Control-flow sensitive escape analysis in Falcon JIT

Artur Pilipenko,
apilipenko@azul.com
Azul Systems

Agenda

- Introductions
- CaptureTracking analysis
- Falcon’s FlowSensitiveEA analysis
- FlowSensitiveEA transforms
- Performance results
- Conclusion

Agenda

- Introductions
- CaptureTracking analysis
- Falcon’s FlowSensitiveEA analysis
- FlowSensitiveEA transforms
- Performance results
- Conclusion

What is Falcon?

- JIT compiler for Java based on LLVM
  - Java bytecode => native
  - Inside of a running JVM
- Final tier compiler in Azul's Zing JVM
  - Compiles only the hottest methods
  - Focus on performance

What is Falcon?

If you want to learn more

- LLVM Dev Meeting 15 - LLVM for a managed language: what we've learned
  https://llvm.org/devmtg/2015-10/#talk14
- LLVM Dev Meeting 17 - Falcon: An optimizing Java JIT
  https://llvm.org/devmtg/2017-10/#talk12
- EuroLLVM 17 - Expressing high level optimizations within LLVM
  http://llvm.org/devmtg/2017-03//2017/02/20/accepted-sessions.html#10
- EuroLLVM 18 - New PM: taming a custom pipeline of Falcon JIT
  https://llvm.org/devmtg/2018-04/talks.html#Talk_13

What is escape analysis?

- Pointer analysis to determine dynamic scope of pointers & objects
- Whether an object or a pointer is accessible outside the scope of the current function or thread?
- This information enables various optimizations
  - E.g. a lock can be eliminated if the lock object is not accessible outside of one thread

Escape analysis for Java

Why is it important?

- Java doesn't have value types other than builtin primitive types
- Any record-like type is heap allocated by default
- As a result, idiomatic Java code has a lot of short lived allocations
- These allocations often don’t escape the thread or the method
  - This opens opportunities for optimizations!

Escape analysis for Java

Typical applications

- Optimize storage for unescaped allocations
  - Scalar replacement, e.g. [1]
  - Stack allocation, e.g. [2]
- Downgrade of thread safe operations
  - Lock elision [1, 2], atomics, etc

[1] "Escape analysis in the context of dynamic compilation and deoptimization." (Kotzmann, Mössenböck 2005)
[2] "Stack allocation and synchronization optimizations for Java using escape analysis." (Choi, Gupta, et al. 2003)

Escape related facts

What do we need for different optimizations?

- Different optimizations need different facts
- For example:
  - Constant fold comparisons involving new allocation — can the pointer be inspected outside of the function?
  - Optimize allocation storage — can the contents of the object be inspected outside of the function?
  - Downgrade atomics — can the contents of the object be inspected outside of the thread?

Pointer value can’t be inspected outside of the function scope

=>

Contents of the object can’t be inspected outside of the function scope

=>

Contents of the object can’t be inspected outside of the thread

Pointer value can’t be inspected outside of the function scope

=>

Contents of the object can’t be inspected outside of the function scope

=>

Contents of the object can’t be inspected outside of the thread

Compute the stronger fact and assume weaker facts from it

Pointer value can’t be inspected outside of the function scope

We will call this property “no escape” or “no capture”

Agenda

- Introductions
- CaptureTracking analysis
- Falcon’s FlowSensitiveEA analysis
- FlowSensitiveEA transforms
- Performance results
- Conclusion

CaptureTracking analysis in LLVM

Can bits of the pointer be inspected outside of the function scope?

```
bool llvm::PointerMayBeCaptured(const Value *V,
                                bool ReturnCaptures,
                                bool StoreCaptures,
                                unsigned MaxUsesToExplore)
```

CaptureTracking analysis in LLVM

How does it work?

- Analyze uses of the pointer
- Each use either
  - Captures — e.g. pointer is stored into a global
  - Doesn’t capture — e.g. pointer is passed as an nocapture argument
  - Produces an alias — need to analyze uses of the alias as well
    - E.g. getelementptr, bitcast, addrspacecast

Users of CaptureTracking in LLVM

- Used either via BasicAliasAnalysis
  - GVN, EarlyCSE, LICM, DSE, etc
- Or directly
  - LICM, InstSimplify, ThreadSanitizer, etc
- Often used as a conservative approximation of weaker facts

EA optimizations in Falcon

- Initial implementation of EA-based optimizations used CaptureTracking
- Identified a few limitations
    - Handling of unescaped object graphs
    - Limited control-flow sensitivity
    - Compile time impact
- Eventually had to build our own analysis

What is missing in CaptureTracking?

Handling of unescaped graphs

- CaptureTracking considers any store as capture
- In fact a store to unescaped memory doesn’t escape or capture
- This is an unused StoreCapture parameter and >10 year old TODO

```
a = new A
b = new B
; Doesn’t capture!
a.field = b
; Can be eliminated!
monitor_enter(b)
b.value = 5
; Can be eliminated!
monitor_exit(b)
```

What is missing in CaptureTracking?

Handling of unescaped graphs

- Can work around some cases by iterative optimizations
- E.g. scalarize leaf allocation a first

```
b = new B
; Not a store anymore!
a_field = b
; Can be eliminated!
monitor_enter(b)
b.value = 5
; Can be eliminated!
monitor_exit(b)
```

What is missing in CaptureTracking?

Handling of unescaped graphs

- Doesn’t work if there are cycles in unescaped object graphs
  - Doubly-linked list kind of structures
- Unfortunately, appears in the standard library in Java :(

```
a = new A
b = new B
; Doesn’t capture!
a.field = b
; Doesn’t capture!
b.field = a
; Can be eliminated!
monitor_enter(b)
b.value = 5
; Can be eliminated!
monitor_exit(b)
```

What is missing in CaptureTracking?

Limited control-flow sensitivity

- Even if the allocation escapes we want to optimize the code before escape
  - E.g. thread safe initialization before escape,
  - or slow-path escapes
- CaptureTracking has limited control flow sensitivity
  - Prune uses which are not relevant for the given context in the function
  - Conservatively using DominatorTree and isPotentiallyReachableFrom
  - Often too conservative

What is missing in CaptureTracking?

Compile time impact

- CaptureTracking is a non-caching analysis
  - Scanning allocation uses on every query
- As a mitigation has a cutoff on the maximum number of uses to scan
  - 20 by default
- We have seen unescaped allocations with thousands of uses

Agenda

- Introductions
- CaptureTracking analysis
- Falcon’s FlowSensitiveEA analysis
- FlowSensitiveEA transforms
- Performance results
- Conclusion

Falcon’s FlowSensitiveEA

- Flow-sensitive analysis which models points-to graph of unescaped object by abstract interpretation
  - Tracked state is points-to graph of unescaped allocations
  - Traverse CFG in reverse-post order
  - Scan through instructions modeling their effects on the tracked state
  - Similar to [1] but intentionally separate analysis and transformations

[1] “Partial escape analysis and scalar replacement for Java" (Stadler, Würthinger, Mössenböck 2014)

Falcon’s FlowSensitiveEA

- Downstream analysis and transformations
  - Relies on some of the downstream concepts
  - Potentially can be upstreamed with some work

State tracking

Tracked allocations

Keep track of allocations which haven't yet escaped

```
; empty state
a = new A
; alloc: %a, type=A
b = new B
; alloc: %a, type=A
; alloc: %b, type=B
escape(a)
; alloc: %b, type=B
escape(b)
; empty state
```

State tracking

Tracked allocations

Keep track of allocations which haven't yet escaped

```
; empty state
a = new A
; alloc: %a, type=A
b = new B
; alloc: %a, type=A
; alloc: %b, type=B
escape(a)
; alloc: %b, type=B
escape(b)
; empty state
```

State tracking

Tracked allocations

Keep track of allocations which haven't yet escaped

```
; empty state
a = new A
; alloc: %a, type=A
b = new B
; alloc: %a, type=A
; alloc: %b, type=B
escape(a)
; alloc: %b, type=B
escape(b)
; empty state
```

State tracking

Tracked pointers

- Keep track of all pointers to tracked allocations
    - Including derived pointers

```
a = new A
; alloc: %a, type=A
a.8 = getelementptr a, 8
; alloc: %a, type=A
; alias: %a.8 - %a +8
a.8.i32 = bitcast a.8 to i32*
; alloc: %a, type=A
; alias: %a.8 - %a +8
; alias: %a.8.i32 - %a +8
```

State tracking

Tracked pointers

- Keep track of all pointers to tracked allocations
  - Including derived pointers

```
a = new A
; alloc: %a, type=A
a.8 = getelementptr a, 8
; alloc: %a, type=A
; alias: %a.8 - %a +8
a.8.i32 = bitcast a.8 to i32*
; alloc: %a, type=A
; alias: %a.8 - %a +8
; alias: %a.8.i32 - %a +8
```

State tracking

Points-to graph

- Tracked pointers can be stored in unescaped objects
- Need to track these pointers
- For example:
    - Object can escape if the holder object escapes

```
a = new A
; alloc: %a, type=A
b = new B
; alloc: %a, type=A
; alloc: %b, type=B
a.field = b ; b doesn't escape
; alloc: %a, type=A
; field = %b
; alloc: %b, type=B
escape(a);
; both a and b escaped
```

State tracking

Points-to graph

- Tracked pointers can be stored in unescaped objects
- Need to track these pointers
- For example:
  - Object can escape if the holder object escapes

```
a = new A
; alloc: %a, type=A
b = new B
; alloc: %a, type=A
; alloc: %b, type=B
a.field = b ; b doesn't escape
; alloc: %a, type=A
; field = %b
; alloc: %b, type=B
escape(a);
; both a and b escaped
```

State tracking

Points-to graph

- Tracked pointers can be stored in unescaped objects
- Need to track these pointers
- For example:
  - Load from an unescaped object might be an alias to another allocation

```
a = new A
; alloc: %a, type=A
b = new B
; alloc: %a, type=A
; alloc: %b, type=B
a.field = b
; alloc: %a, type=A
; field = %b
; alloc: %b, type=B
b' = a.field
; alloc: %a, type=A
; field = %b
; alloc: %b, type=B
; alias: %b'
```

State tracking

Points-to graph

- Tracked pointers can be stored in unescaped objects
- Need to track these pointers
- For example:
    - Load from an unescaped object might be an alias to another allocation

```
a = new A
; alloc: %a, type=A
b = new B
; alloc: %a, type=A
; alloc: %b, type=B
a.field = b
; alloc: %a, type=A
; field = %b
; alloc: %b, type=B
b' = a.field
; alloc: %a, type=A
; field = %b
; alloc: %b, type=B
; alias: %b'
```

State tracking

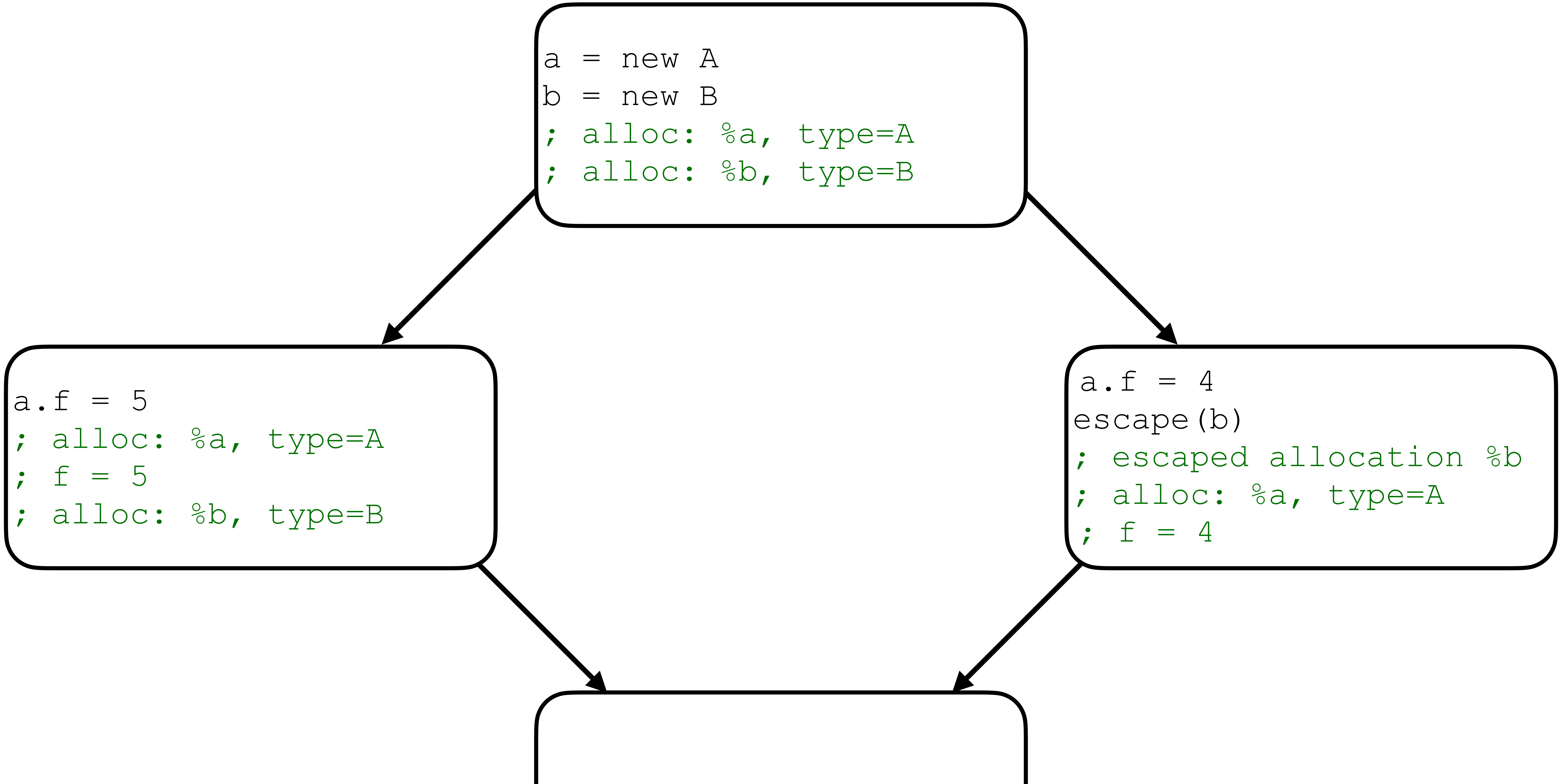
Allocation state

For escape analysis we only need pointer fields, but our implementation tracks all fields

```
a = new A
; alloc: %a, type=A
a.field = b
; alloc: %a, type=A
; field = %b
a.int = 5
; alloc: %a, type=A
; field = %b
; int = 5
```

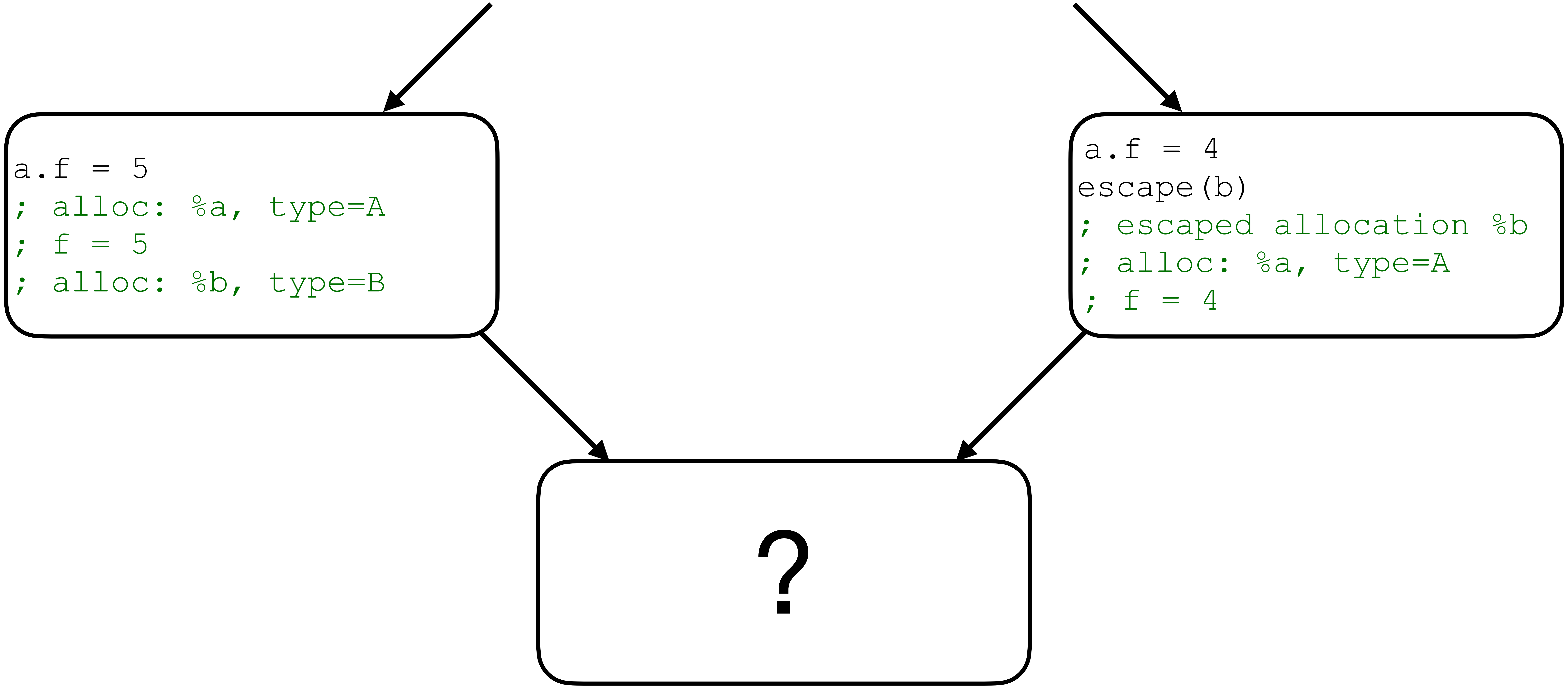
Example

Compute block out states



Example

Merge incoming states



Merge incoming states

Take an intersection of tracked allocations across all incoming paths

$$MergedState.TrackedAllocations = \bigcup_{S \in IncomingStates} S.TrackedAllocations$$

If there is a path where an allocation escaped — the allocation is escaped in the merge state as well

Merge incoming states

For every allocation in the intersection

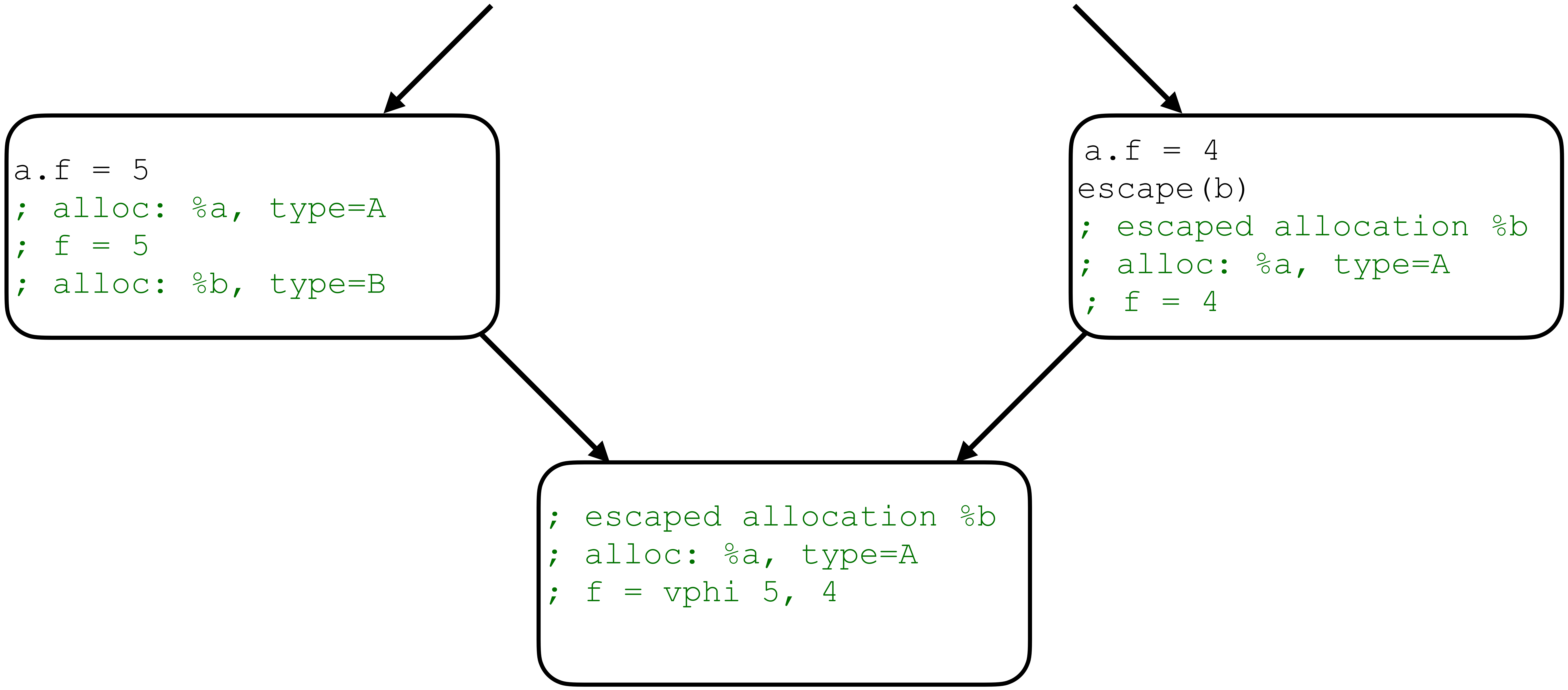
- Compute tracked pointers

$$MergedState.TrackedPointers = \bigcap_{S \in IncomingStates} S.TrackedPointers$$

- Produce merged allocation state
  - For every field in the allocation produce a value describing merged field value
  - If different values come from different paths produce a (virtual) PHI value
  - Don’t materialize PHINodes in the IR during analysis

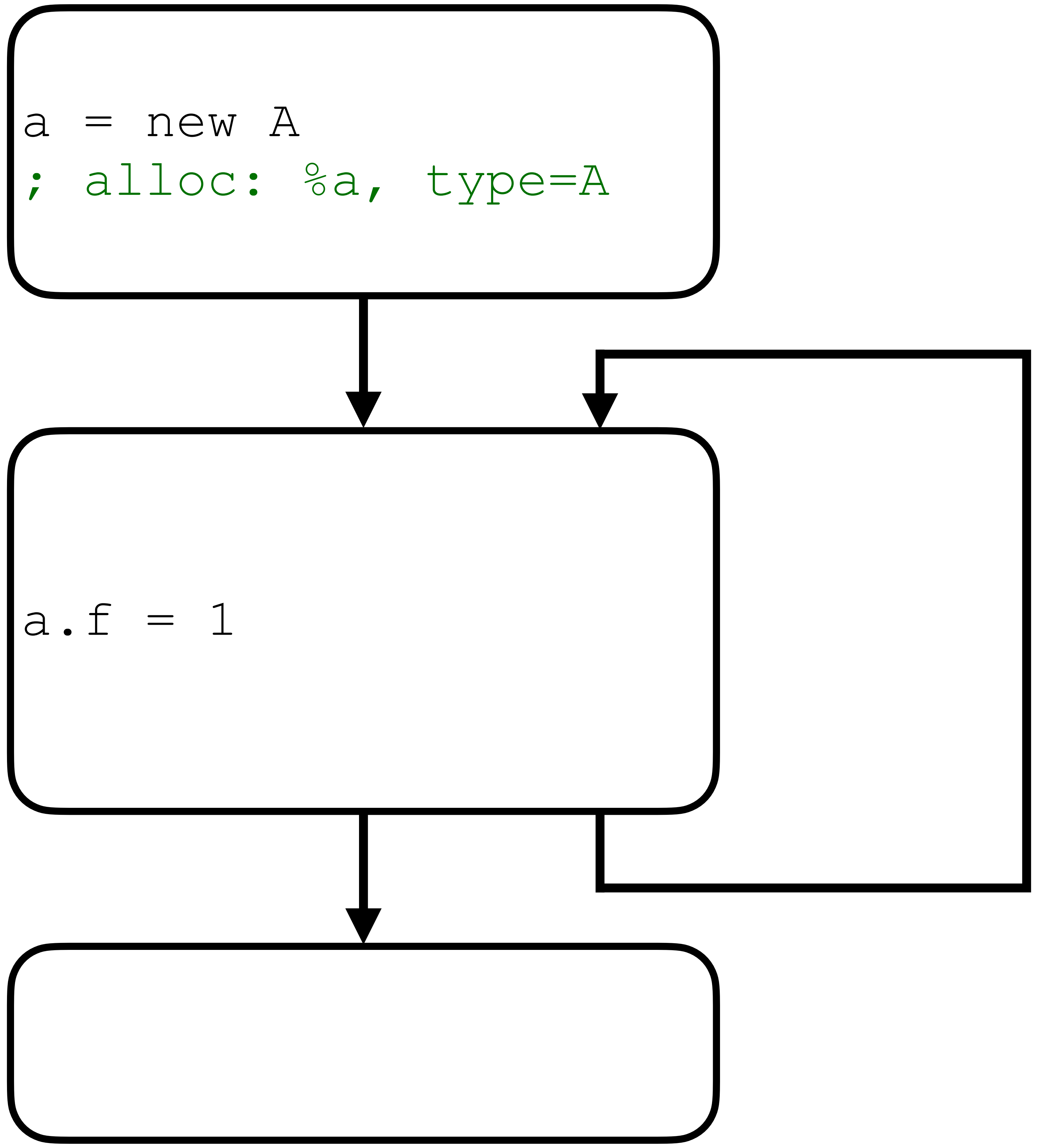
Example

Merge incoming states



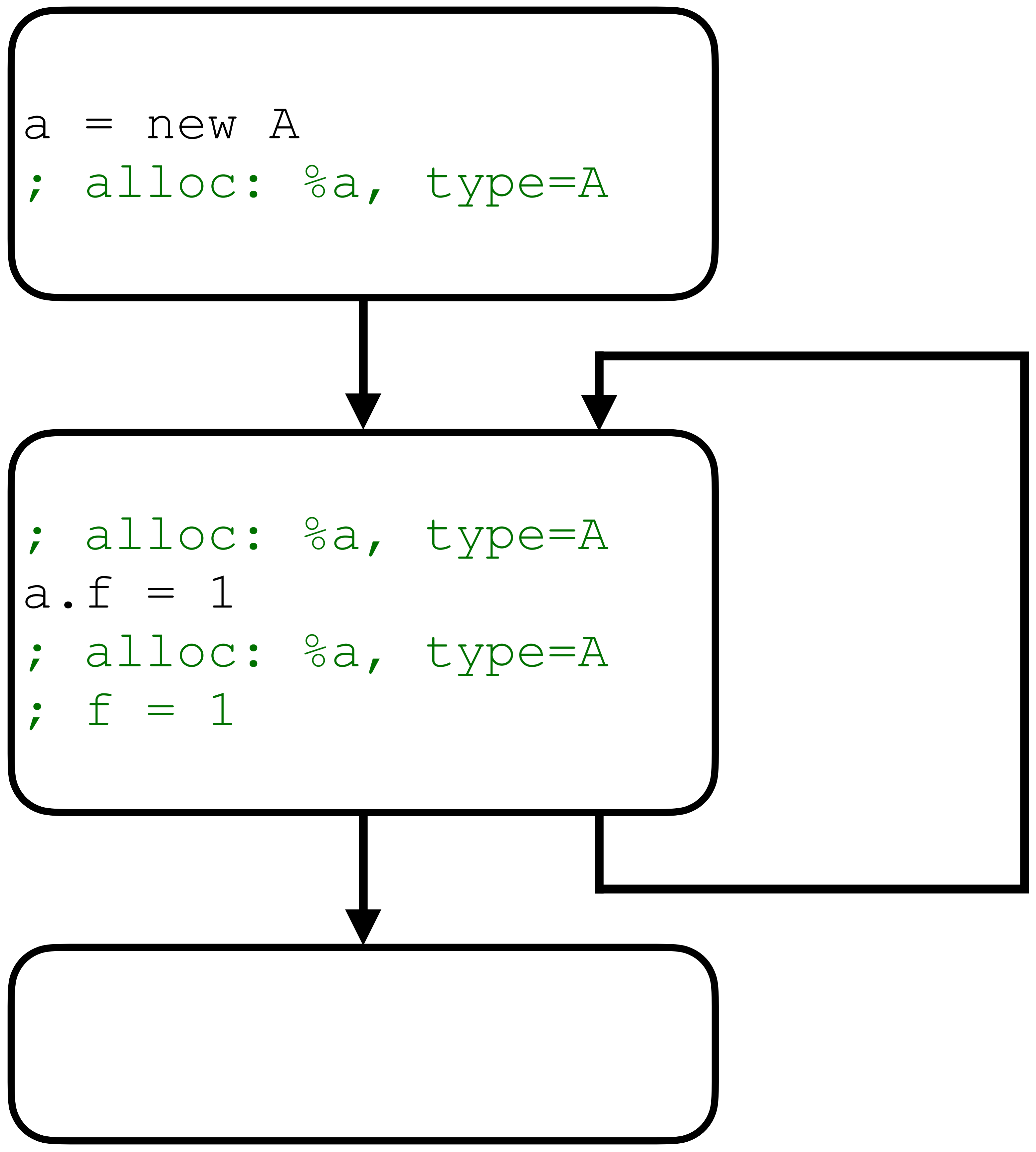
Handling CFG cycles

- If there is a cycle the back edge state will be unknown
- Perform optimistic merge
  - Assume the back edge doesn't affect the merged state
  - Once the back edge state is available re-evaluate the merge
- The tracked state is supposed to be a lattice, so the iteration eventually converges



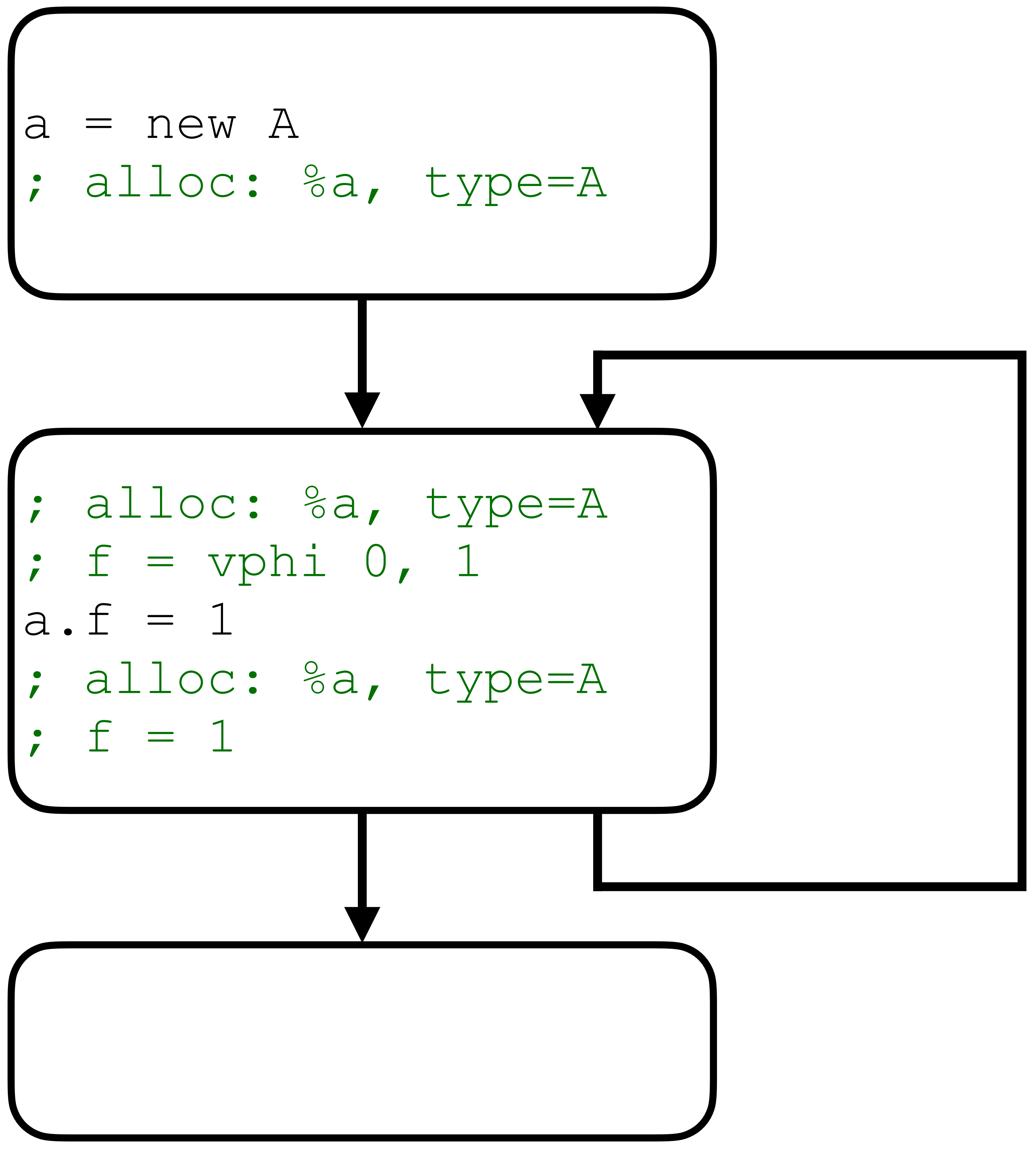
Handling CFG cycles

- If there is a cycle the back edge state will be unknown
- Perform optimistic merge
  - Assume the back edge doesn't affect the merged state
  - Once the back edge state is available re-evaluate the merge
- The tracked state is supposed to be a lattice, so the iteration eventually converges



Handling CFG cycles

- If there is a cycle the back edge state will be unknown
- Perform optimistic merge
    - Assume the back edge doesn't affect the merged state
    - Once the back edge state is available re-evaluate the merge
- The tracked state is supposed to be a lattice, so the iteration eventually converges



Agenda

- Introductions
- CaptureTracking analysis
- Falcon’s FlowSensitiveEA analysis
- FlowSensitiveEA transforms
- Performance results
- Conclusion

Analysis invalidation/update

- Analysis maintains non-trivial state
  - Allocations with all of their fields
- Currently doesn’t support updates as the IR is transformed
  - Usually it’s hard to get it right

Analysis invalidation/update

- Instead we collect the transformations based on EA and then apply
    1. Build EA
    2. Collect transforms
    3. Discard EA
    4. Apply transforms
- Only care about update/invalidation of individual transforms
    - ValueHandles do the job

FlowSensitiveEA users

- Currently is organized as a single pass which does various transforms using the analysis
- Scalar replacement as a series of transforms like
  - Store-load forwarding for unescapes objects
  - Constant folding of comparisons
  - Dematerialization in deopt states
- Downgrade of thread safe operations - e.g. locks/atomics
- Dead store elimination for unescaped objects

Integrate with AliasAnalysis

- We have ad-hoc transforms for unescapes allocations
  - Store-load forwarding, dead store elimination, etc
- LLVM already has these optimizations, we just need to feed the results of the analysis to the existing transforms
  - It’s hard because we need to solve update/invalidation problem

Agenda

- Introductions
- CaptureTracking analysis
- Falcon’s FlowSensitiveEA analysis
- FlowSensitiveEA transforms
  - Scalar replacement example
- Performance results
- Conclusion

Scalar replacement

- If an allocation doesn’t escape we want to
  - Scalarize its fields
  - Eliminate the allocation

```
a = new A
a.f = 5
b = foo()
x = a.f
if (a == b) ...
```

Scalar replacement

Rewrite allocation uses

- Store-load forwarding to scalarize the fields

```
a = new A
a.f = 5
b = foo()
; alloc: %a, type=A
; f = 5
x = a.f
if (a == b) ...
```

Scalar replacement

Rewrite allocation uses

- Store-load forwarding to scalarize the fields

```
a = new A
a.f = 5
b = foo()
x = 5
if (a == b) ...
```

Scalar replacement

Rewrite allocation uses

- Store-load forwarding to scalarize the fields
- Note: this can also be done by EarlyCSE/GVN
- But they don’t benefit from flow-sensitive EA facts, so are less powerful

```
a = new A
a.f = 5
b = foo()
x = 5
if (a == b) ...
```

Scalar replacement

Rewrite allocation uses

- Constant fold comparisons of unescapes pointers

```
a = new A
a.f = 5
b = foo()
x = 5
; alloc: %a, type=A
; f = 5
if (a == b) ...
```

Scalar replacement

Rewrite allocation uses

- Constant fold comparisons of unescapes pointers

```
a = new A
a.f = 5
b = foo()
x = 5
if (false) ...
```

Scalar replacement

Rewrite allocation uses

- Constant fold comparisons of unescapes pointers
- Note: this can also be done by InstSimplify
- But again, it doesn't have access to EA facts

```
a = new A
a.f = 5
b = foo()
x = 5
if (false) ...
```

Scalar replacement

Rewrite allocation uses

Are we done yet?

```
a = new A
a.f = 5
b = foo()
x = 5
if (false) ...
```

Scalar replacement

Rewrite allocation uses

Deopt bundle use prevents elimination of the allocation!

```
a = new A
a.f = 5
b = foo() [ deopt(a) ]
x = 5
if (false) ...
```

Deoptimizations

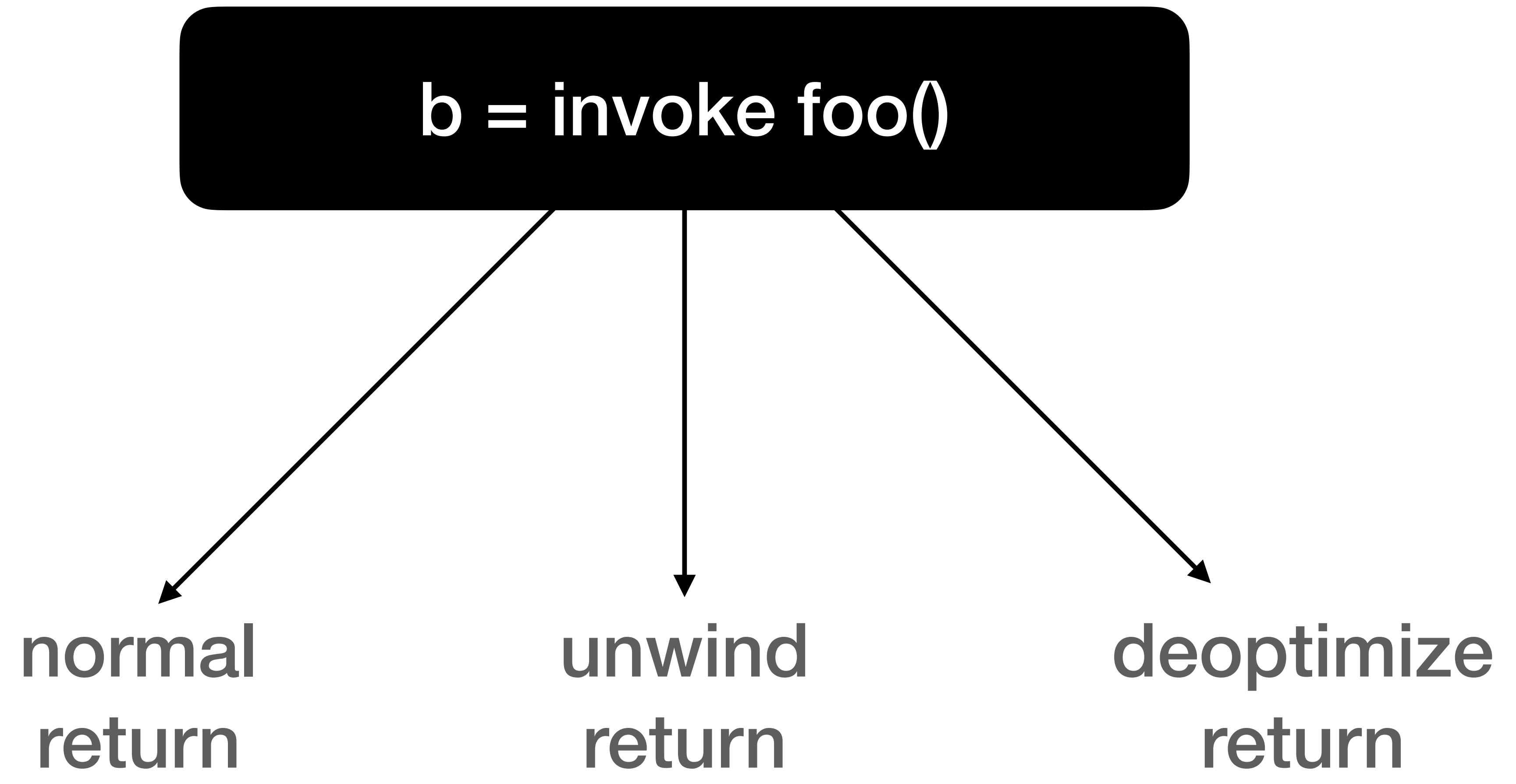
Side note

- Falcon uses speculative assumptions about the world to optimize the code
  - E.g. constant fold a load from a global field assuming it will never change
- We rely on runtime to check and invalidate the assumptions
- If any of the assumptions is invalidated the compiled code is no longer correct and should be *deoptimized*
  - If we are currently executing the code the execution is resumed in the interpreter

Deoptimizations

Side note

- Any call can invalidate some of the speculative assumptions of the caller
- In this case we can't resume execution of the compiler code on return
- Instead jump to runtime to deoptimize and resume execution in the interpreter

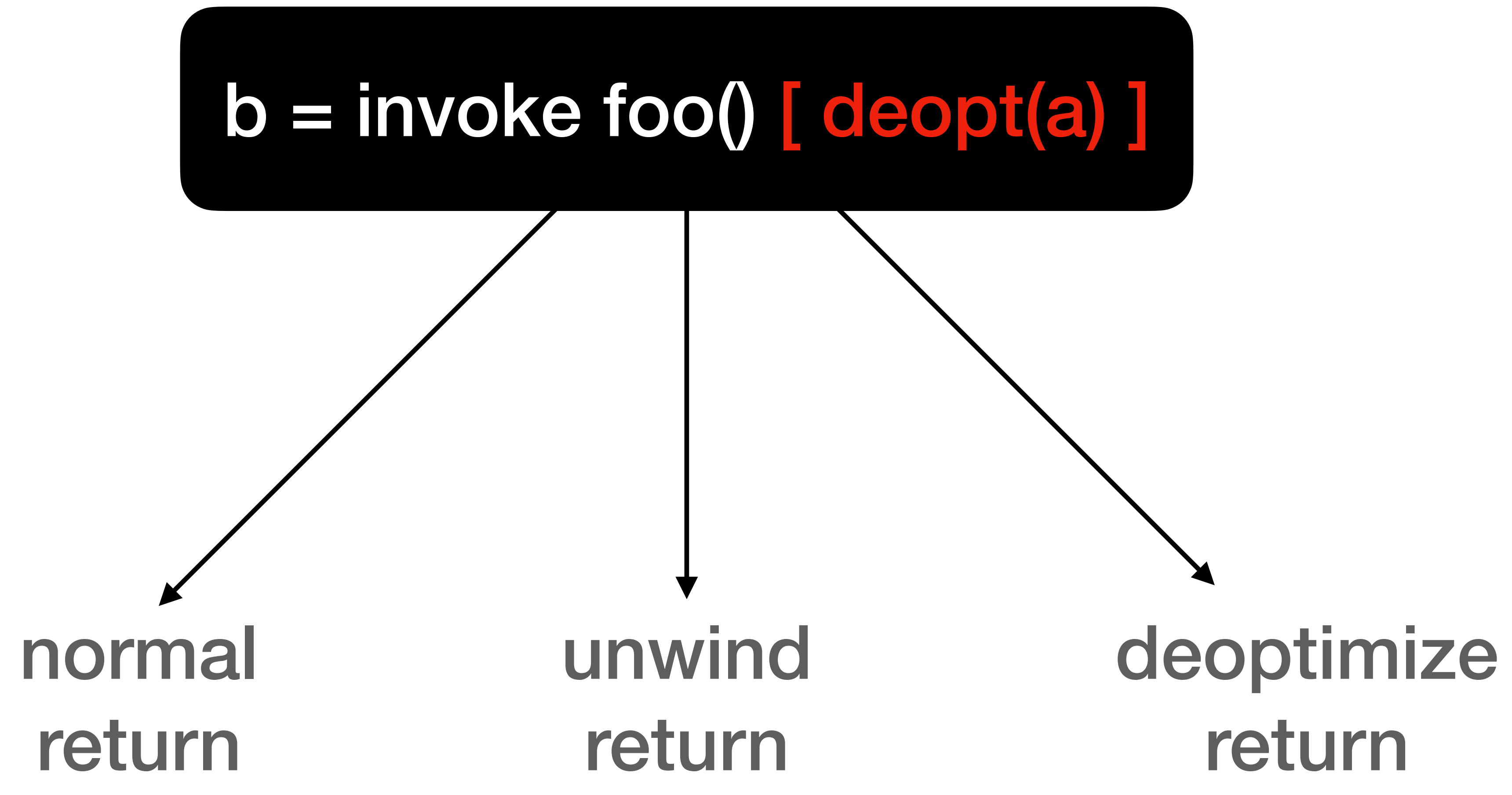


Deoptimizations

Side note

- Deopt state contains the values describing the abstract state to resume execution from
  - Interpreter expression stack, locals, etc.
- Only used if deoptimization occurs

=> doesn’t caputre/escape



Scalar replacement

Dematerialization

- Replace the allocation value with symbolic description on how to materialize the same allocation on deopt path [1]
- Effectively sinking the allocation into deoptimization path

```
a = new A
a.f = 5
b = foo() [ deopt(a) ]
x = 5
if (false) ...
```

[1] "Run-time support for optimizations based on escape analysis." (Kotzmann, Mössenböck 2007)

Scalar replacement

Dematerialization

- Use allocation state to produce symbolic descrption
- We know the exact state of the allocation, i.e. we know values for all fields

```
a = new A
a.f = 5
; alloc: %a, type=A
; f = 5
b = foo() [ deopt(a) ]
x = 5
```

Scalar replacement

Dematerialization

- Use allocation state to produce symbolic descrption
- We know the exact state of the allocation, i.e. we know values for all fields

```
a = new A
a.f = 5
; alloc: %a, type=A
; f = 5
b = foo() [
  lazy_object #1 {new A(), f=5},
  deopt(#1) ]
x = 5
```

Scalar replacement

Eliminate unused allocations

- Now the allocation becomes removable
- Has only initializing uses

```
a = new A
a.f = 5
b = foo() [
  lazy_object #1 {new A(), f=5},
  deopt(#1) ]
x = 5
```

Scalar replacement

Eliminate unused allocations

- Now the allocation becomes removable
- Has only initializing uses

```
b = foo() [
  lazy_object #1 {new A(), f=5},
  deopt(#1) ]
x = 5
```

Agenda

- Introductions
- CaptureTracking analysis
- Falcon’s FlowSensitiveEA analysis
- FlowSensitiveEA transforms
  - EA-driven loop unroll example
- Performance results
- Conclusion

EA-driven loop unroll

- Newly allocated unescaped linked-list-like structure
- While loop iterating over the structure
- This loop is non-analyzable!

```
node3 = new ListNode()
node3.f = 3
node3.next = null
node2 = new ListNode()
node2.f = 2
node2.next = node3
node1 = new ListNode()
node1.f = 1
node1.next = node2

summ = 0
current = node3
while (current != null) {
  summ += current.f
  current = current.next
}
```

EA-driven loop unroll

- FlowSensitiveEA effectively models the object graph for this structure
- This model can be used to rewrite the loop
- And make it analyzable/unrollable

```
; alloc: %node3, type=ListNode
; next = %node2
; alloc: %node2, type=ListNode
; next = %node1
; alloc: %node1, type=ListNode
; next = null

summ = 0
current = node3
while (current != null) {
  summ += current
  current = current.next
}
```

```
; alloc: %node3, type=ListNode
; next = %node2
; alloc: %node2, type=ListNode
; next = %node1
; alloc: %node1, type=ListNode
; next = null

loop:
  %curr = phi [%node3, %incoming], [%next, %backedge]
  %cont = icmp eq, %curr, null
  br %cont, %exit, %cont

cont:
  ...
  %next = load %curr.next
  br %loop

exit:
```

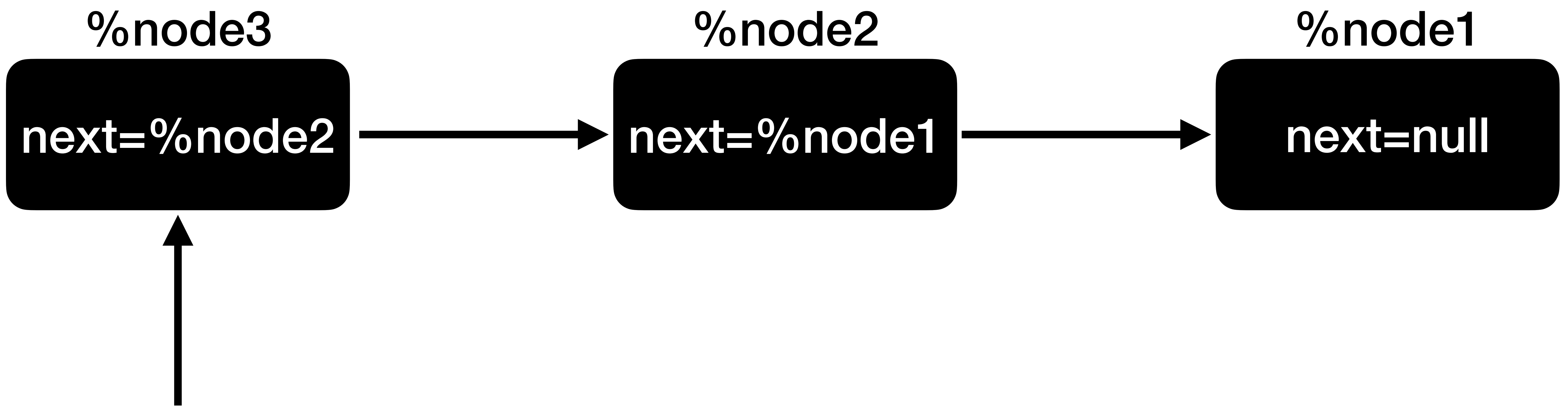
```
; alloc: %node3, type=ListNode
; next = %node2
; alloc: %node2, type=ListNode
; next = %node1
; alloc: %node1, type=ListNode
; next = null

loop:
  %curr = phi [%node3, %incoming], [%next, %backedge]
  %cont = icmp eq, %curr, null
  br %cont, %exit, %cont

cont:
  ...
  %next = load %curr.next
  br %loop

exit:
```

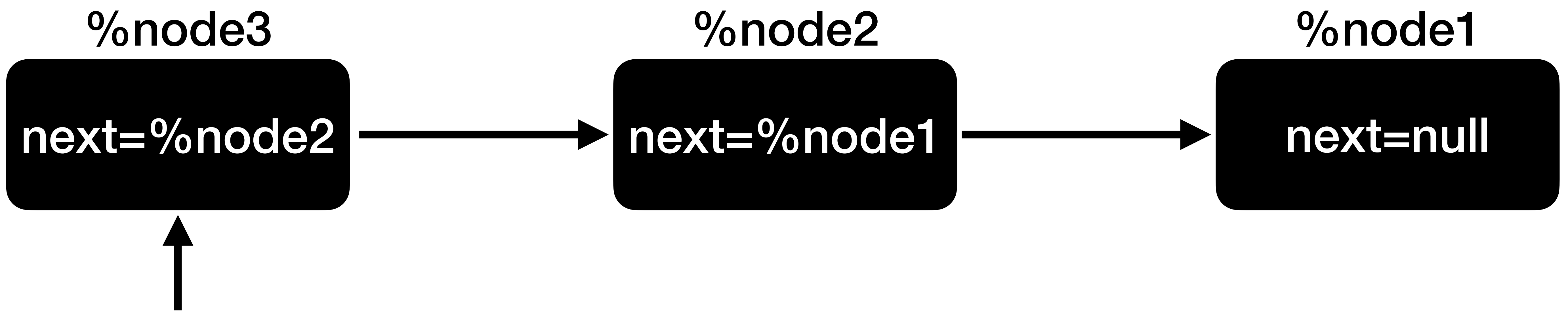
Identify iteration over linked-list-like structure




```
loop:
  %curr = phi [%node3, %incoming], [%next, %backedge]
  %cont = icmp eq, %curr, null
  br %cont, %exit, %cont

cont:
  ...
  %next = load %curr.next
  br %loop

exit:
```




```
loop:
  %curr = phi [%node3, %incoming], [%next, %backedge]
  %canonical.iv = phi [0, %incoming], [%iv.next, %backedge]
  %cont = icmp eq, %canonical.iv, 3
  br %cont, %exit, %cont

cont:
  ...
  %iv.next = add %canonical.iv, 1
  %next = load %curr.next
  br %loop

exit:
```

Insert canonical IV

Rewrite the exit in term of the canonical IV

Now the loop exit is analyzable!

EA-driven loop unroll

- The loop is now analyzable and unrollable

```
node3 = new ListNode()
node3.f = 3
node3.next = null
node2 = new ListNode()
node2.f = 2
node2.next = node3
node1 = new ListNode()
node1.f = 1
node1.next = node2
summ = 0
; Unrolled loop
summ += node3.f
summ += node2.f
summ += node1.f
```

EA-driven loop unroll

- The loop is now analyzable and unrollable
- After unrolling store-load forwarding kicks in

```
node3 = new ListNode()
node3.f = 3
node3.next = null
node2 = new ListNode()
node2.f = 2
node2.next = node3
node1 = new ListNode()
node1.f = 1
node1.next = node2
summ = 0
; Unrolled loop
summ += node3.f
summ += node2.f
summ += node1.f
```

EA-driven loop unroll

- The loop is now analyzable and unrollable
- After unrolling store-load forwarding kicks in

```
node3 = new ListNode()
node3.f = 3
node3.next = null
node2 = new ListNode()
node2.f = 2
node2.next = node3
node1 = new ListNode()
node1.f = 1
node1.next = node2
summ = 0
; Unrolled loop
summ += 3
summ += 2
summ += 1
```

EA-driven loop unroll

- The loop is now analyzable and unrollable
- After unrolling store-load forwarding kicks in
- The allocations become removable

```
node3 = new ListNode()
node3.f = 3
node3.next = null
node2 = new ListNode()
node2.f = 2
node2.next = node3
node1 = new ListNode()
node1.f = 1
node1.next = node2
summ = 6
```

EA-driven loop unroll

- The loop is now analyzable and unrollable
- After unrolling store-load forwarding kicks in
- The allocations become removable

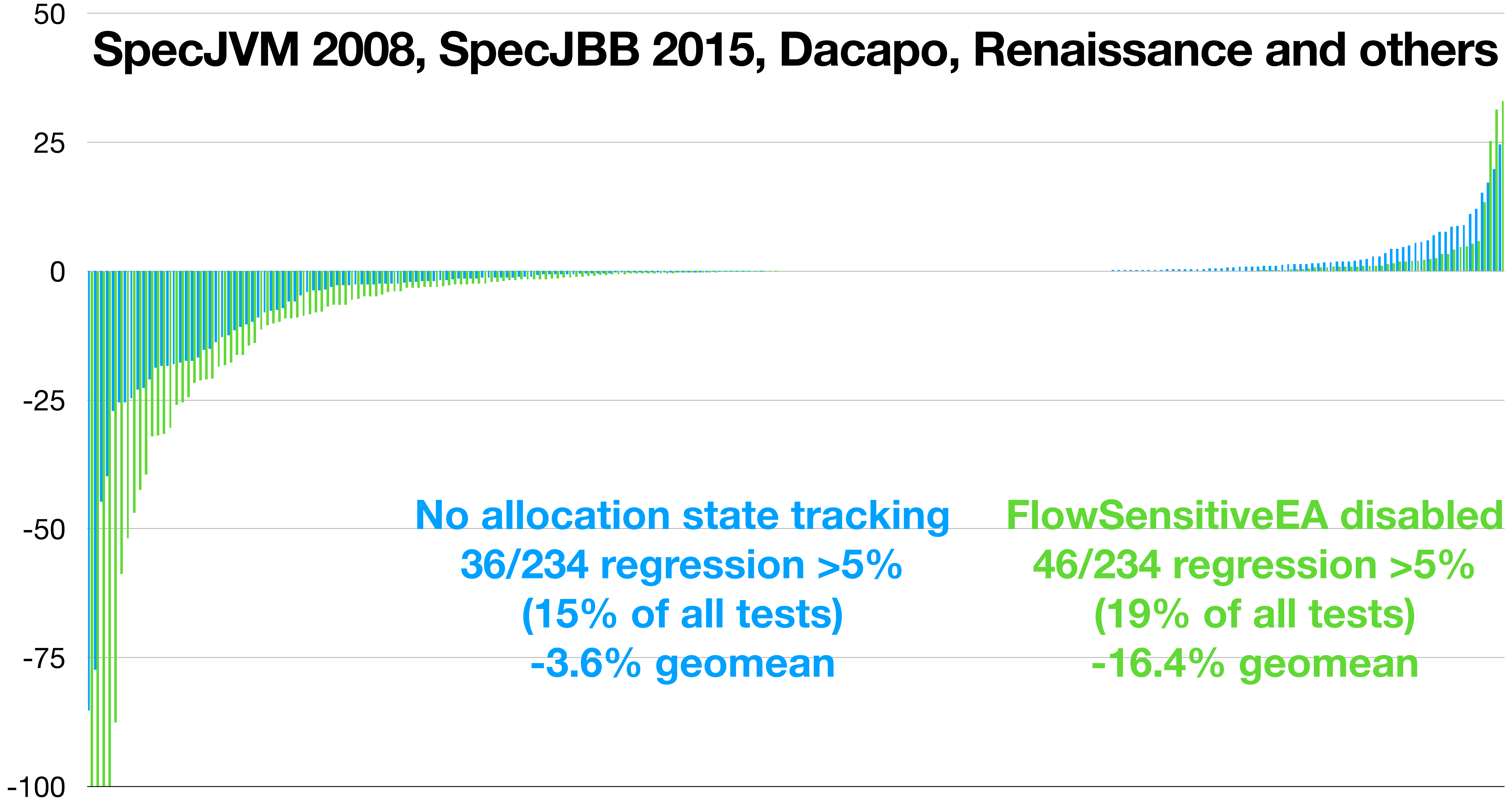
```
summ = 6
```

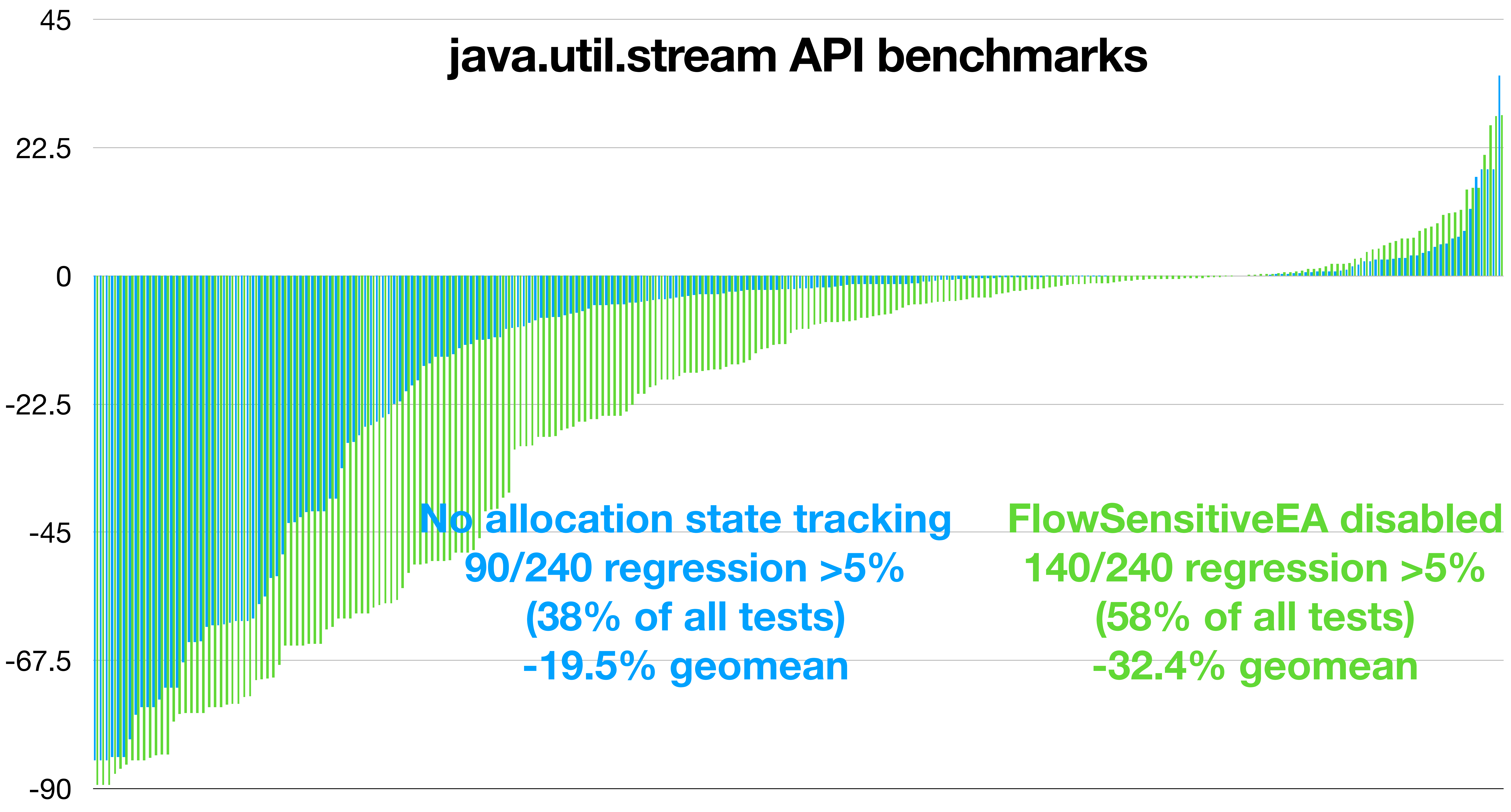
Agenda

- Introductions
- CaptureTracking analysis
- Falcon’s FlowSensitiveEA analysis
- FlowSensitiveEA transforms
- Performance results
- Conclusion

Performance results

- Compare default (with FlowSensitiveEA enabled) with
  - Disabled allocation state tracking (no points-to graph)
    - Object graphs are still handled by iterative optimization
  - Disabled FlowSensitiveEA pass


50
SpecJVM 2008, SpecJBB 2015, Dacapo, Renaissance and others
25
0
-25
-50
-75
-100
No allocation state tracking
36/234 regression >5%
(15% of all tests)
-3.6% geomean
FlowSensitiveEA disabled
46/234 regression >5%
(19% of all tests)
-16.4% geomean



java.util.stream API benchmarks
45
22.5
0
-22.5
-45
-67.5
-90
No allocation state tracking
90/240 regression >5%
(38% of all tests)
-19.5% geomean
FlowSensitiveEA disabled
140/240 regression >5%
(58% of all tests)
-32.4% geomean

Agenda

- Introductions
- CaptureTracking analysis
- Falcon’s FlowSensitiveEA analysis
- FlowSensitiveEA transforms
- Performance results
- Conclusion

Conclusion

- Java code has a lot of opportunities for EA
- We identified some limitations in CaptureTracking
    - E.g. handling of unescaped object graphs
- We implemented downstream analysis and transforms to solve those limitations
    - As a result observed substantial performance gains
- Integration with existing passes in non-trivial due to update/invalidation problem

Questions?