

# **Alias Analysis and Load/Store Elimination**

**15-411/15-611 Compiler Design**

Ben L. Titzer and Seth Copen Goldstein

April 10, 2025

# Today

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

# Optimizations

- Optimizations covered so far
  - Constant propagation / folding
  - Copy propagation (SSA)
  - Dataflow optimization
  - Locality optimization (e.g. loop optimizations)
  - Loop invariant code motion
  - Lazy code motion
- What's left?
  - Optimizing memory accesses
  - Inlining
  - Control-flow optimizations
  - Instruction scheduling

# Optimizing Memory Accesses

- Dependence analysis and loop opts targeted arrays
  - Interchange and tiling for better locality
  - Generally assumed that arrays don't alias each other
- LICM to reduce work per iteration
- Lazy code motion (re)moves redundant expressions
- What about accesses to non-array memory?
  - Becomes tricky in the presence of pointers

```
struct color {  
    byte red;  
    byte green;  
    byte blue;  
};
```

```
int shade(color* c) {  
    c->red    = c->red * 2;  
    c->green  = c->red / 2;  
    c->blue   = c->red / 4;  
}
```

# Optimizing Memory Accesses

- Dependence analysis and loop opts targeted arrays
  - Interchange and tiling for better locality
  - Generally assumed that arrays don't alias each other
- LICM to reduce work per iteration
- Lazy code motion (re)moves redundant expressions
- What about accesses to non-array memory?
  - Becomes tricky in the presence of pointers

**Two redundant memory accesses!**

```
struct color {  
    byte red;  
    byte green;  
    byte blue;  
};
```

```
int shade(color* c) {  
    c->red    = c->red * 2;  
    c->green  = c->red / 2;  
    c->blue   = c->red / 4;  
}
```

# Optimizing Memory Accesses

- Dependence analysis and loop opts targeted arrays
  - Interchange and tiling for better locality
  - Generally assumed that arrays don't alias each other
- LICM to reduce work per iteration
- Lazy code motion (re)moves redundant expressions
- What about accesses to non-array memory?
  - **What if there were two different objects?**
  - Becomes tricky in the presence of pointers

```
struct color {  
    byte red;  
    byte green;  
    byte blue;  
};
```

```
int shade(color* c, color* d) {  
    c->red    = d->red * 2;  
    c->green  = d->red / 2;  
    c->blue   = d->red / 4;  
}
```

# Optimizing Memory Accesses

- Standard optimizations like constant/copy propagation work on local variables
- SSA renaming exposes explicit dataflow relationships
- Memory and pointers represent less explicit dataflow
- Alias analysis allows compiler to reason about potentially-overlapping memory accesses

```
struct color {  
    byte red;  
    byte green;  
    byte blue;  
};
```

```
int shade(color* c, color* d) {  
    c->red    = d->red * 2;  
    c->green  = d->red / 2;  
    c->blue   = d->red / 4;  
}
```

# Optimizing Memory Accesses

- Standard optimizations like constant/copy propagation work on local variables
- SSA renaming exposes explicit dataflow relationships
- Memory and pointers represent less explicit dataflow
- Alias analysis allows compiler to reason about potentially-overlapping memory accesses

How many redundant accesses are there now?

```
struct color {  
    byte red;  
    byte green;  
    byte blue;  
};
```

```
int shade(color* c, color* d) {  
    c->red    = d->red * 2;  
    c->green  = d->red / 2;  
    c->blue   = d->red / 4;  
}
```



# Optimizing Memory Accesses

- Abstract assembly has a complete mess of memory accesses

```
shade:
  x0 ← M[d]
  x1 ← x0 * 2;
  M[c] ← x1;
  x2 ← M[d]
  x3 ← x2 / 2;
  c2 ← c + 1;
  M[c2] ← x3;
  x4 ← M[d]
  x5 ← x4 / 4;
  c3 ← c + 2;
  M[c3] ← x5;
```

```
struct color {
    byte red;
    byte green;
    byte blue;
};
```

```
int shade(color* c, color* d) {
    c->red    = d->red * 2;
    c->green  = d->red / 2;
    c->blue   = d->red / 4;
}
```

# Optimizing Memory Accesses

- Abstract assembly has a complete mess of memory accesses
- Which of these are redundant loads?

?

```
shade:
  x0 ← M[d]
  x1 ← x0 * 2;
  M[c] ← x1;
  x2 ← M[d]
  x3 ← x2 / 2;
  c2 ← c + 1;
  M[c2] ← x3;
  x4 ← M[d]
  x5 ← x4 / 4;
  c3 ← c + 2;
  M[c3] ← x5;
```

```
struct color {
  byte red;
  byte green;
  byte blue;
};
```

```
int shade(color* c, color* d) {
  c->red    = d->red * 2;
  c->green  = d->red / 2;
  c->blue   = d->red / 4;
}
```

# Optimizing Memory Accesses

- Abstract assembly has a complete mess of memory accesses
- Which of these are redundant loads?
- Depends on if c and d are “aliases”.

```
shade:
  x0 ← M[d]
  x1 ← x0 * 2;
  M[c] ← x1;
  x2 ← M[d]
  x3 ← x2 / 2;
  c2 ← c + 1;
  M[c2] ← x3;
  x4 ← M[d]
  x5 ← x4 / 4;
  c3 ← c + 2;
  M[c3] ← x5;
```

?

```
struct color {
  byte red;
  byte green;
  byte blue;
};
```

```
int shade(color* c, color* d) {
  c->red    = d->red * 2;
  c->green  = d->red / 2;
  c->blue   = d->red / 4;
}
```

# Alias Analysis

- Alias analysis is how compilers reason about whether two memory locations may, must, or must not be the same at runtime.
- Results of alias analysis drive memory access optimizations.

```
shade:
  x0 ← M[d]
  x1 ← x0 * 2;
  M[c] ← x1;
  x2 ← M[d]
  x3 ← x2 / 2;
  c2 ← c + 1;
  M[c2] ← x3;
  x4 ← x2
  x5 ← x4 / 4;
  c3 ← c + 2;
  M[c3] ← x5;
```

```
struct color {
  byte red;
  byte green;
  byte blue;
};
```

```
int shade(color* c, color* d) {
  c->red    = d->red * 2;
  c->green  = d->red / 2;
  c->blue   = d->red / 4;
}
```

# Alias Analysis

Conservatively assuming c may-alias d

```
shade:
  x0 ← M[d]
  x1 ← x0 * 2;
  M[c] ← x1;
  x2 ← M[d];
  x3 ← x2 / 2;
  c2 ← c + 1;
  M[c2] ← x3;
  x4 ← x2;
  x5 ← x4 / 4;
  c3 ← c + 2;
  M[c3] ← x5;
```

```
struct color {
  byte red;
  byte green;
  byte blue;
};
```

```
int shade(color* c, color* d) {
  c->red    = d->red * 2;
  c->green  = d->red / 2;
  c->blue   = d->red / 4;
}
```

# Alias Analysis

Conservatively assuming c may-alias d

corresponding source-level caching

```
shade:
  x0 ← M[d]
  x1 ← x0 * 2;
  M[c] ← x1;
  x2 ← M[d];
  x3 ← x2 / 2;
  c2 ← c + 1;
  M[c2] ← x3;
  x4 ← x2;
  x5 ← x4 / 4;
  c3 ← c + 2;
  M[c3] ← x5;
```

```
struct color {
  byte red;
  byte green;
  byte blue;
};
```

```
int shade(color* c, color* d) {
  c->red    = d->red * 2;
  byte tmp  = d->red;
  c->green  = tmp / 2;
  c->blue   = tmp / 4;
}
```

# Alias Analysis

Assuming c must-not-alias d

```
shade:
  x0 ← M[d]
  x1 ← x0 * 2;
  M[c] ← x1;
  x2 ← x0
  x3 ← x2 / 2;
  c2 ← c + 1;
  M[c2] ← x3;
  x4 ← x2
  x5 ← x4 / 4;
  c3 ← c + 2;
  M[c3] ← x5;
```

```
struct color {
  byte red;
  byte green;
  byte blue;
};
```

```
int shade(color* c, color* d) {
  c->red    = d->red * 2;
  c->green  = d->red / 2;
  c->blue   = d->red / 4;
}
```

# Alias Analysis

Assuming c must-not-alias d

corresponding source-level caching

```
shade:
  x0 ← M[d]
  x1 ← x0 * 2;
  M[c] ← x1;
  x2 ← x0
  x3 ← x2 / 2;
  c2 ← c + 1;
  M[c2] ← x3;
  x4 ← x2
  x5 ← x4 / 4;
  c3 ← c + 2;
  M[c3] ← x5;
```

```
struct color {
  byte red;
  byte green;
  byte blue;
};
```

```
int shade(color* c, color* d) {
  byte tmp = d->red;
  c->red    = tmp * 2;
  c->green  = tmp / 2;
  c->blue   = tmp / 4;
}
```



# Aliasing Possibilities

- The possible aliasing relationships depends on the programming language.
- Java:
  - Only has references to objects and arrays
  - Objects and their fields are statically typed
  - No pointers to locals
  - Call-by-value

```
class Color {  
    byte red;  
    byte green;  
    byte blue;  
};
```

```
class Rectangle {  
    int width;  
    int height;  
}
```

# 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

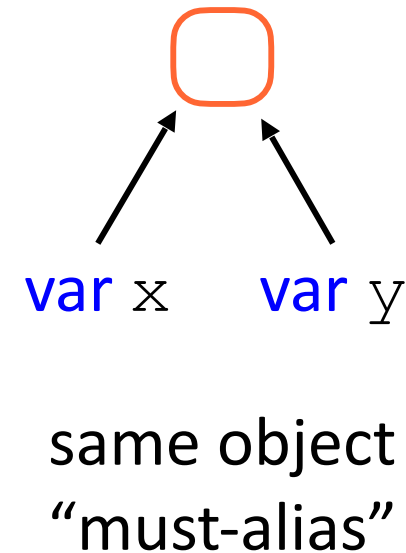
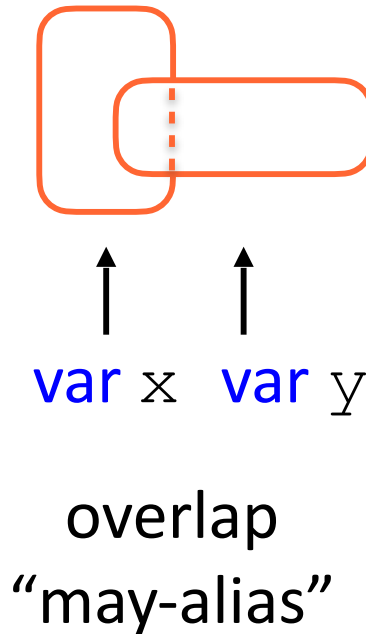
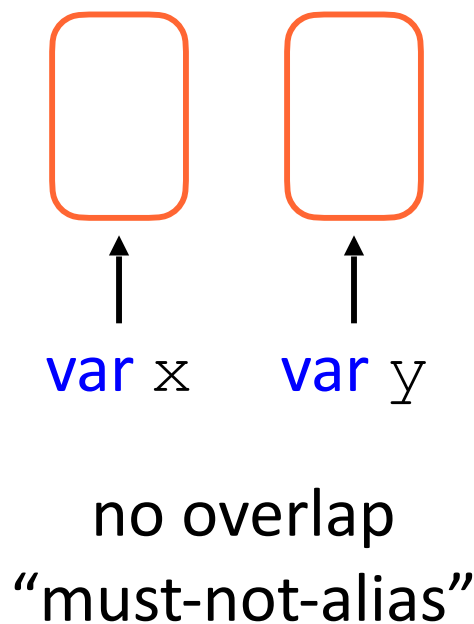
- C0
  - unions
  - pointers to structs and arrays
  - pointers to locals
  - pointers to fields
  - pointers to array elements
  - pointer arithmetic
  - type punning

# Aliasing Possibilities

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

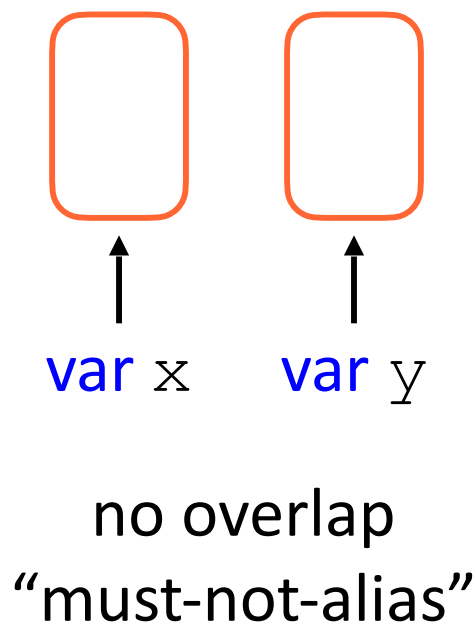
# Aliasing Relations

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

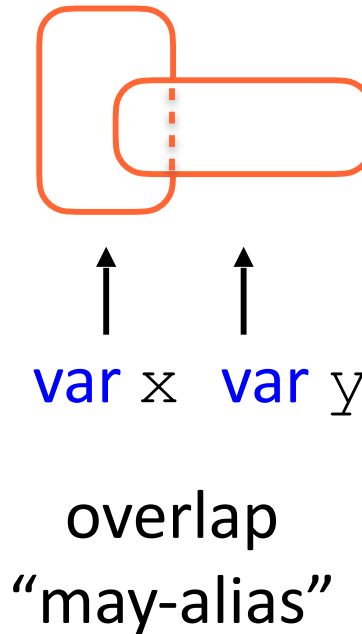


# Aliasing Relations

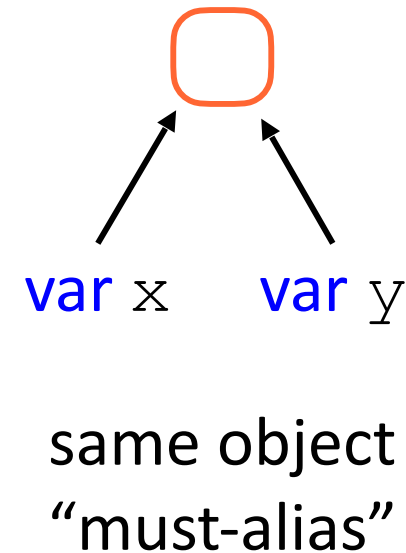
- Primarily interested in:
  - For any two pointers in the program, what set of objects could they point to?



reorder at will



treat conservatively

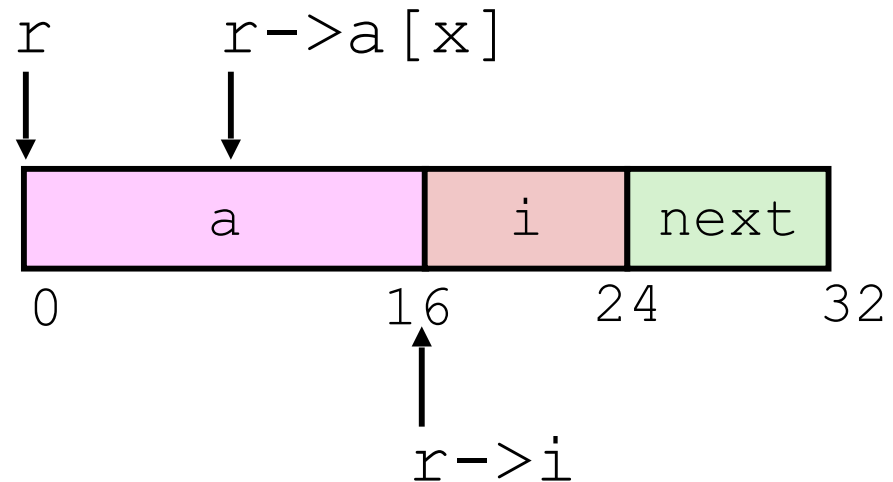


cache reads  
propagate writes

# 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;
```

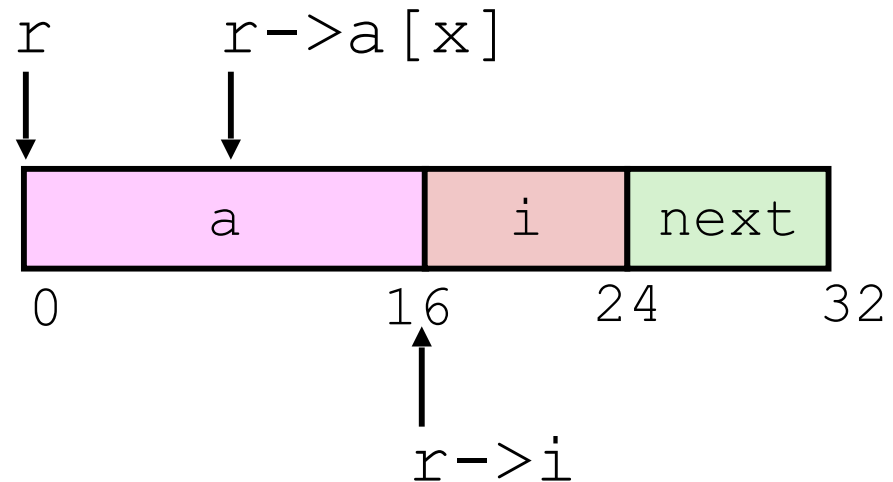




# Type-based Alias Analysis

- 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;
```

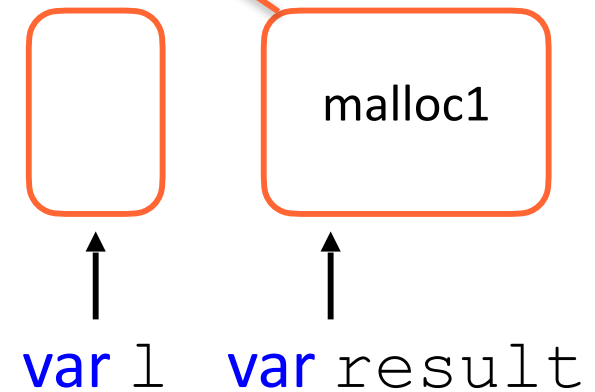


Most modern compilers  
make at least some use of types  
in alias analysis.

# 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;  
}
```



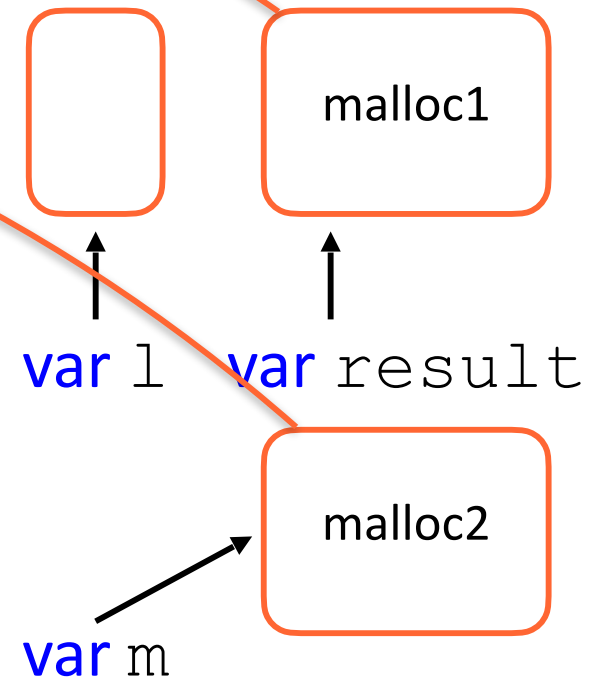
Most modern compilers  
make use of allocation sites  
for alias analysis.

# 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.

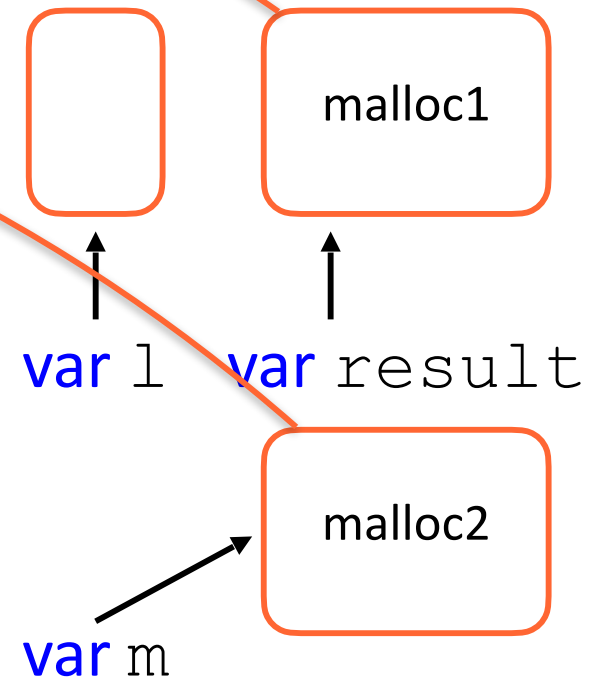


# Allocation Sites

- 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;  
}
```

Any issues with this?



# 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-Sensitive Pointer 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.
- Many compilers implement simpler forms of alias analysis which we cover near the end.
- More advanced forms of alias analysis are often used for program analysis and understanding (i.e. not optimization).

# What is Flow-Sensitivity?

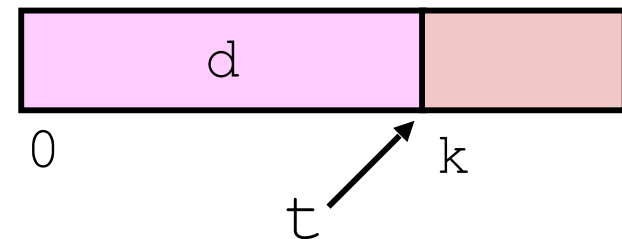
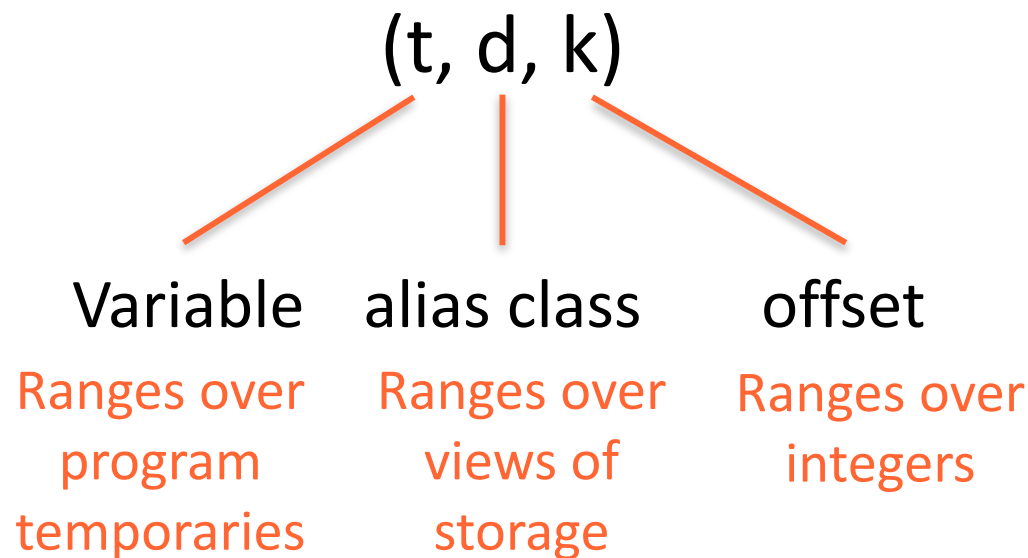
# Flow-Sensitivity

- 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).



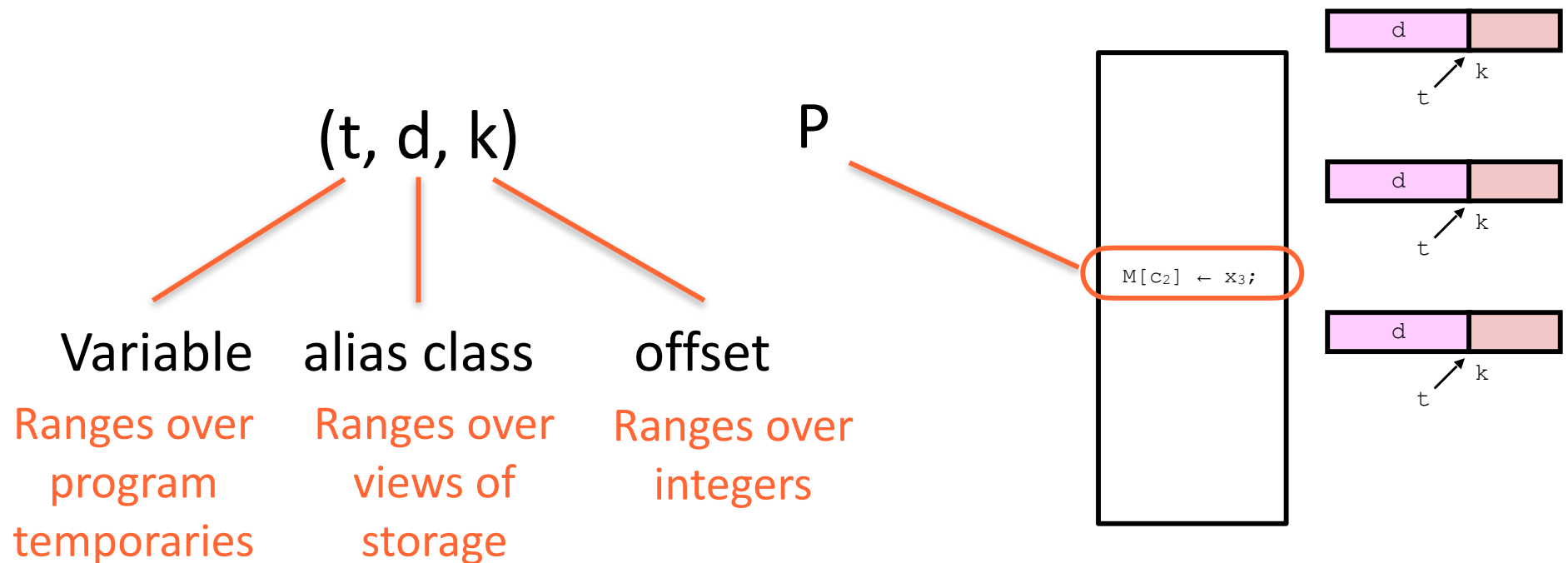
# Simple Flow-Sensitive Alias Analysis

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



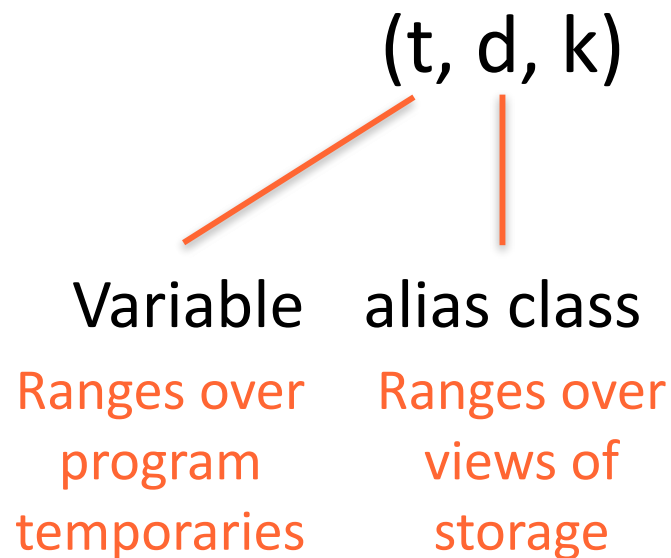
# Simple Flow-Sensitive Alias Analysis

- 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  $\tau$
- structs of type  $\tau$
- arrays of type  $\tau$
- struct fields  $S.f$
- globals of type  $\tau$

# 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.

$$\text{in}[\text{start}] = A$$

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

$$\text{out}[P] = \text{trans}_P(\text{in}[P])$$

# 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.

$$\text{in}[\text{start}] = A$$

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

$$\text{out}[P] = \text{trans}_P(\text{in}[P])$$

# 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.

$\text{in}[\text{start}] = A$

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

$\text{out}[P] = \text{trans}_P(\text{in}[P])$

Initial relation A

$(x, \text{struct } T, 0)$

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.

$\text{in}[\text{start}] = A$

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

$\text{out}[P] = \text{trans}_P(\text{in}[P])$

predecessor relations  $\text{in}[q]$

$(x, \text{struct } T, 0)$

x points to

`void compute(struct T* x) {`

`y = x->f;`

`}`

# 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.

$\text{in}[\text{start}] = A$

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

$\text{out}[P] = \text{trans}_P(\text{in}[P])$

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

```
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(...)
```

trans(A)

$$(A - A(t)) \cup A(b \mapsto t)$$

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

Assume  $A(b \mapsto t)$   
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

```
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(...)
```

trans(A)

$$(A - A(t)) \cup \{(t, d, i) \mid (b, d, i - k) \in A\}$$

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

```
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(...)
```

trans(A)

$(A - A(t))$   
∪  
unknown(t)

# Transfer Functions

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

Statement

```
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(...)
```

trans(A)

$(A - A(t))$

U

unknown(t)

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

$(t, \text{struct } S, 0)$

# Transfer Functions

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

Statement

```
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(...)
```

trans(A)

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

```
t ← b
t ← b + k
t ← b ⊕ c
t ← M[b]
M[a] ← b
if a > b goto L
goto L
f(a ... )
t ← alloc99(...)
```

trans(A)

$(A - A(t))$   
 $\cup$   
 $(t, \text{alloc}_{99}, 0)$

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:

$p$  may-alias  $q$  at  $P$

$\Leftrightarrow$

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

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  $P$

$\Leftrightarrow$

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

Assuming we have additional  
overlap relation



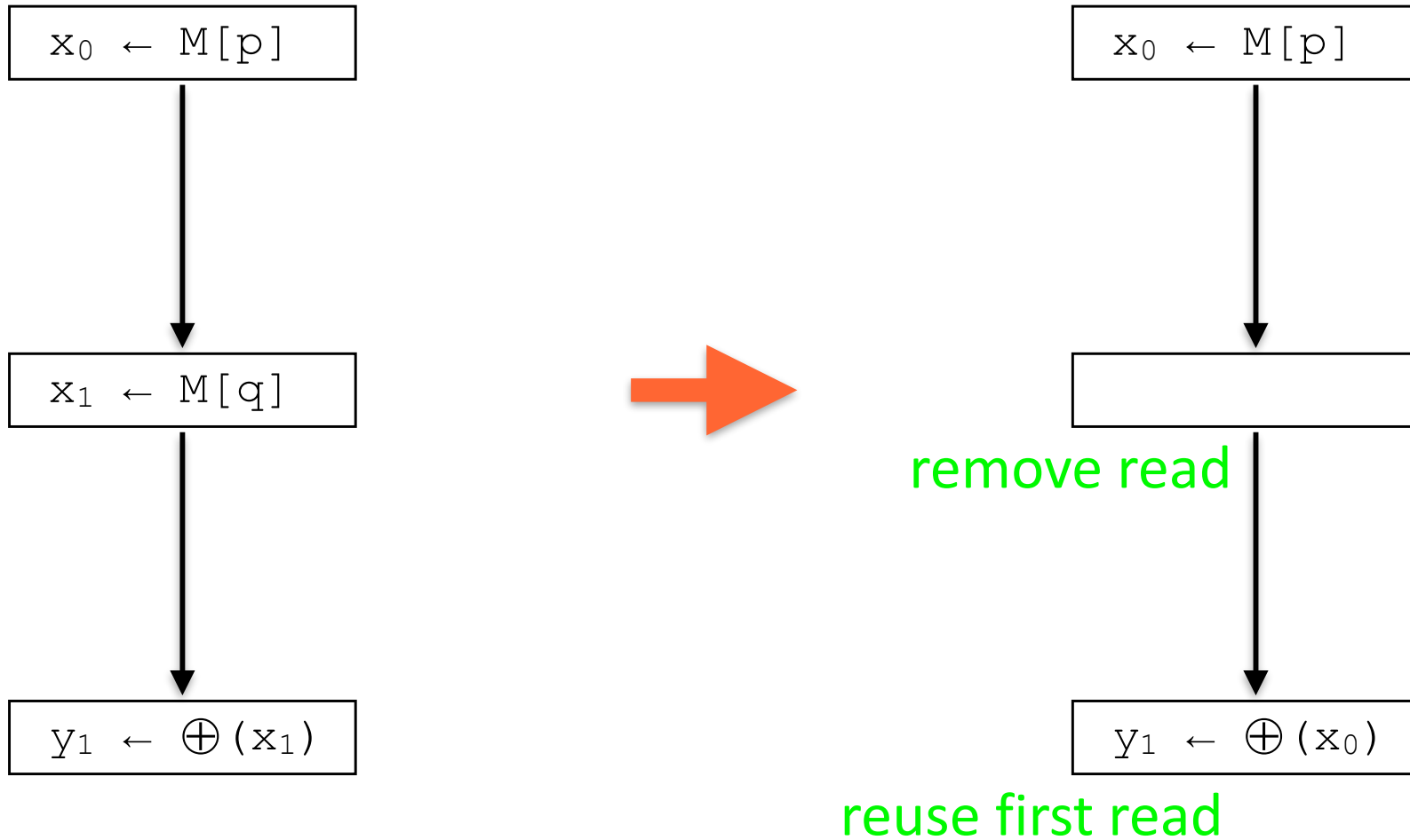
# The Heap Triple Crown

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

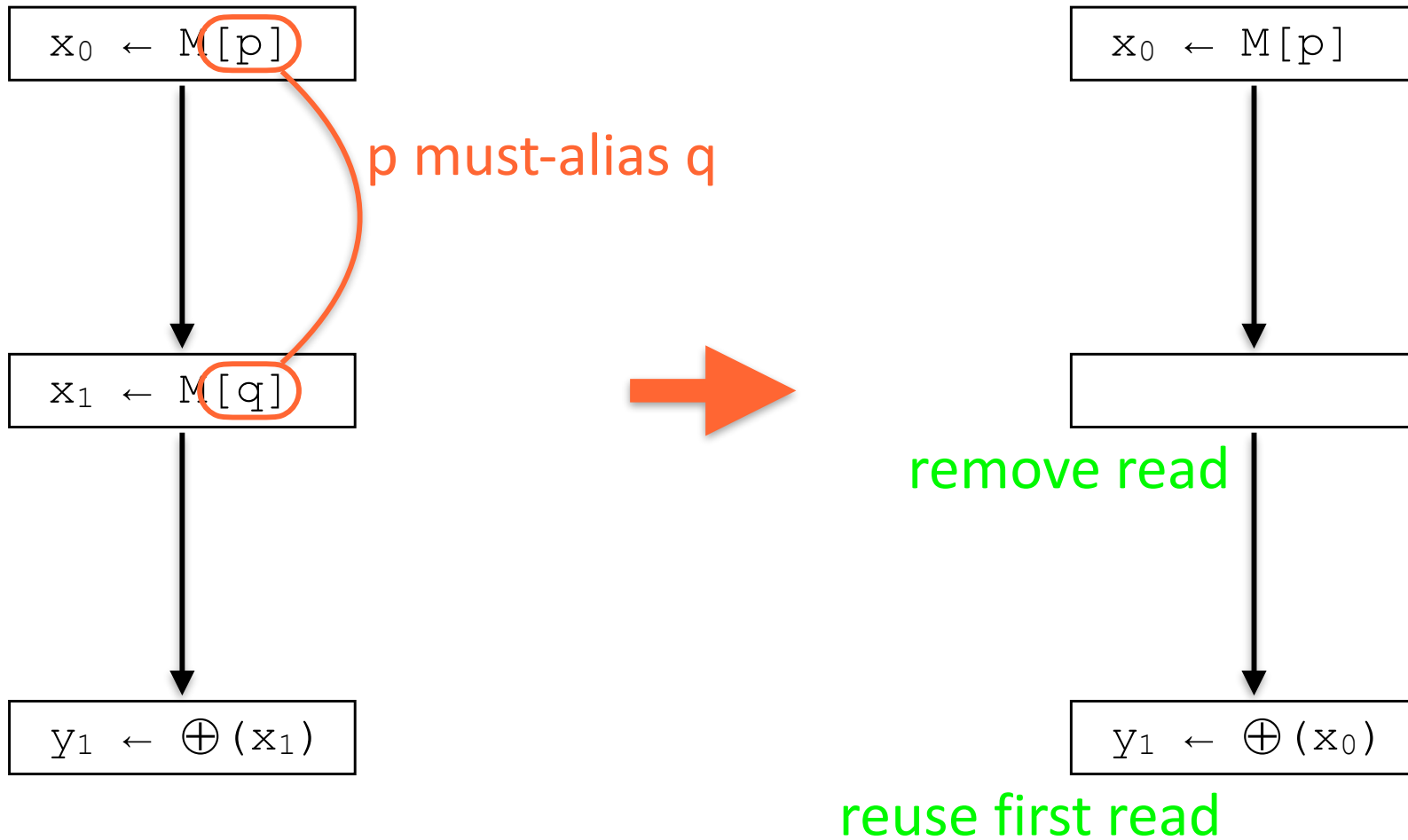
# Load Elimination

- 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.

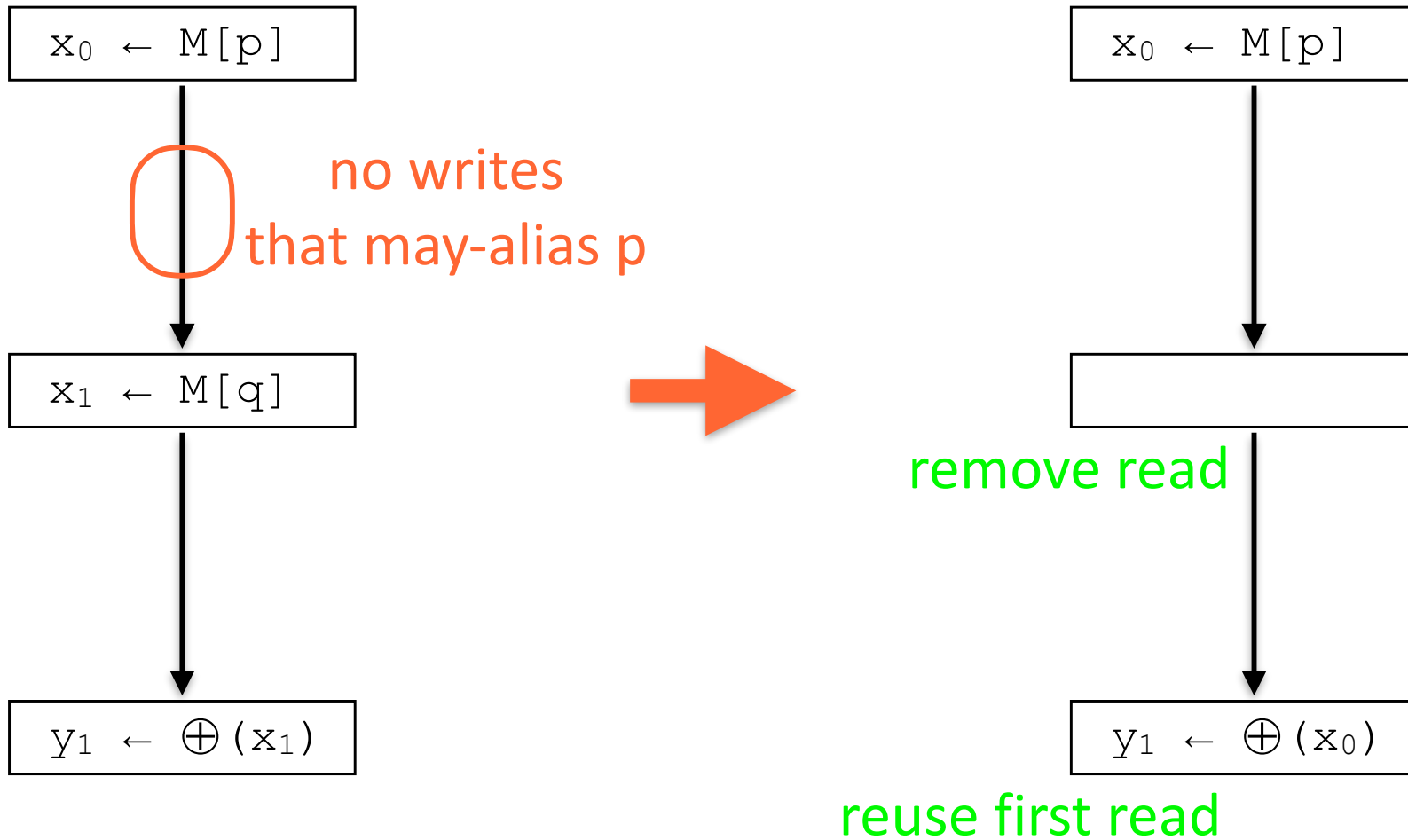
# Load Elimination Illustration



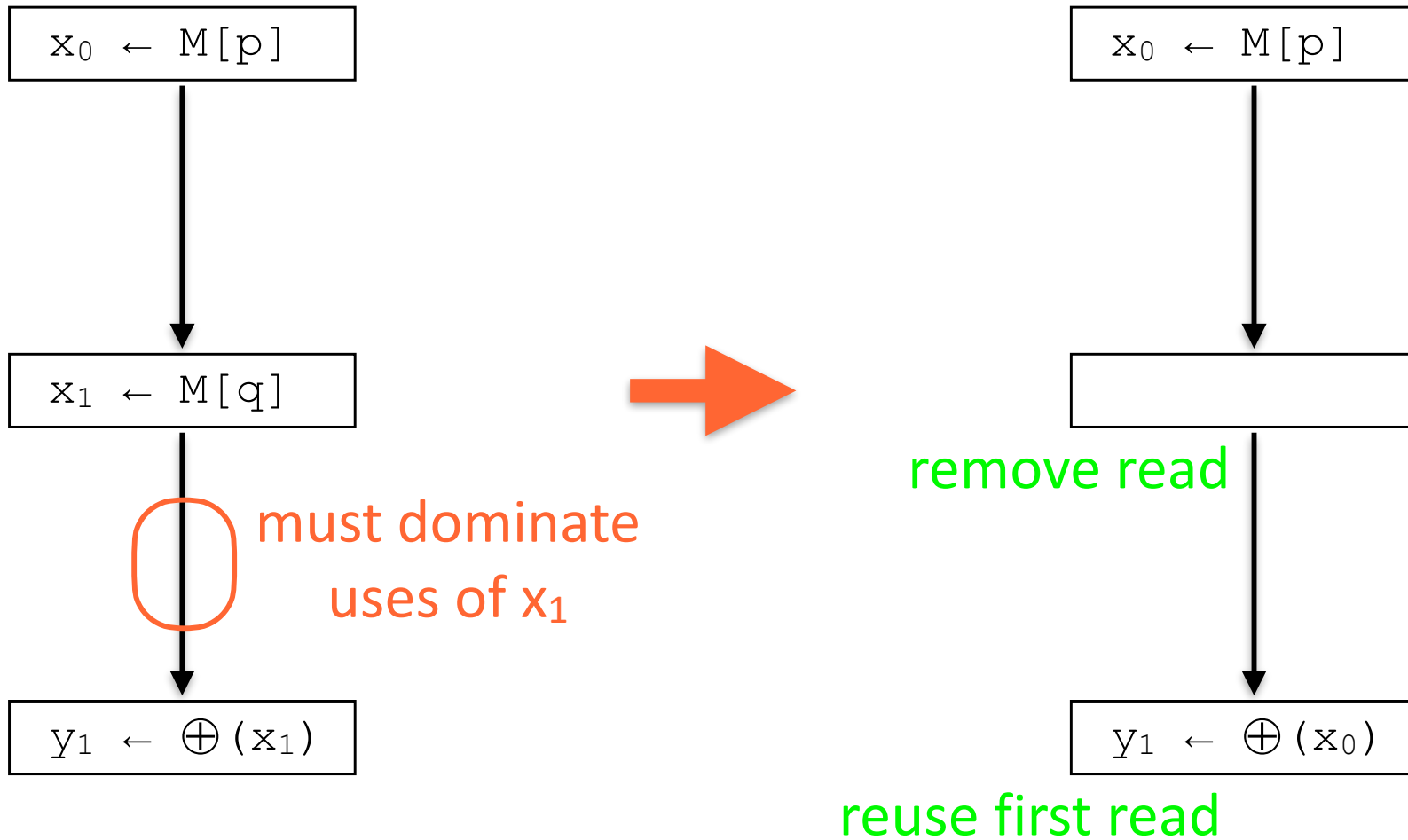
# Load Elimination Illustration



# Load Elimination Illustration



# Load Elimination Illustration



# Load Elimination Illustration

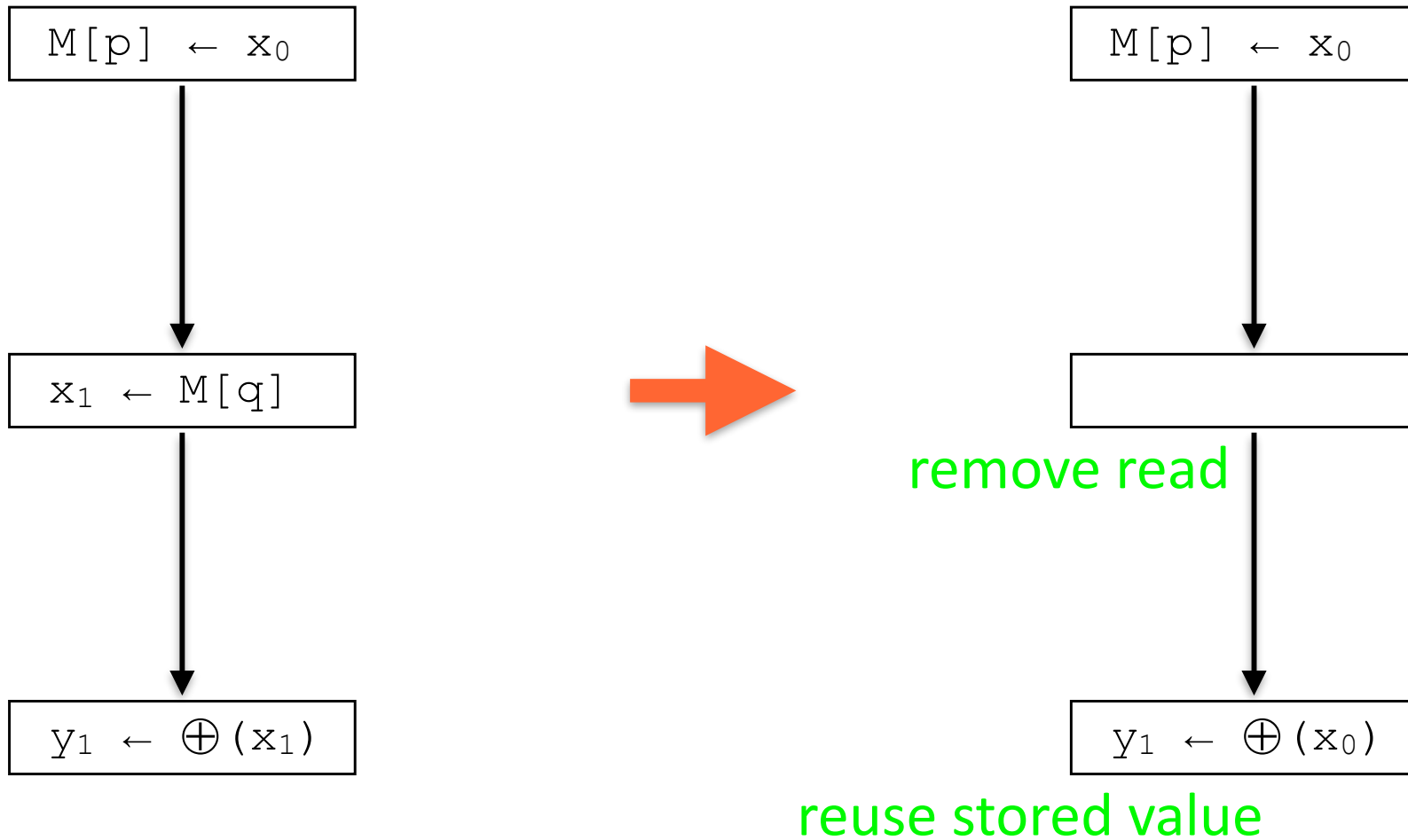


# Load-Store Forwarding

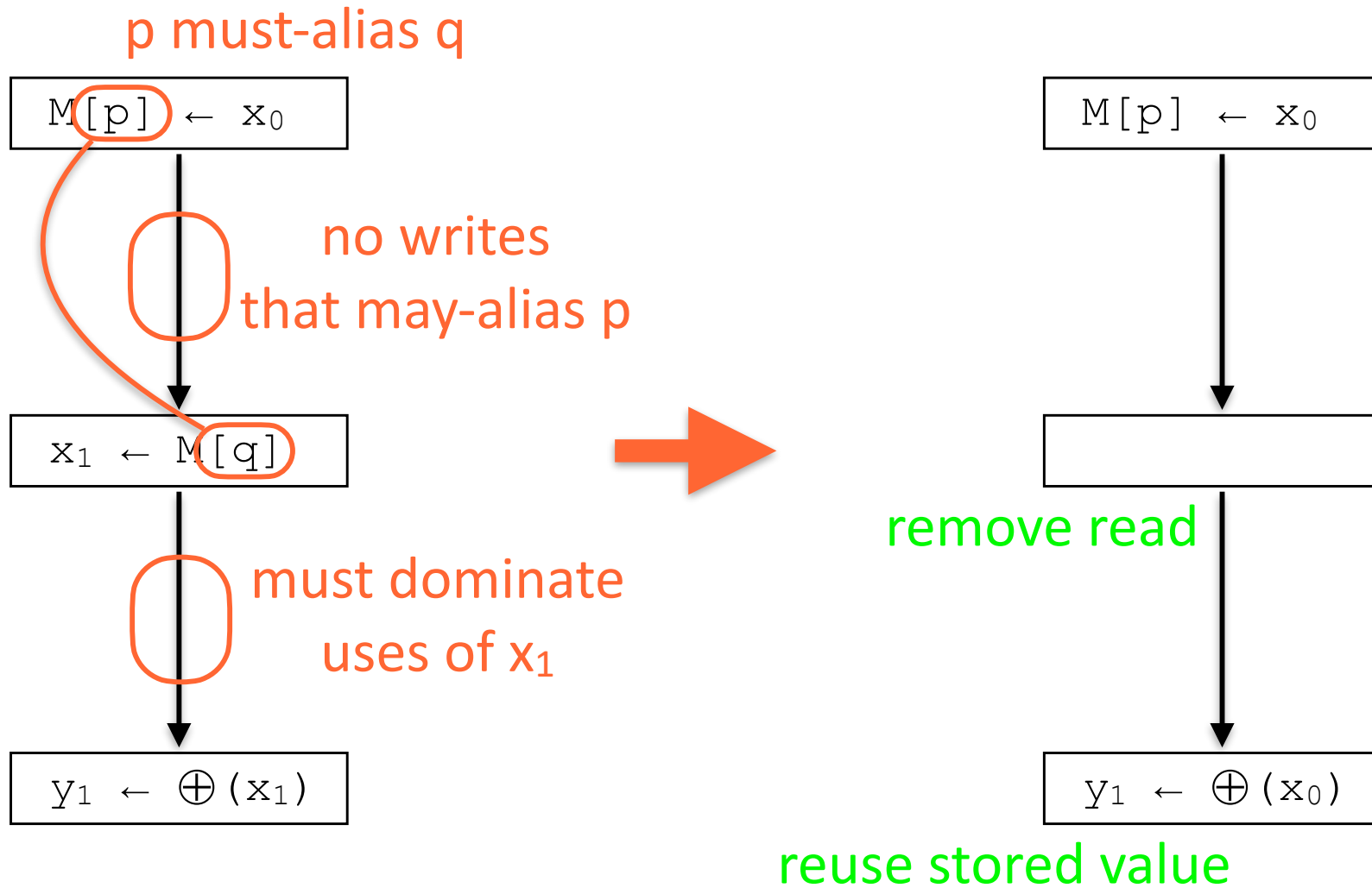
- Many programs store to memory and then immediately load the value back.
- Reusing a previously-stored value (safely, after alias analysis) is a slightly different form of common subexpression elimination.
- Can be done together or in a separate (lightweight) pass.
- Can be done locally or globally.



# Load-Store Forwarding Illustration



# Load-Store Forwarding Illustration



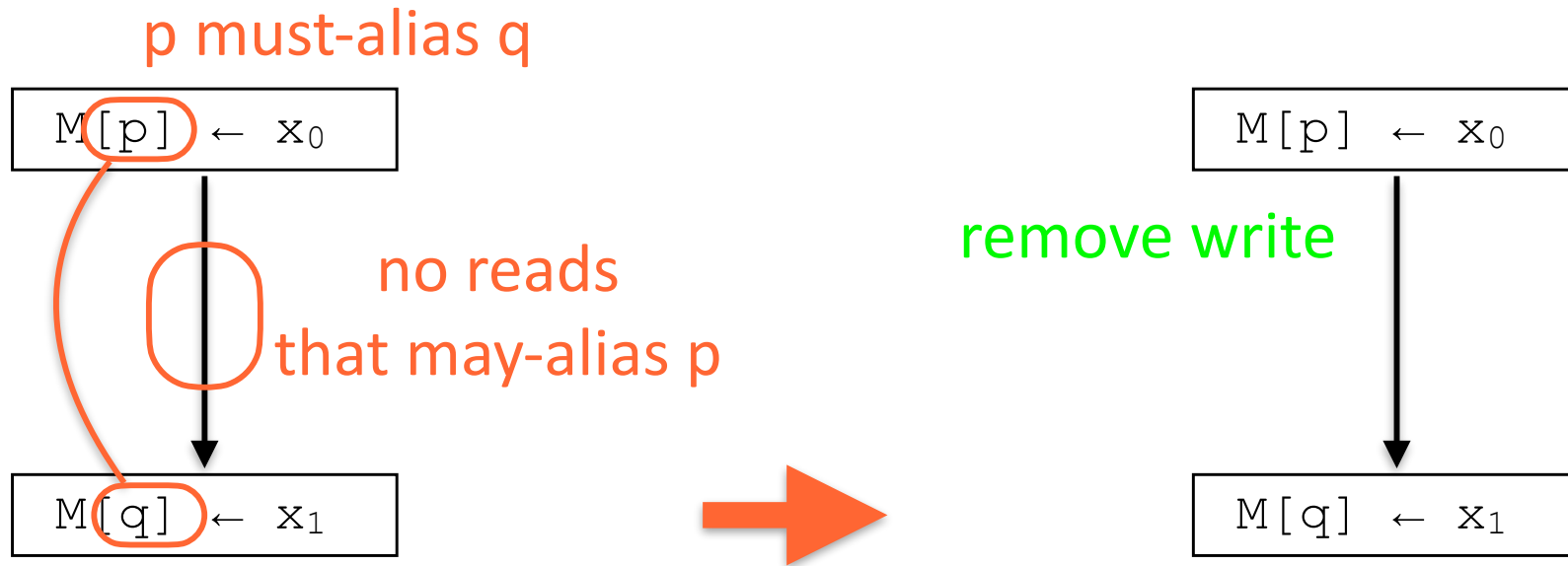
# Store Elimination

- 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 when we  
discuss dominator-  
based global value  
numbering

# Lame Alias Analysis

- (examples on board)