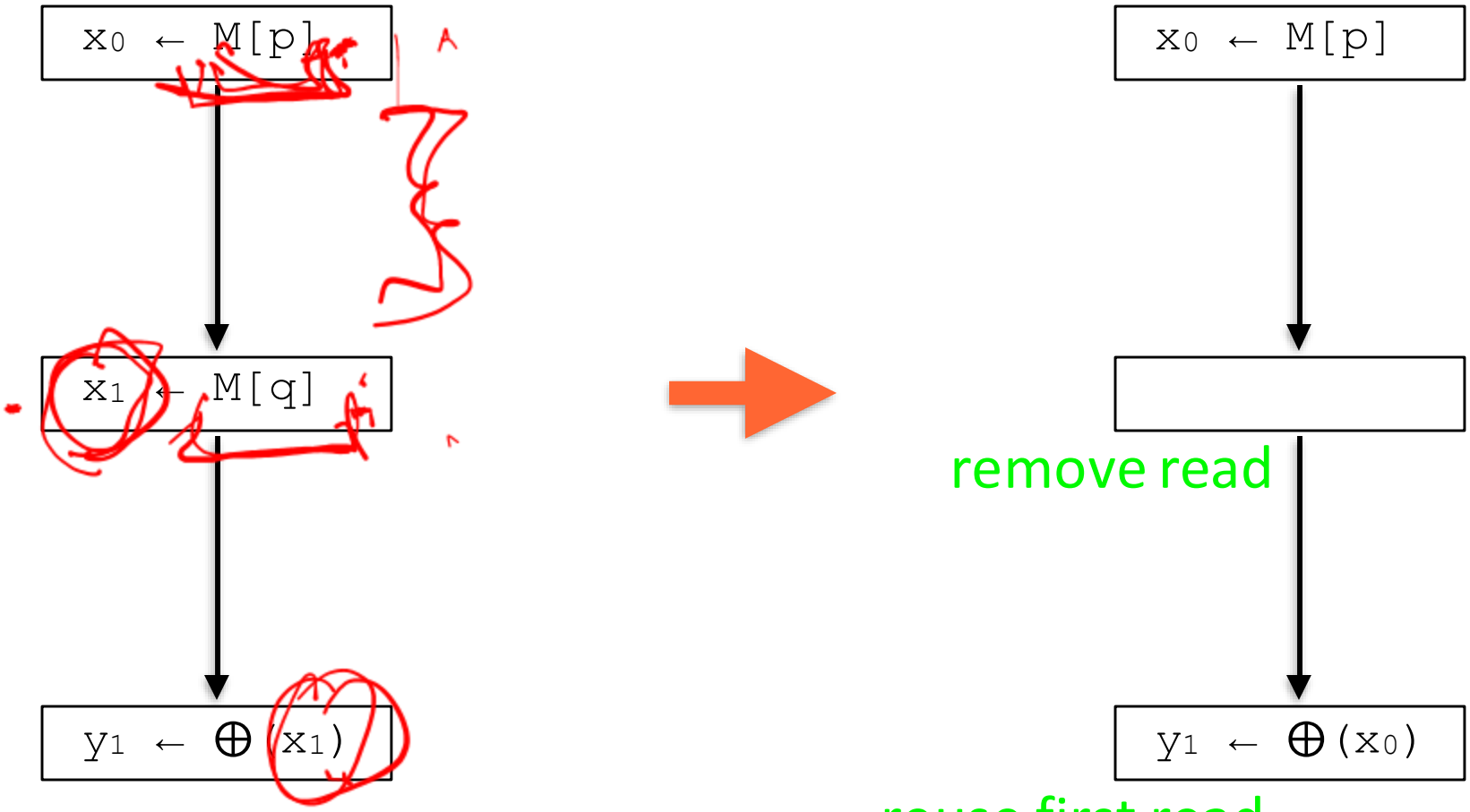
The Heap Triple Crown

- Three optimizations based on alias analysis work great together.
  - Load Elimination
  - Load-Store Forwarding
  - Store Elimination
- All three require correct (conservative) aliasing information.

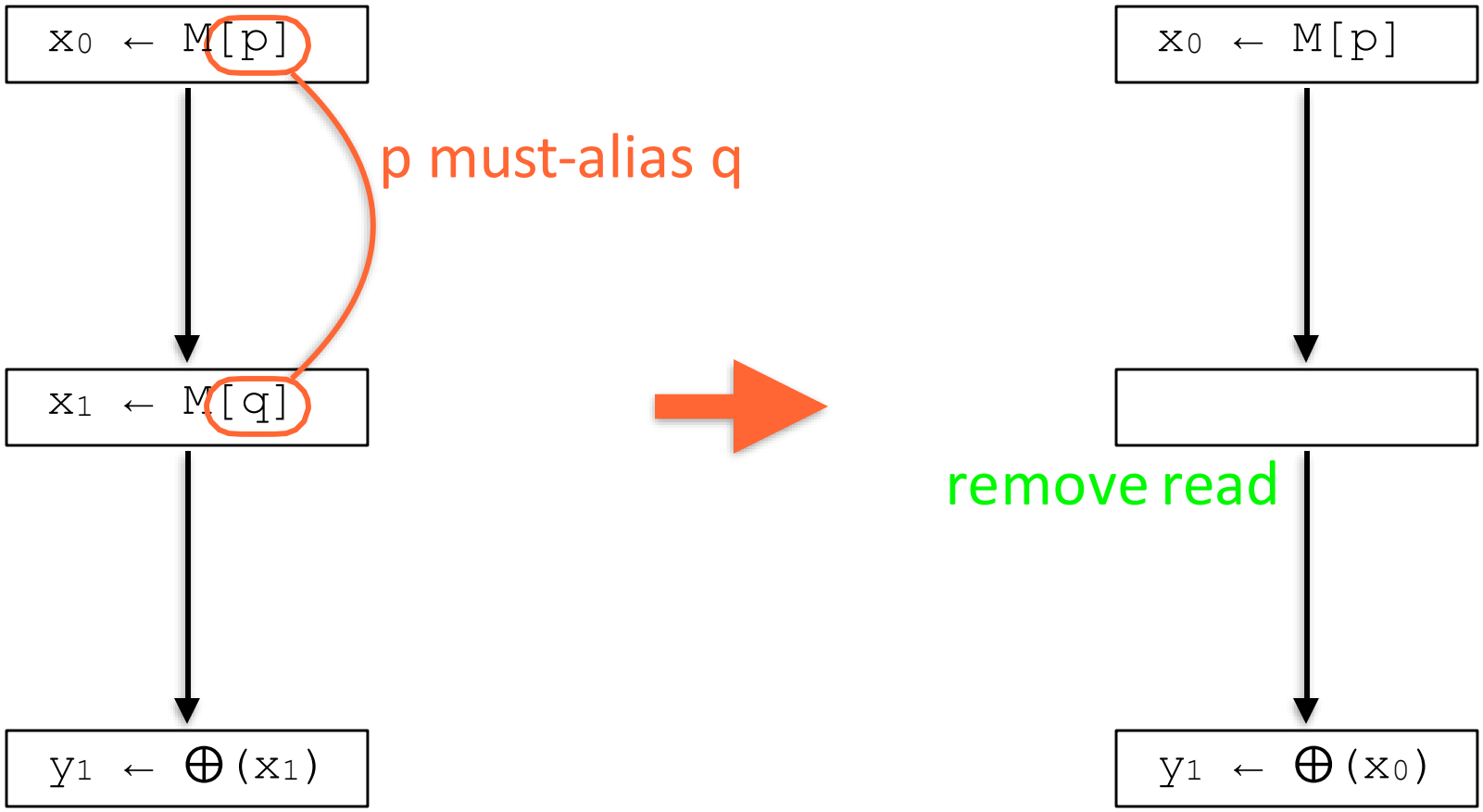
Load Elimination

- Many programs have redundant loads
- Reusing a previously-loaded value (safely, after alias analysis) is a form of common subexpression elimination.
- Can be done together or in a separate (lightweight) pass.
- Can be done locally or globally.

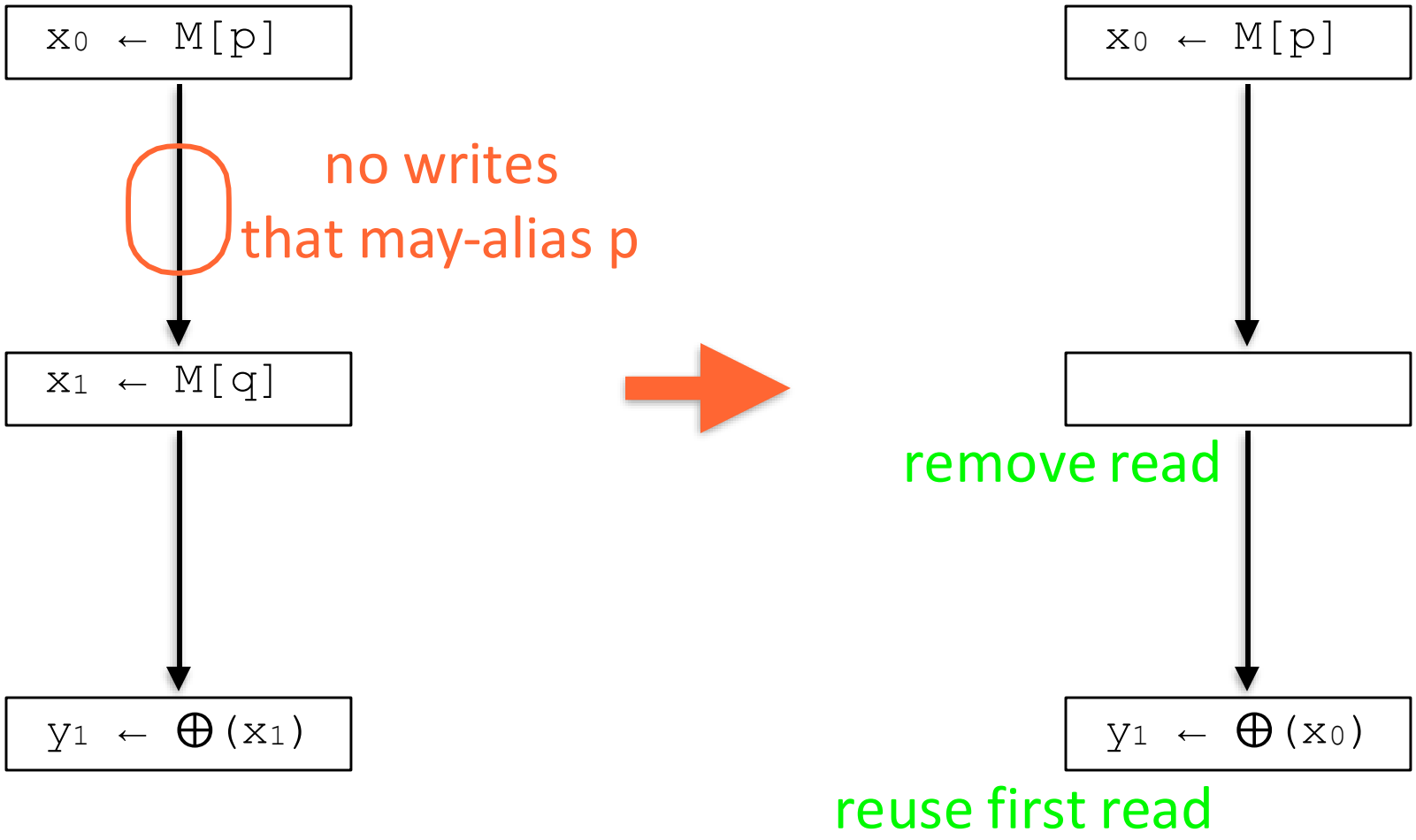
Load Elimination Illustration



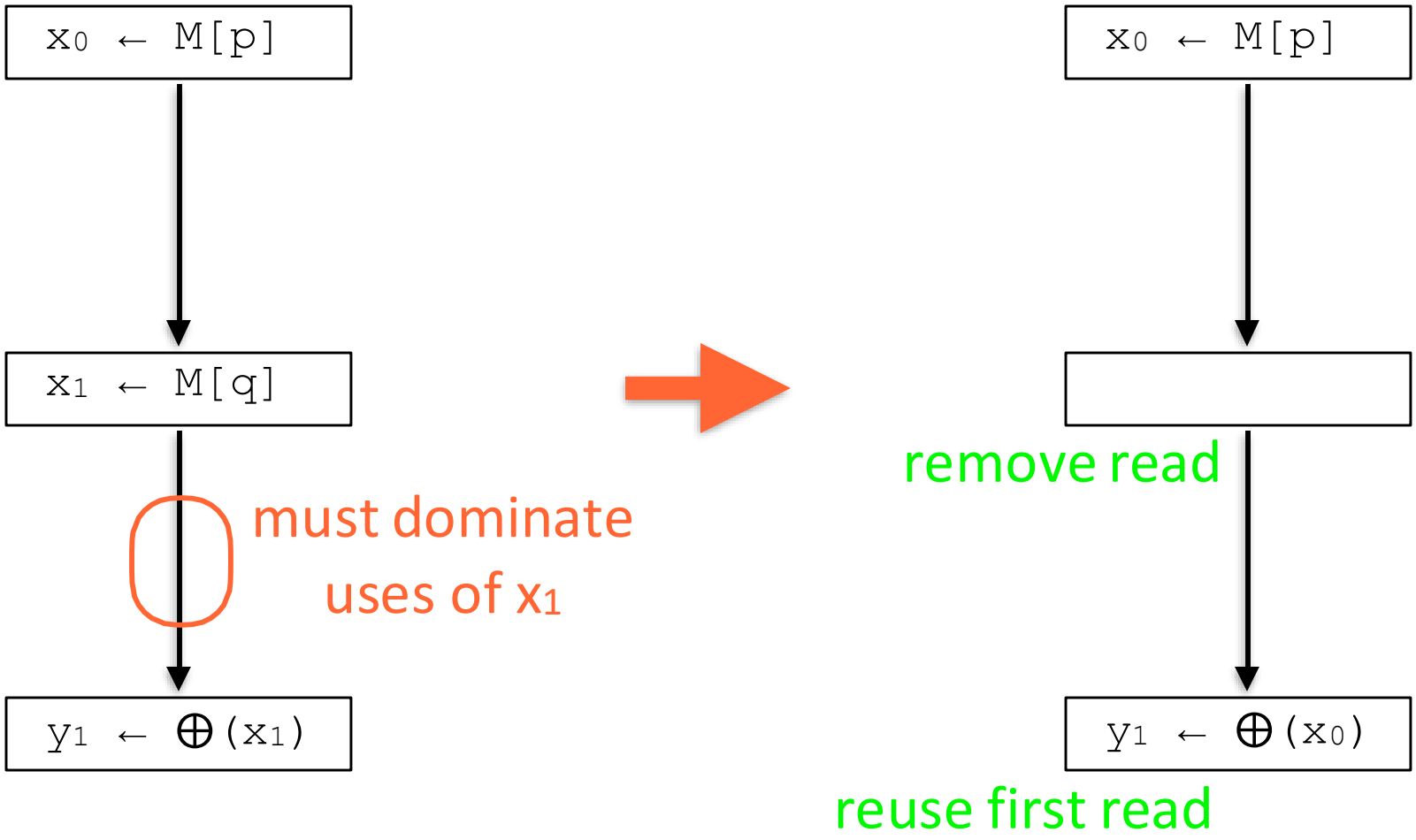
Load Elimination Illustration



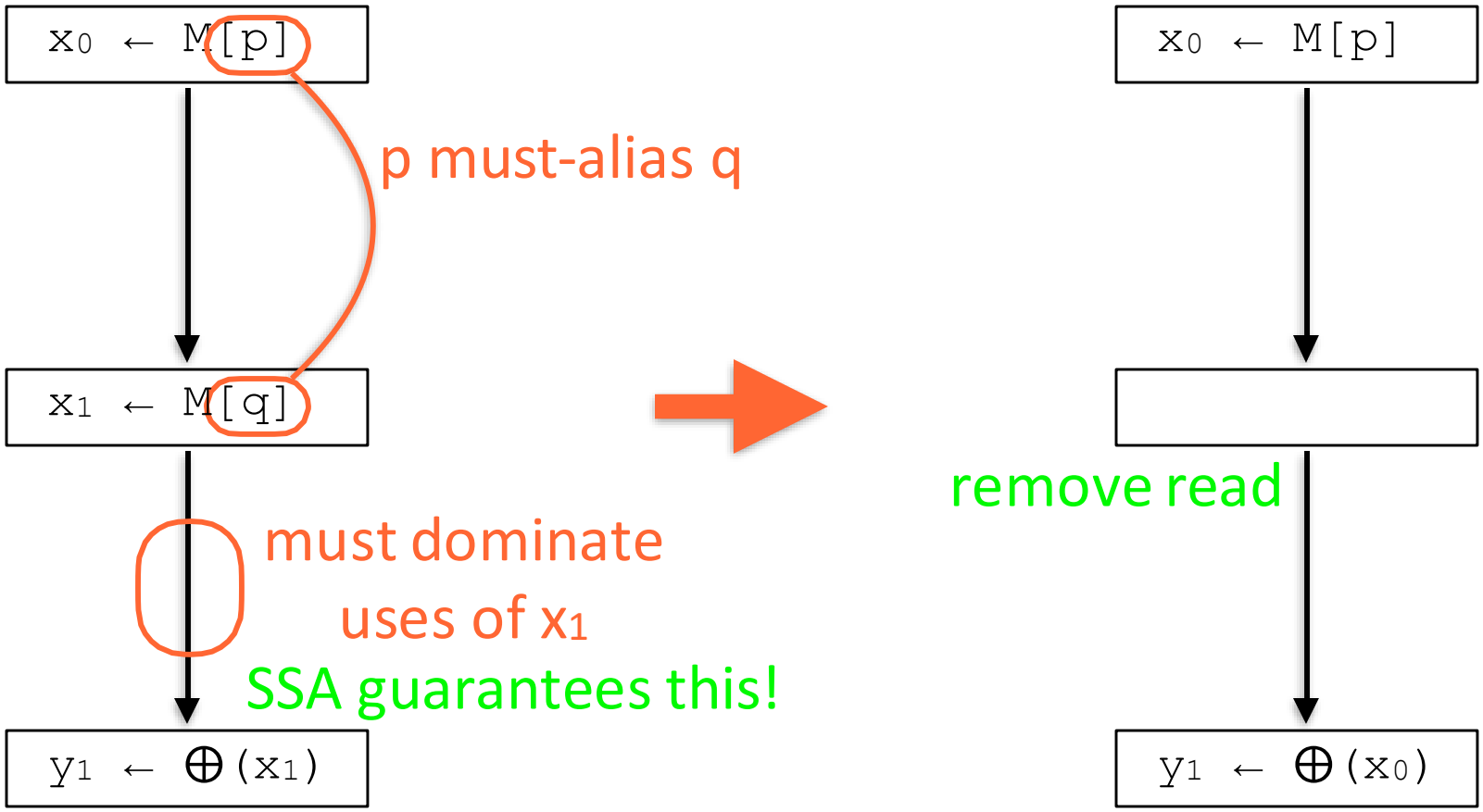
Load Elimination Illustration



Load Elimination Illustration



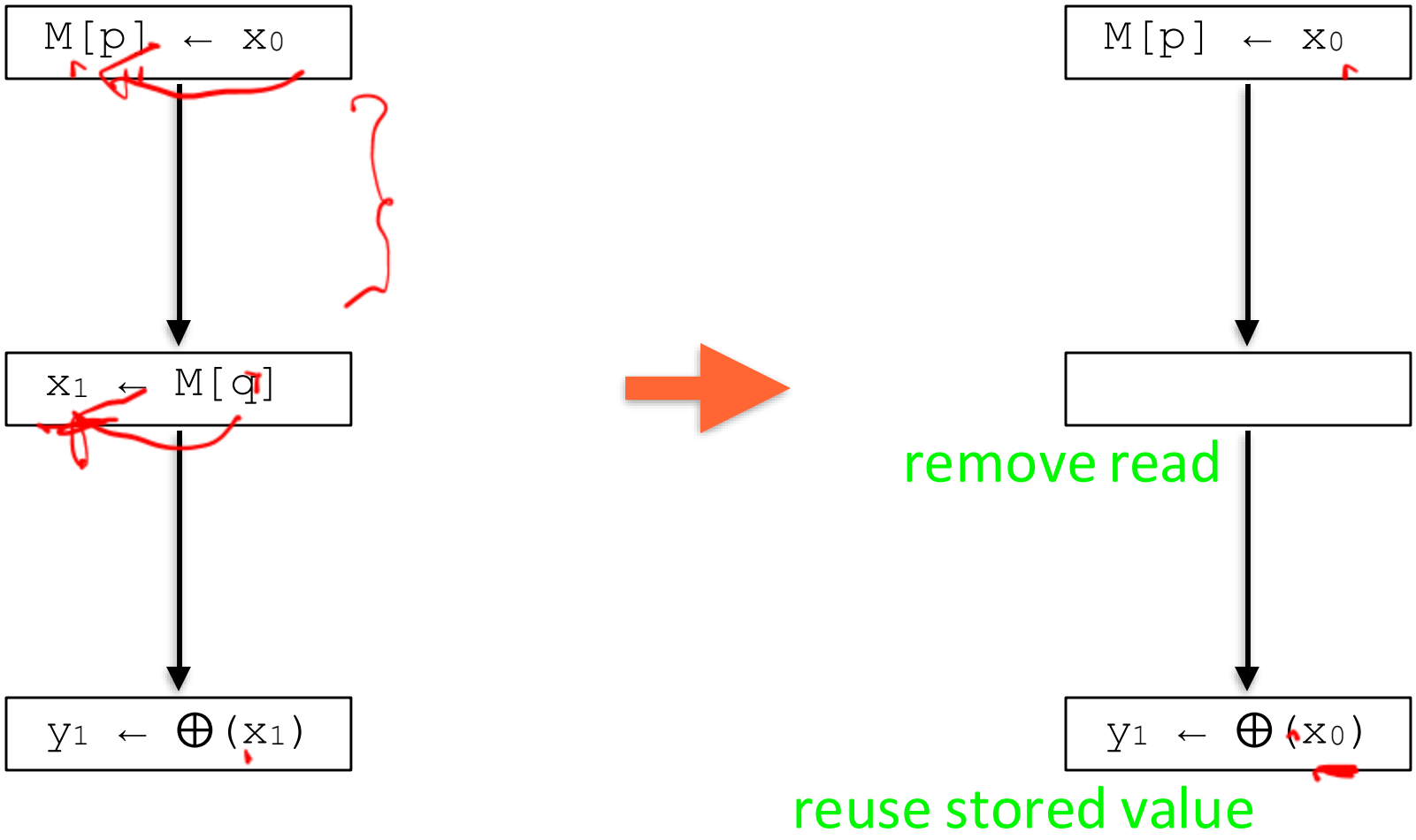
Load Elimination Illustration



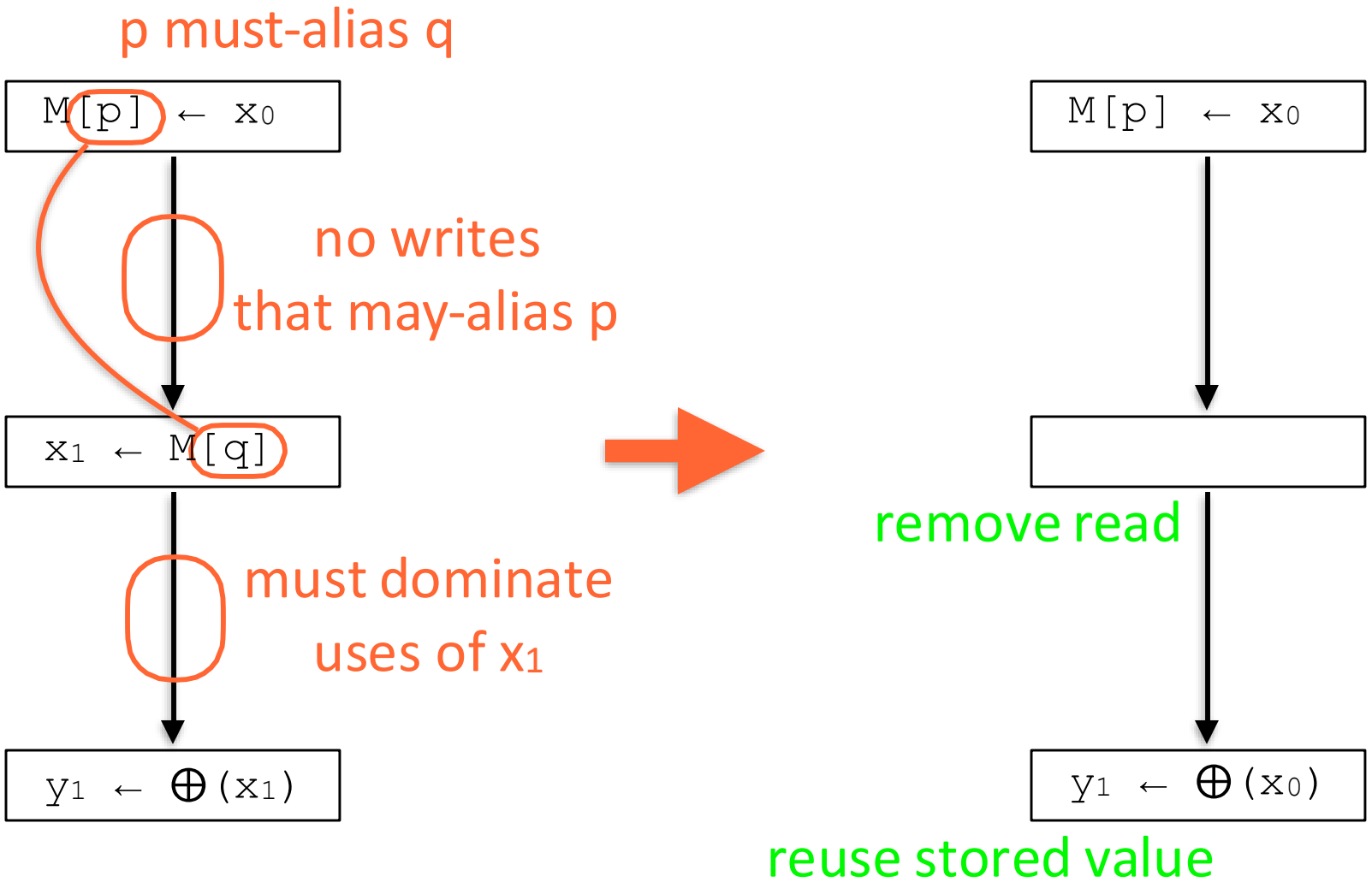
Load-Store Forwarding

- Many programs store to memory and then immediately load the value back.
- Reusing a previously-stored value (safely, after alias analysis) is a slightly different form of common subexpression elimination.
- Can be done together or in a separate (lightweight) pass.
- Can be done locally or globally.

Load-Store Forwarding Illustration



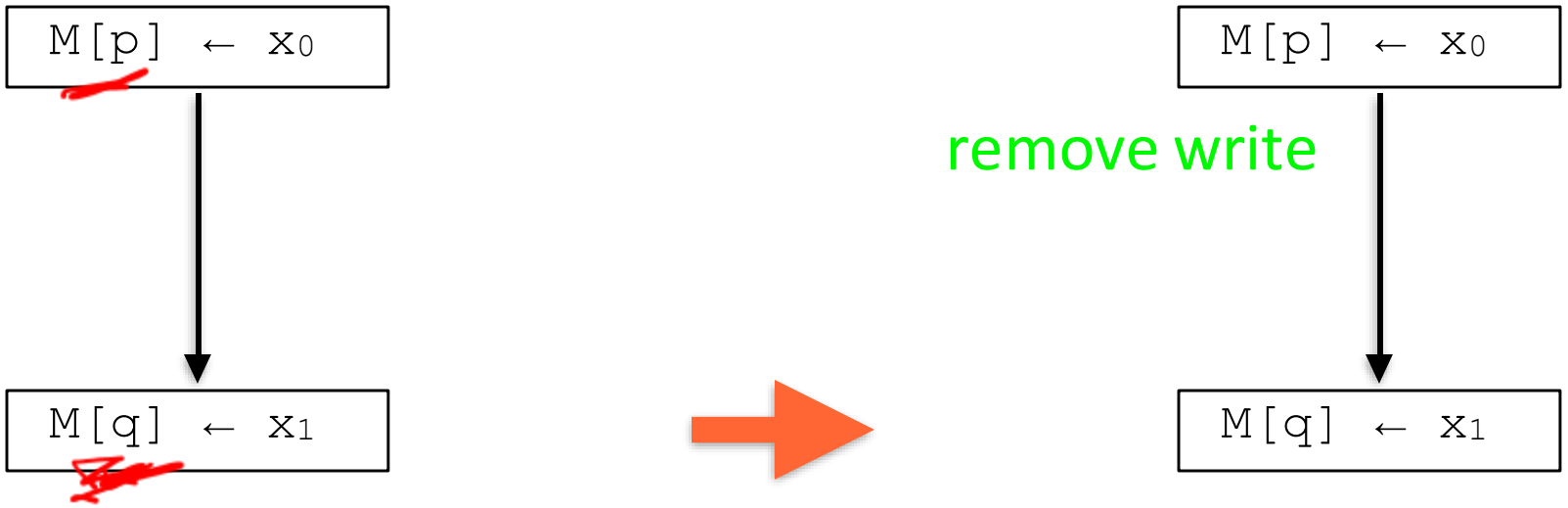
Load-Store Forwarding Illustration



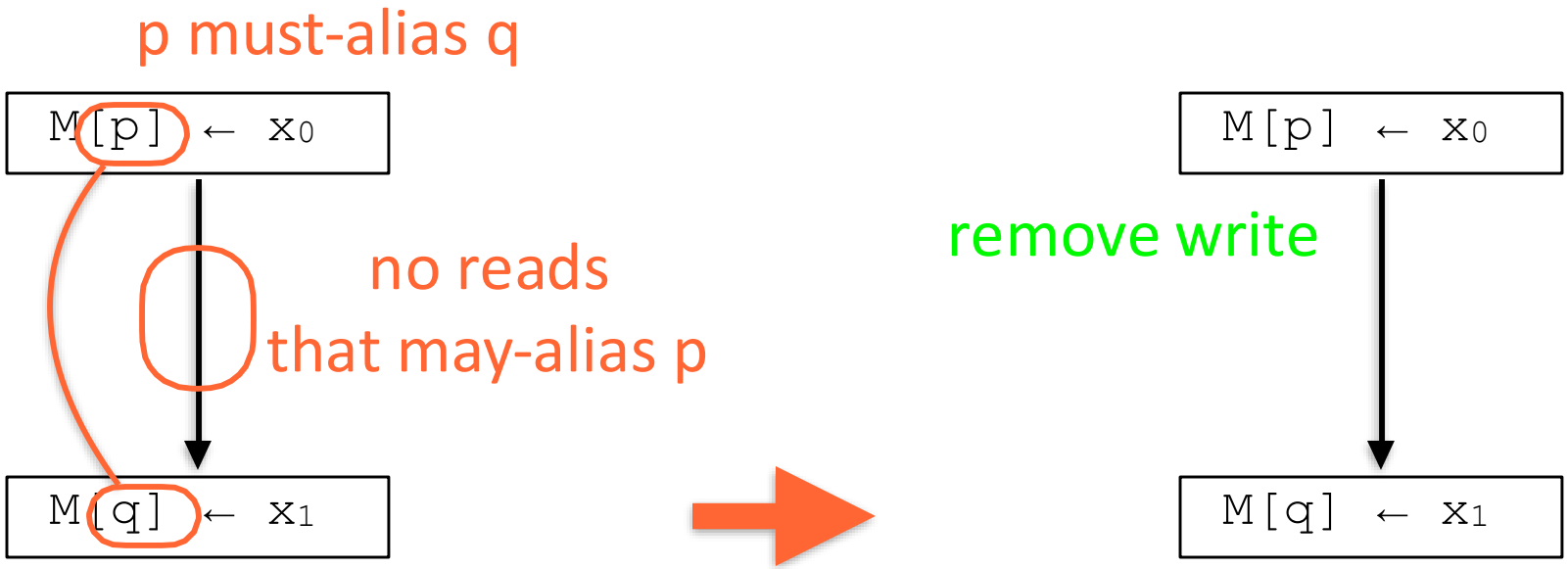
Store Elimination

- Many programs store to memory and then immediately overwrite the previously-stored value.
- Eliminating redundant stores is slightly different than CSE.
- Can be done together or in a separate (lightweight) pass.
- Can be done locally or globally.

Store Elimination Illustration



Store Elimination Illustration



Implementing LE/LSF/SE

- How to compute the loads/stores are available?

Implementing LE/LSF/SE

- How to compute the loads/stores that are available?
- More dataflow analysis!
- Use the standard GEN and KILL strategy.
- OUT[P] = IN[P] - KILL[P] + GEN[P]

Implementing LE/LSF/SE

- How to compute the loads/stores that are available?
- More dataflow analysis!
- Use the standard GEN and KILL strategy.
- OUT[P] = IN[P] - KILL[P] + GEN[P]

IN and OUT sets store expressions that are available to be reused

Stores and calls (other side effects) add to the KILL set, using the results from alias analysis

Loads and stores add to the GEN set for an instruction

Implementing LE/LSF/SE

- How to compute the loads/stores that are available?
- More dataflow analysis!
- Use the standard GEN and KILL strategy.
- OUT[P] = IN[P] - KILL[P] + GEN[P]

Going to cover this in more detail later in dominator-based global value numbering