ægraphs: Acyclic E-graphs

for Efficient Optimization in a Production Compiler

Chris Fallin *(F5)* *with* Jamey Sharp, Trevor Elliott, Nick Fitzgerald, Alex Crichton, Alexa VanHattum
chris@cfallin.org

ægraphs: Acyclic E-graphs

for Efficient Optimization in a Production Compiler

… two years later

Chris Fallin *(F5)* *with* Jamey Sharp, Trevor Elliott, Nick Fitzgerald, Alex Crichton, Alexa VanHattum
chris@cfallin.org

Cranelift

Cranelift

- Open-source general-purpose optimizing compiler backend
  - Written in Rust + a pattern-matching DSL (~200KLoC, ~130KLoC tests)
  - SSA input, four ISAs (x86-64, aarch64, riscv64, s390x)

Cranelift

- Open-source general-purpose optimizing compiler backend
  - Written in Rust + a pattern-matching DSL (~200KLoC, ~130KLoC tests)
  - SSA input, four ISAs (x86-64, aarch64, riscv64, s390x)
- Used in production as part of Wasmtime
- **Goals**: compile speed, simplicity, verifiability

Why egraphs?

- Circa 2022, pass-order problem became apparent

Why egraphs?

- Circa 2022, pass-order problem became apparent

  GVN, LICM

Why egraphs?

- Circa 2022, pass-order problem became apparent

  GVN, LICM, GVN (just in case)

Why egraphs?

- Circa 2022, pass-order problem became apparent

  GVN, LICM, GVN (just in case)
  algebraic rewrites

Why egraphs?

- Circa 2022, pass-order problem became apparent

  GVN, LICM, GVN (just in case)
  algebraic rewrites
  GVN

Why egraphs?

- Circa 2022, pass-order problem became apparent

  GVN, LICM, GVN (just in case)
  algebraic rewrites
  GVN
  alias analysis??

Why egraphs?

- Circa 2022, pass-order problem became apparent

  GVN, LICM, GVN (just in case)
  algebraic rewrites
  GVN
  alias analysis??
  GVN???

Why egraphs?

- Circa 2022, pass-order problem became apparent

  GVN, LICM, GVN (just in case)
  algebraic rewrites
  GVN
  alias analysis??
  GVN???

- We were doing term-rewriting already (ISLE) in instruction selection
  - What if we explored multiple paths?

Why egraphs?

- Circa 2022, pass-order problem became apparent

  GVN, LICM, GVN (just in case)
  algebraic rewrites
  GVN
  alias analysis??
  GVN???

- We were doing term-rewriting already (ISLE) in instruction selection
  - What if we explored multiple paths?
- **How hard could it be?**

Background: ISLE

- Pattern matching language: like classical TRS, with strong Prolog influences
  - Terms are *strongly typed,* rather than homogenous value domain
  - Explicit FFI to external (Rust) code, with defined semantics
    - *Extractors* (“unapply”): match on LHS, return bindings
    - *Constructors*: invoke on RHS, return value
  - Implicit conversions based on types
  - “Overlap checks” and rule priorities
  - Backtracking on the LHS but not on the RHS (commit once match)

Background: ISLE

```
(rule (lower (iadd (imul x y) z))
      (madd x y z))
```

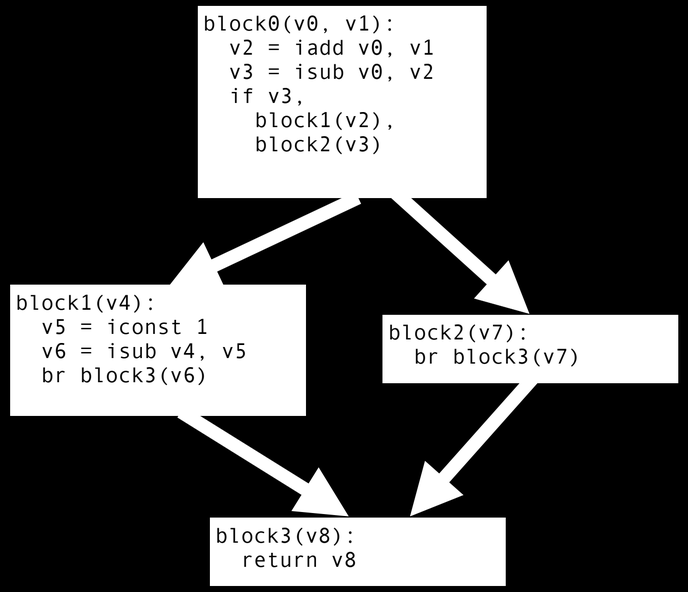
$\Longrightarrow$

```
(rule (lower (iadd (def_inst (imul x y)) z))
  (madd
   (put_in_reg x)
   (put_in_reg y)
   (put_in_reg z)))
```

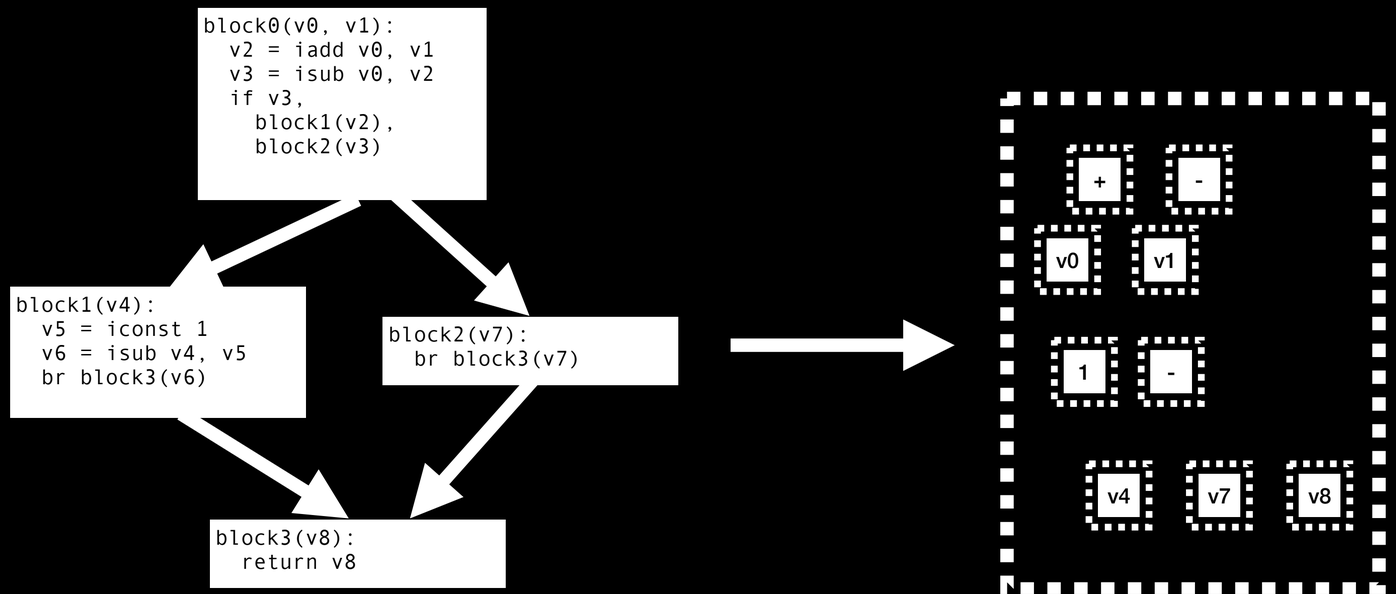
Background: ISLE

- Imperative shell and FFI let us define a glue to the rest of the compiler
- ISLE meta-compiler generates efficient Rust (single large matcher) at a constructor entry-point; equal or better than hand-written code
- We had one “prelude”, for instruction selection
- Why not add another, for the mid-end?
  - Only one new ISLE feature needed: “multi-extractors/constructors”

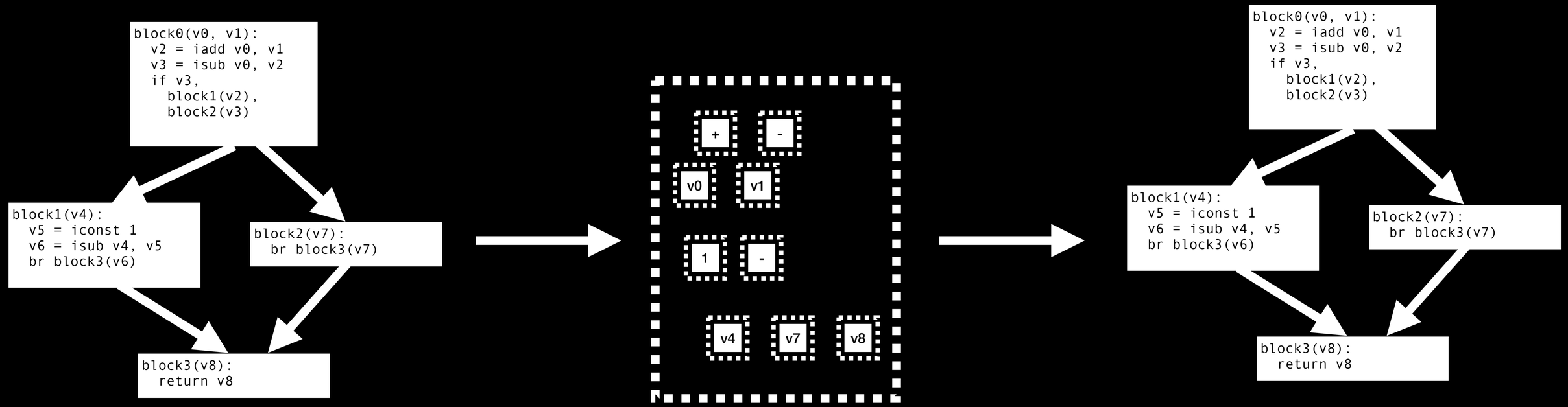
Optimization pipeline



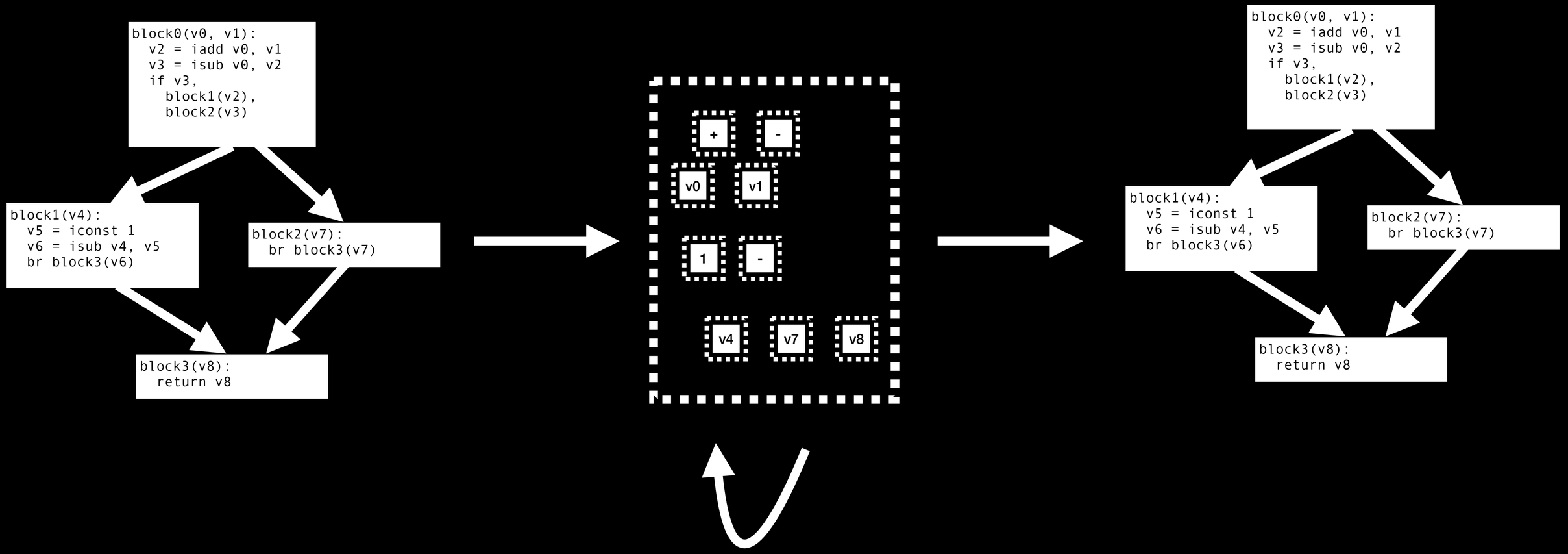
Optimization pipeline



Optimization pipeline



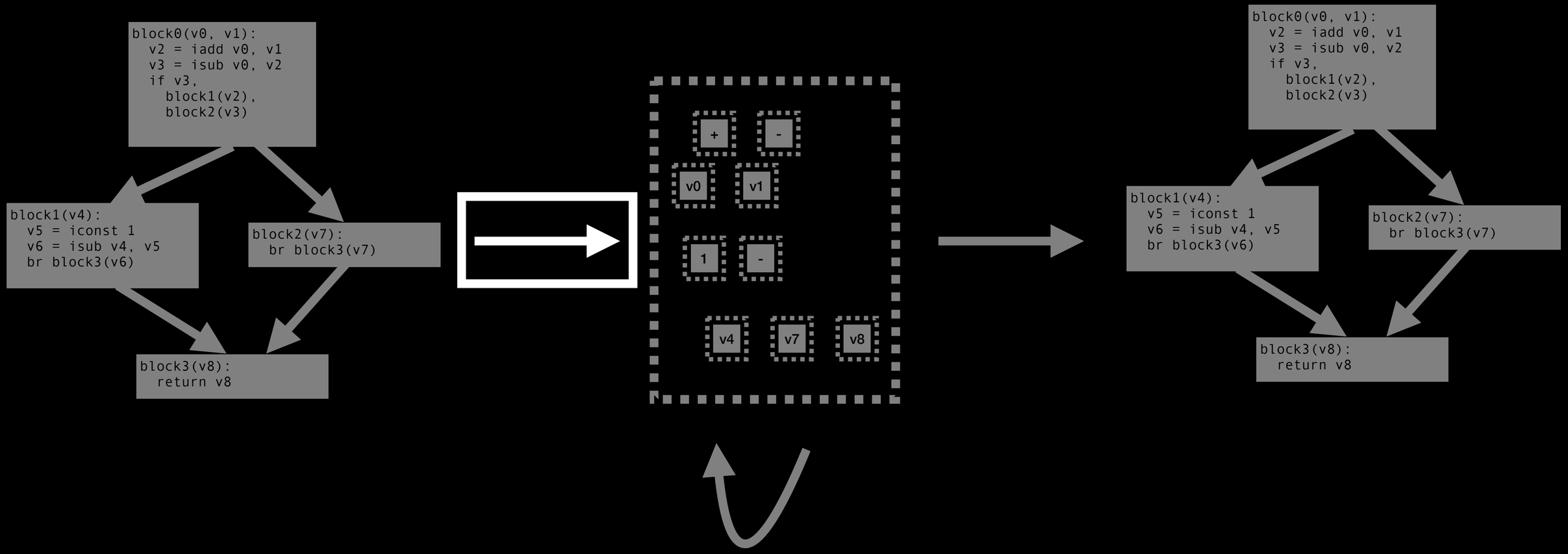
Optimization pipeline



x + 0 => x

...

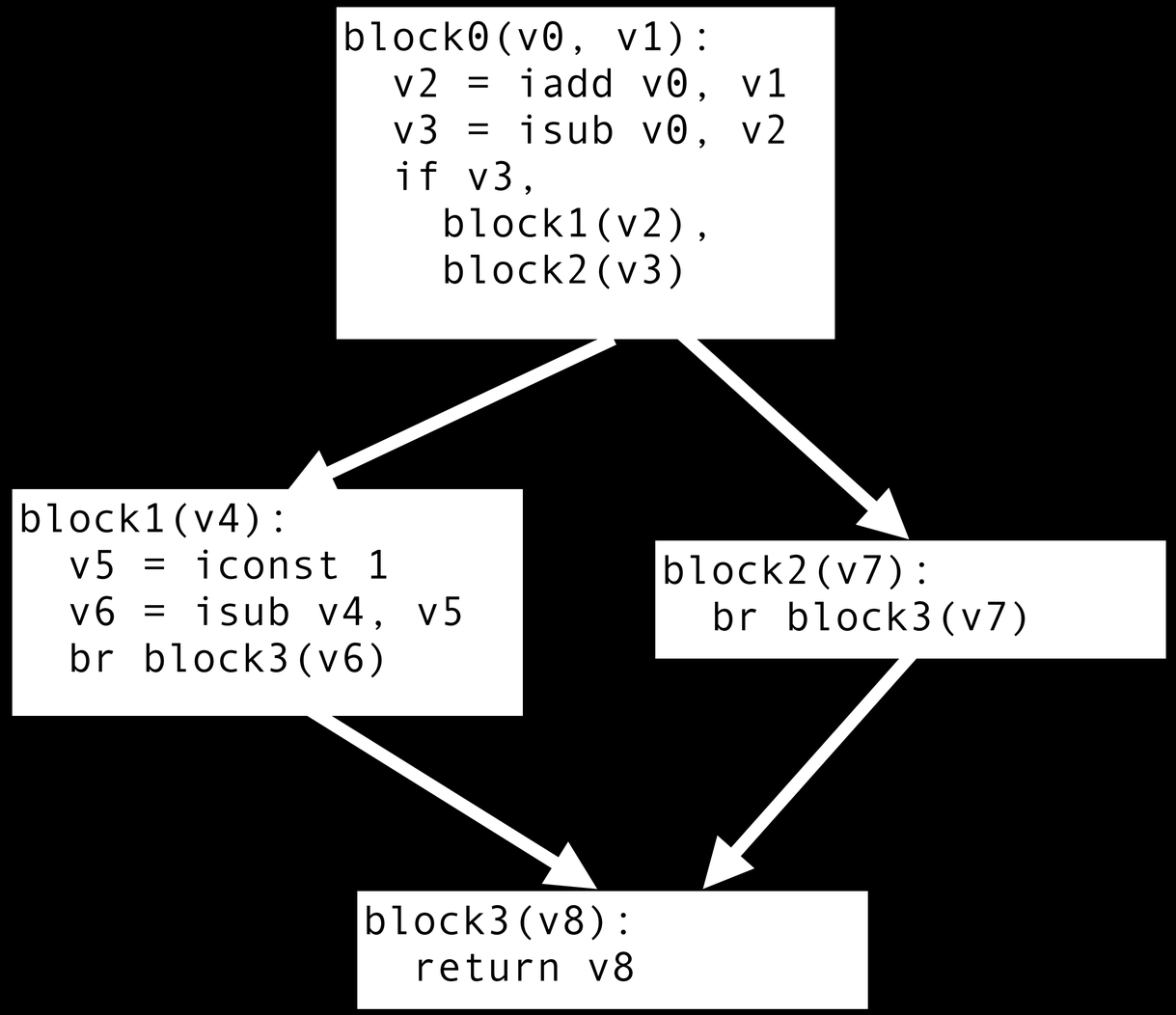
Optimization pipeline



x + 0 => x

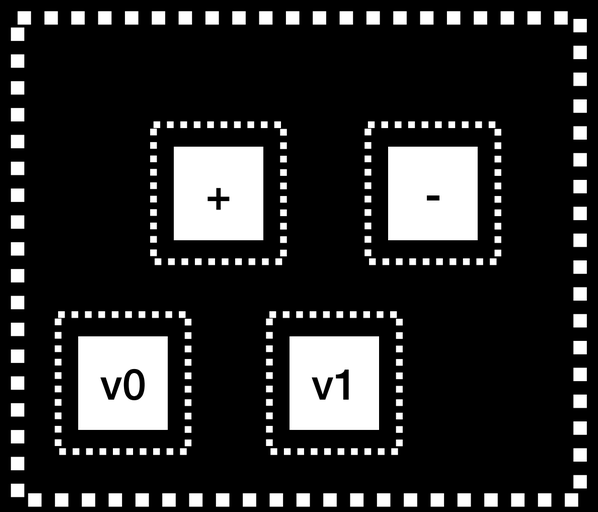
...

E-graph + CFG == ???


block0(v0, v1):
v2 = iadd v0, v1
v3 = isub v0, v2
if v3,
block1(v2),
block2(v3)
block1(v4):
v5 = iconst 1
v6 = isub v4, v5
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8

E-graph + CFG == ???

```
block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)
```

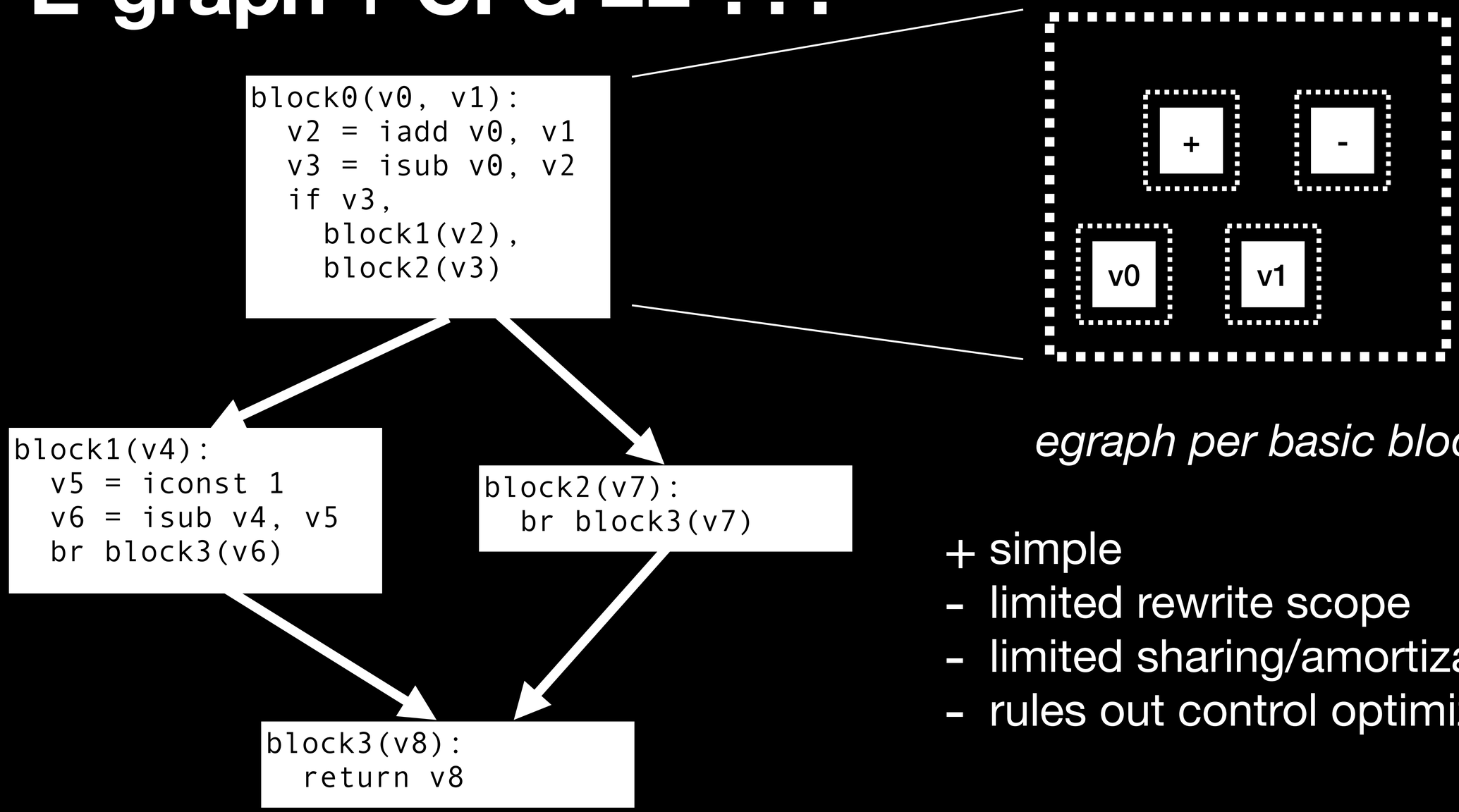



```
block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)
```

```
block2(v7):
  br block3(v7)
```

```
block3(v8):
  return v8
```

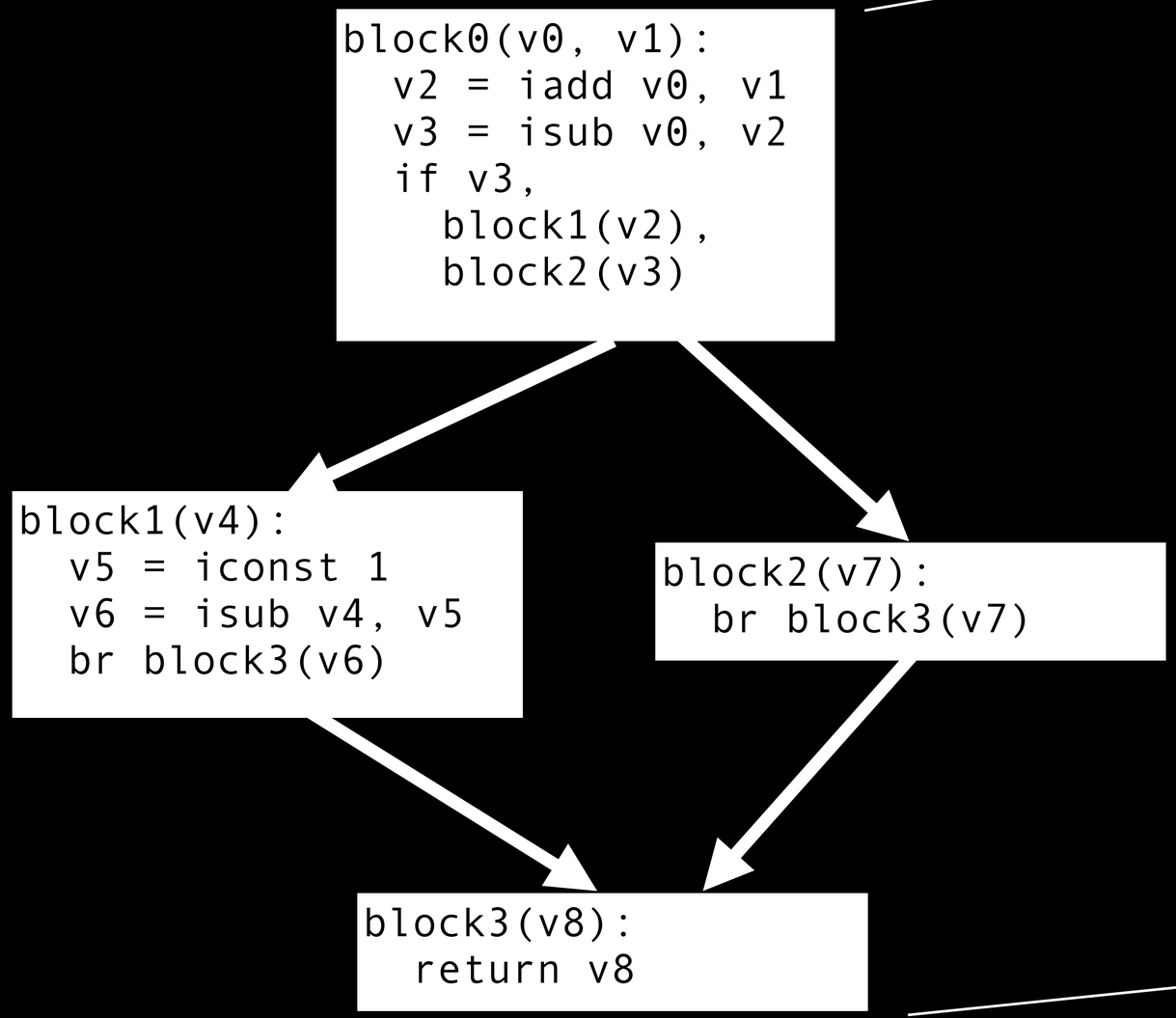
E-graph + CFG == ???

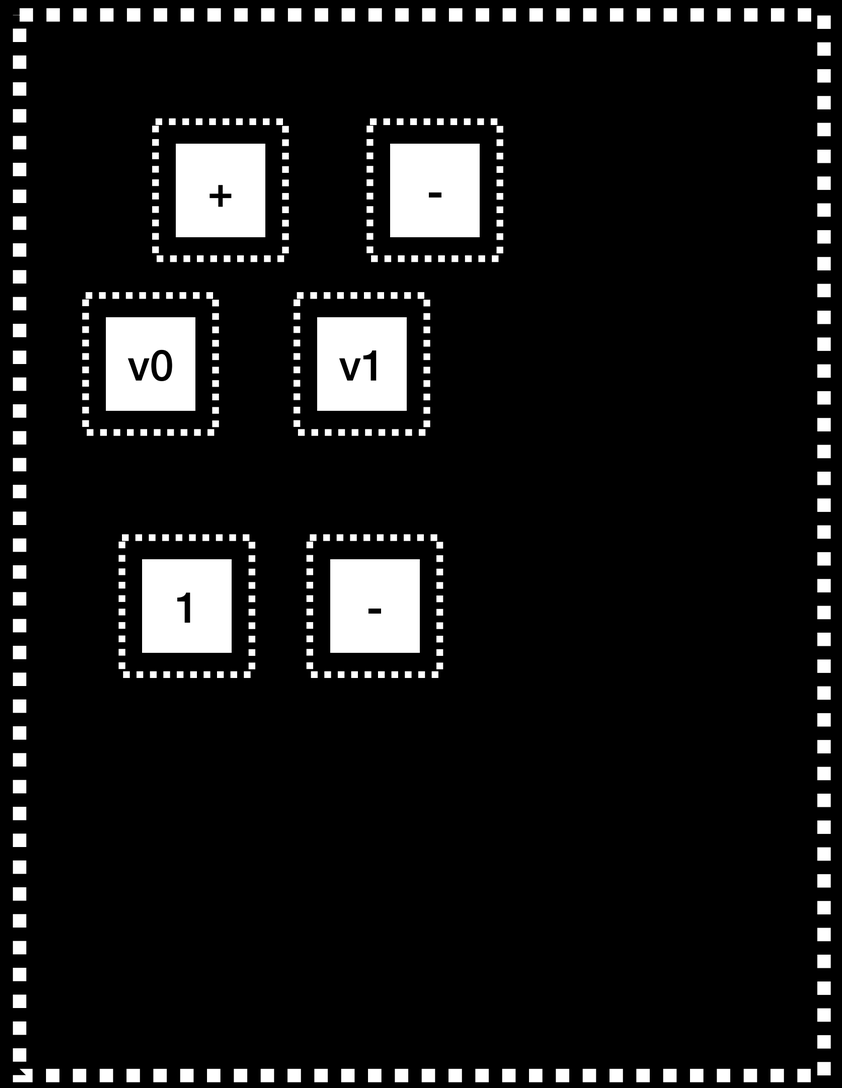


egraph per basic block:

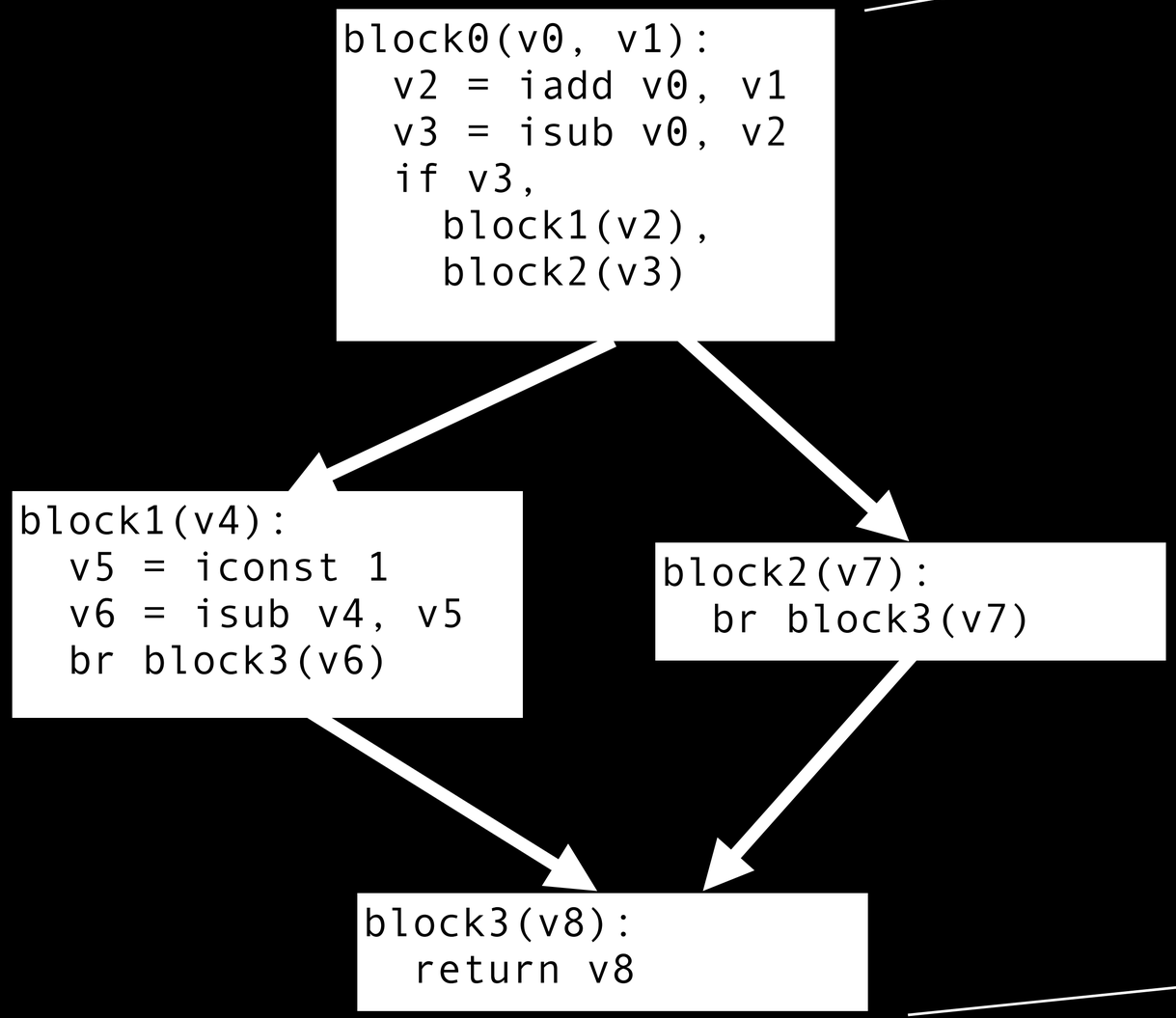
\+ simple
\- limited rewrite scope
\- limited sharing/amortization
\- rules out control optimizations

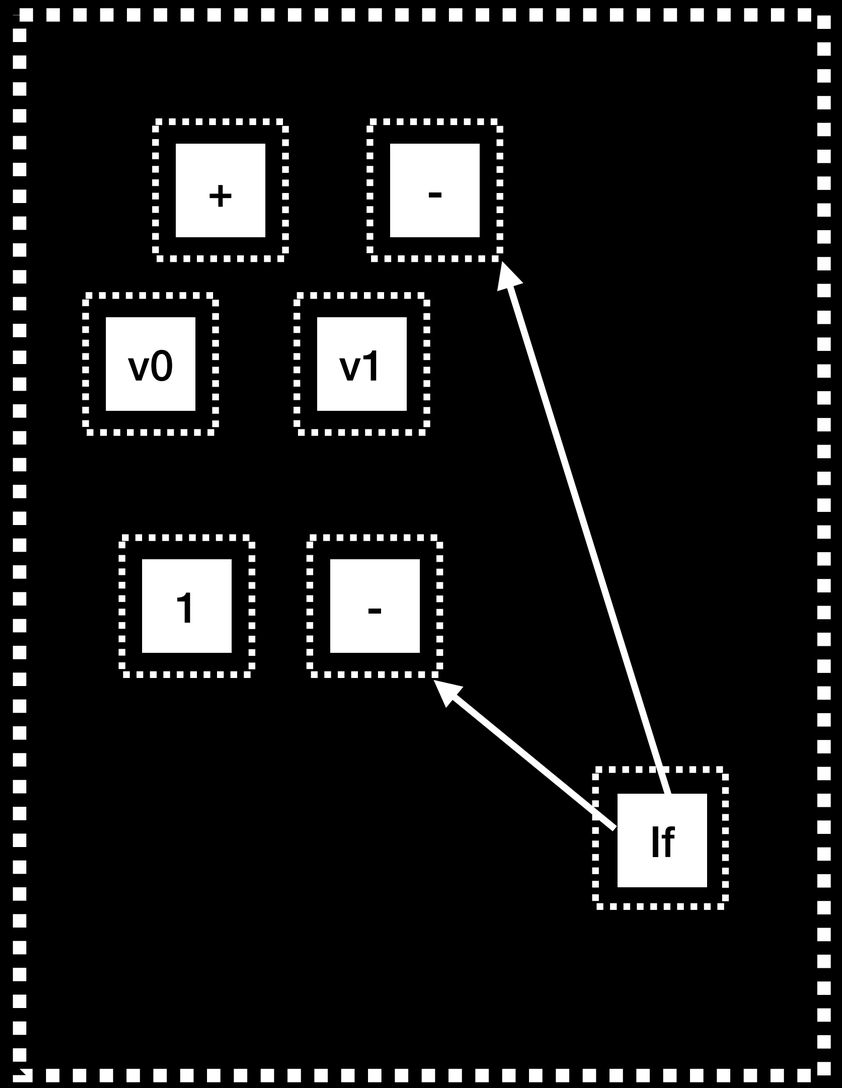
E-graph + CFG == ???


block0(v0, v1):
v2 = iadd v0, v1
v3 = isub v0, v2
if v3,
block1(v2),
block2(v3)
block1(v4):
v5 = iconst 1
v6 = isub v4, v5
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8

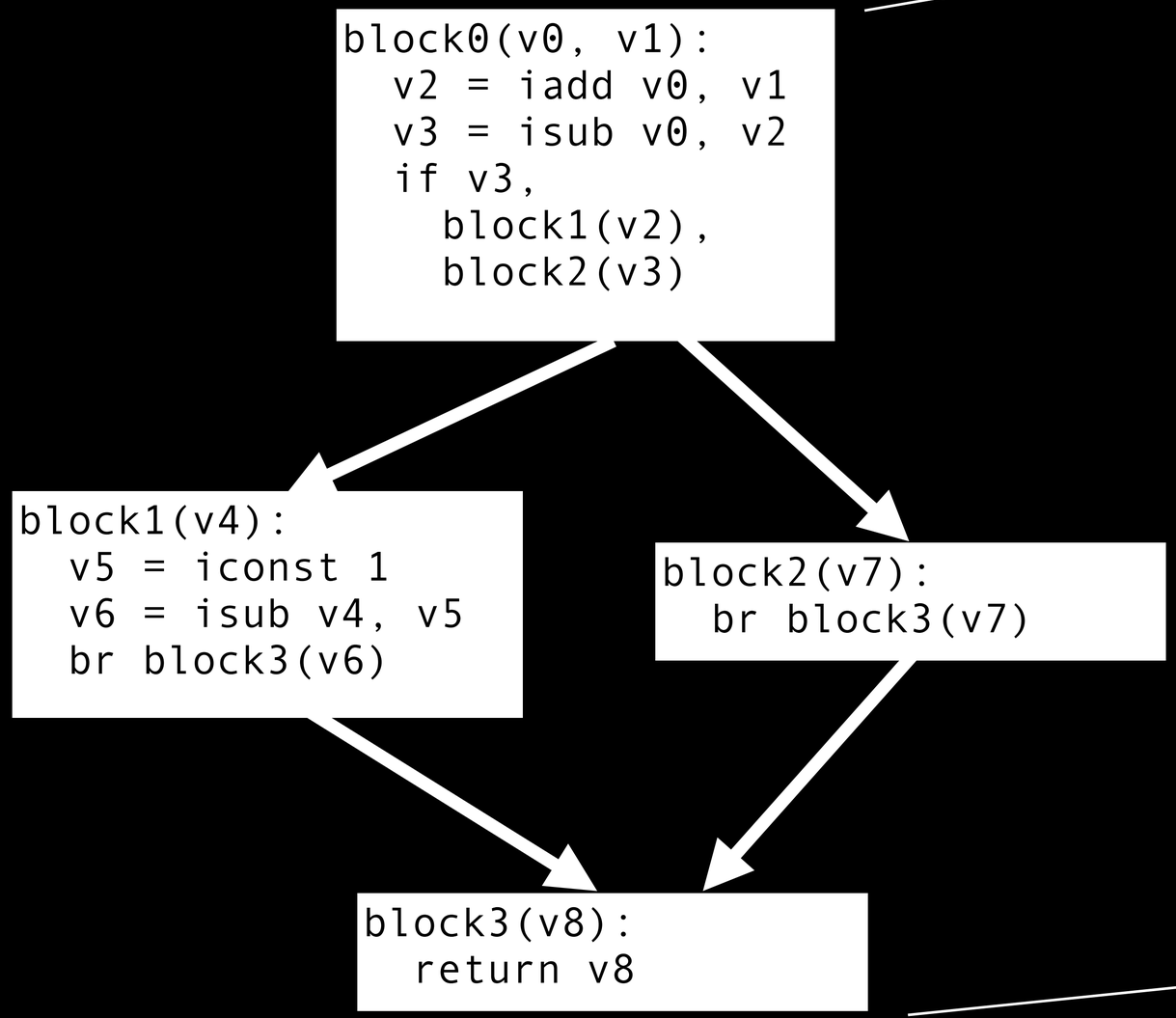

+
-
v0
v1
1
-

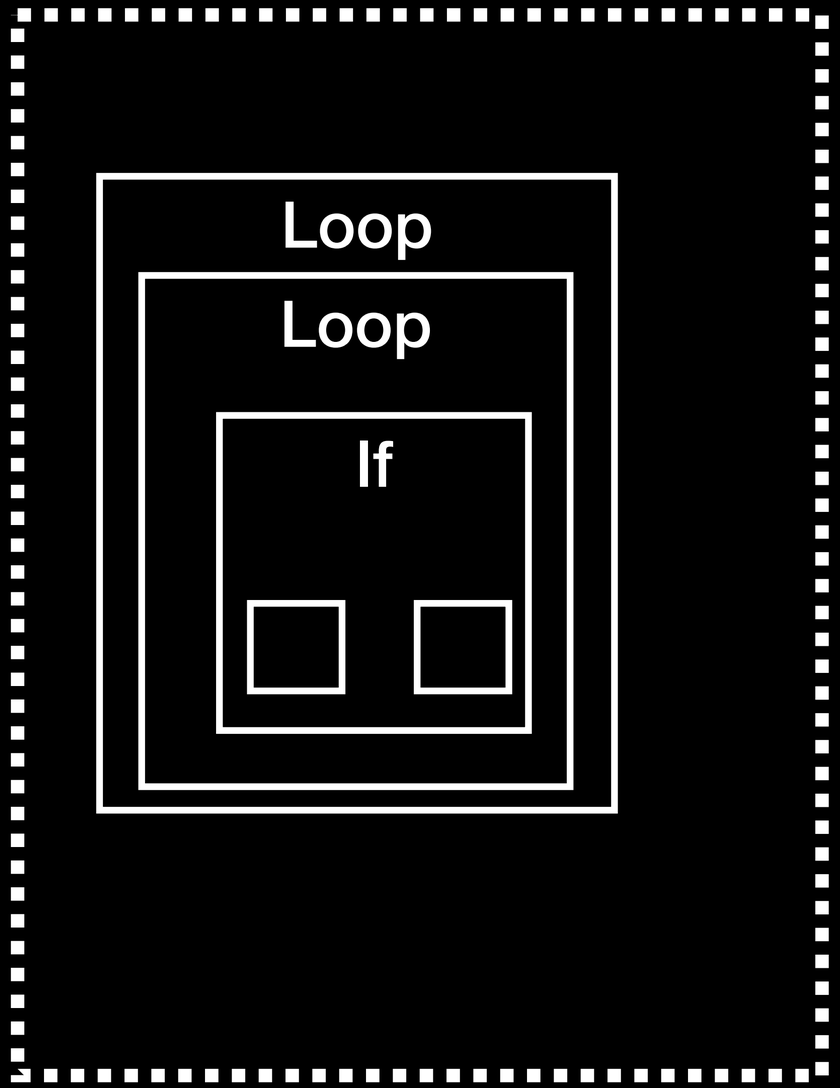
E-graph + CFG == ???


block0(v0, v1):
v2 = iadd v0, v1
v3 = isub v0, v2
if v3,
block1(v2),
block2(v3)
block1(v4):
v5 = iconst 1
v6 = isub v4, v5
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8


+
-
v0
v1
1
-
If

E-graph + CFG == ???


block0(v0, v1):
v2 = iadd v0, v1
v3 = isub v0, v2
if v3,
block1(v2),
block2(v3)
block1(v4):
v5 = iconst 1
v6 = isub v4, v5
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8

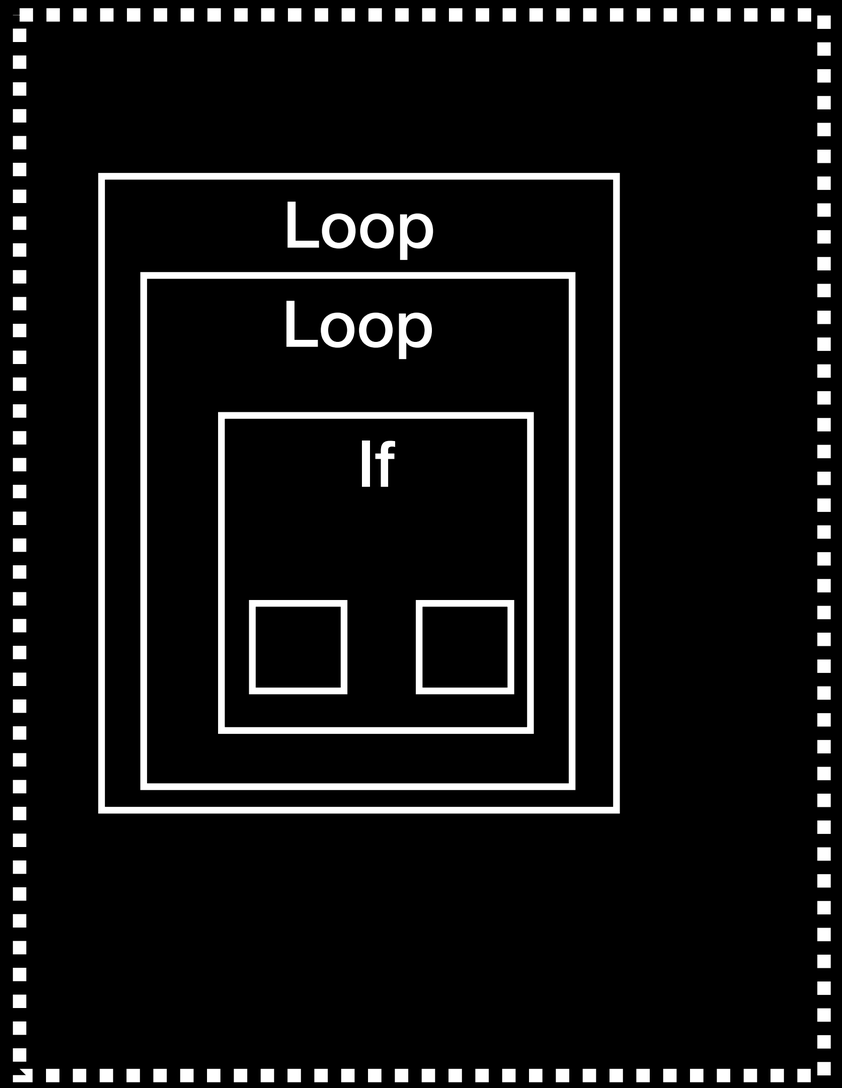

Loop
Loop
If

E-graph + CFG == ???

Region-nodes in egraph:

+ powerful optimizations!
+ strongly normalizing
+ more compact IR
+ cheaper analysis?

- very different from CFG (conversion overheads)
- side-effects are tricky
- issues with irreducible control flow

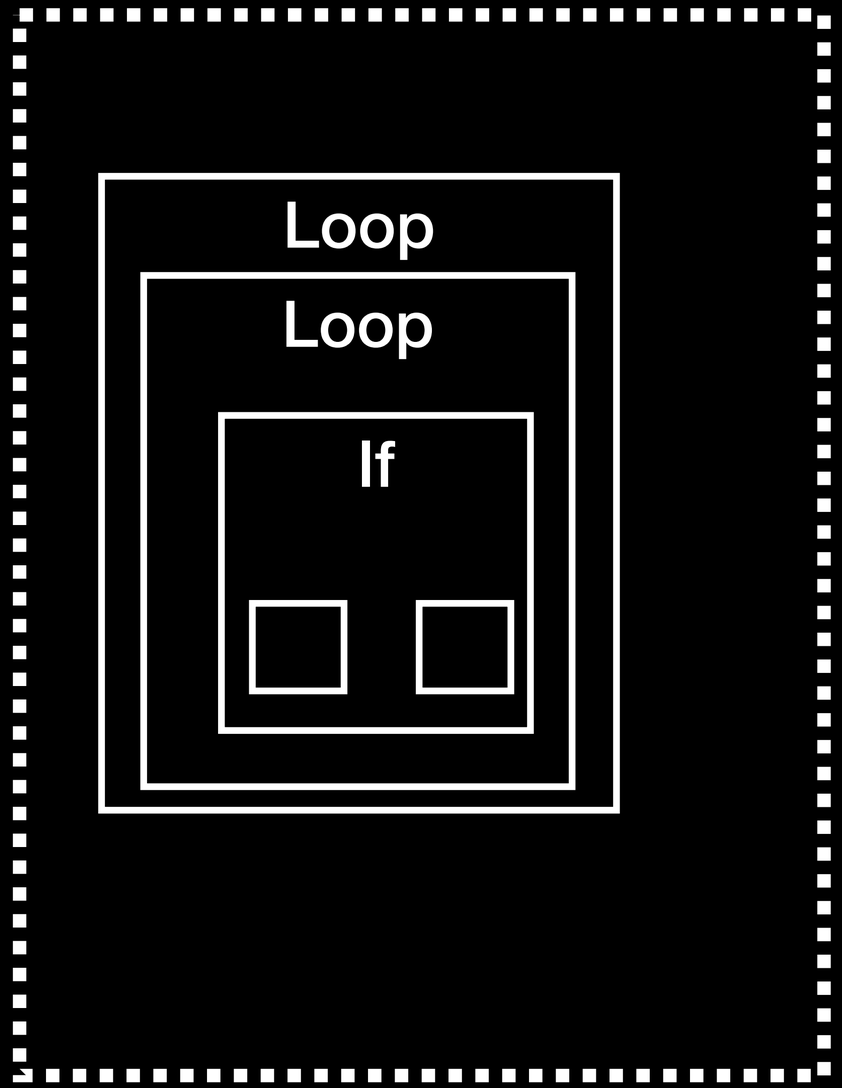


E-graph + CFG == ???

Region-nodes in egraph:

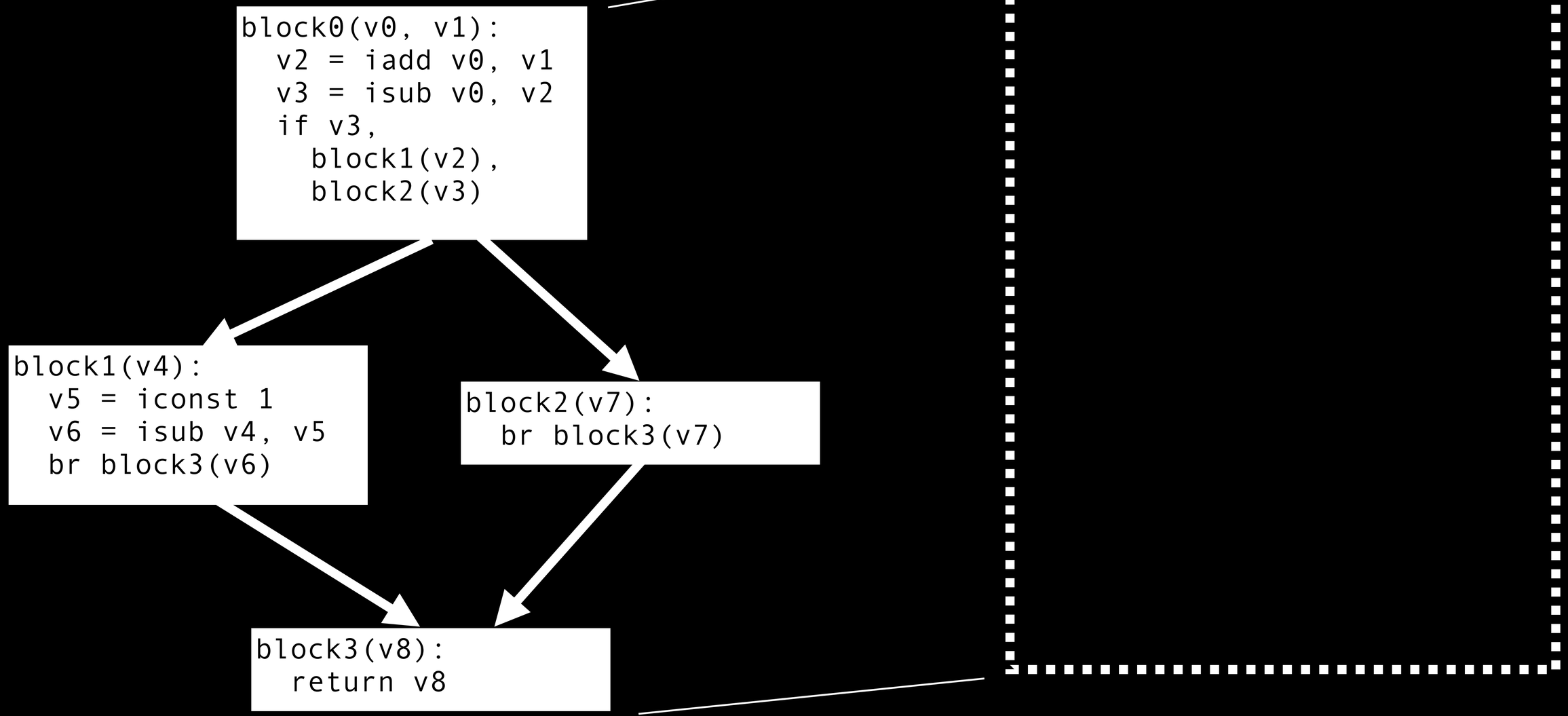
+ powerful optimizations!
+ strongly normalizing
+ more compact IR
+ cheaper analysis?

- very different from CFG (conversion overheads)
- side-effects are tricky
- issues with irreducible control flow

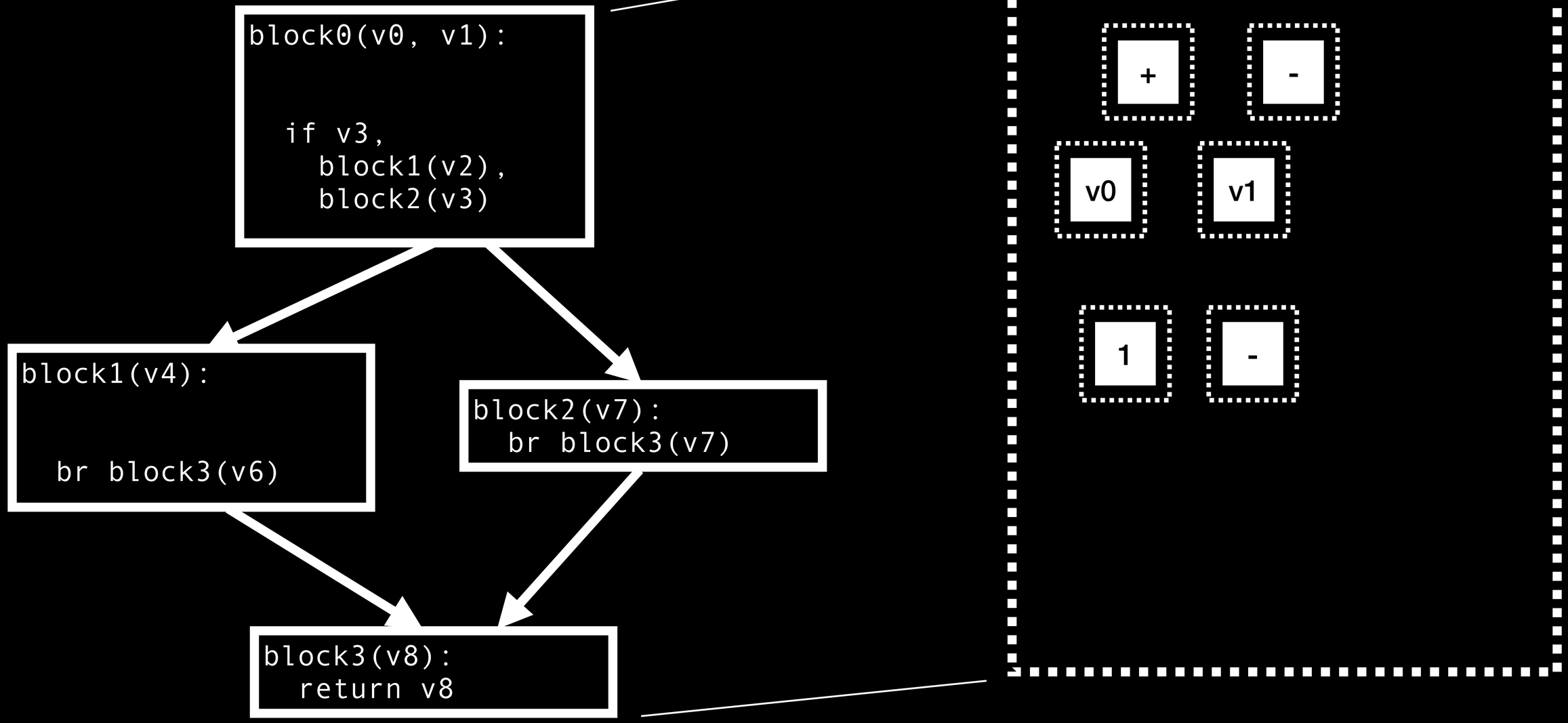


Jamey Sharp's prototype: https://github.com/jameysharp/optir

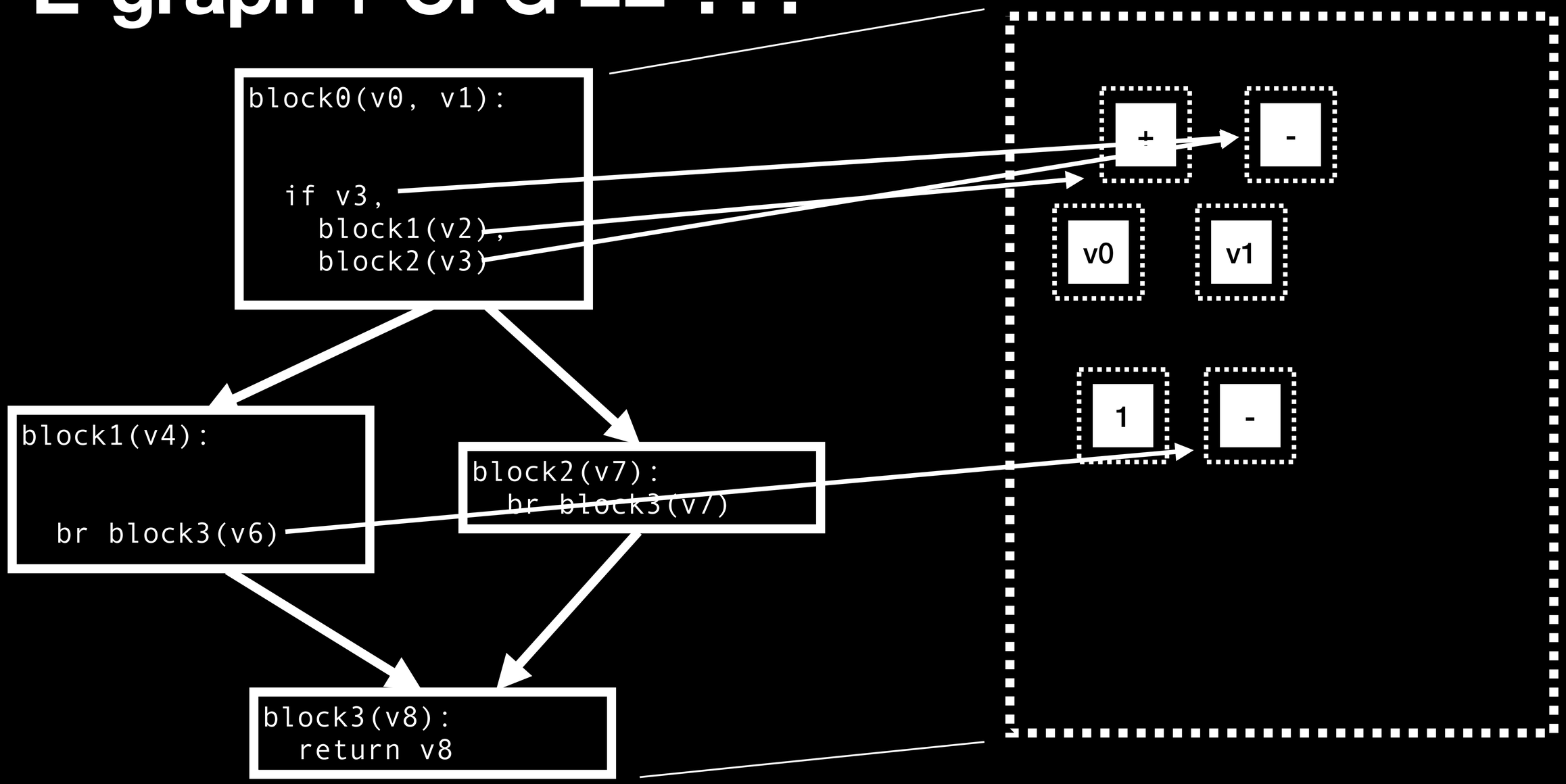
E-graph + CFG == ???


block0(v0, v1):
v2 = iadd v0, v1
v3 = isub v0, v2
if v3,
block1(v2),
block2(v3)
block1(v4):
v5 = iconst 1
v6 = isub v4, v5
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8

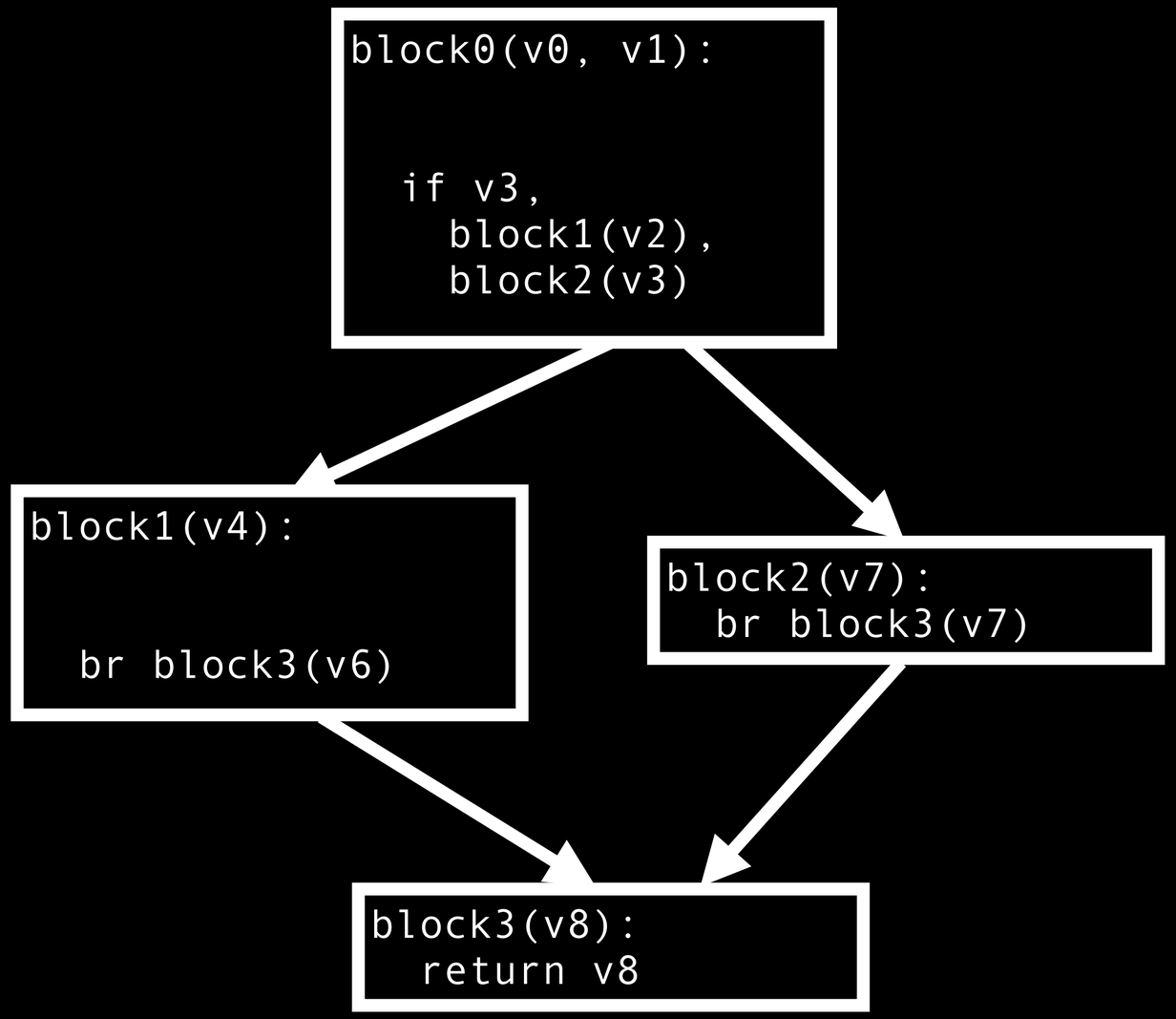
E-graph + CFG == ???


block0(v0, v1):
if v3,
block1(v2),
block2(v3)
block1(v4):
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8
+
-
v0
v1
1
-

E-graph + CFG == ???


block0(v0, v1):
if v3,
block1(v2),
block2(v3)
block1(v4):
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8
+
-
v0
v1
1
-

E-graph + CFG == ???



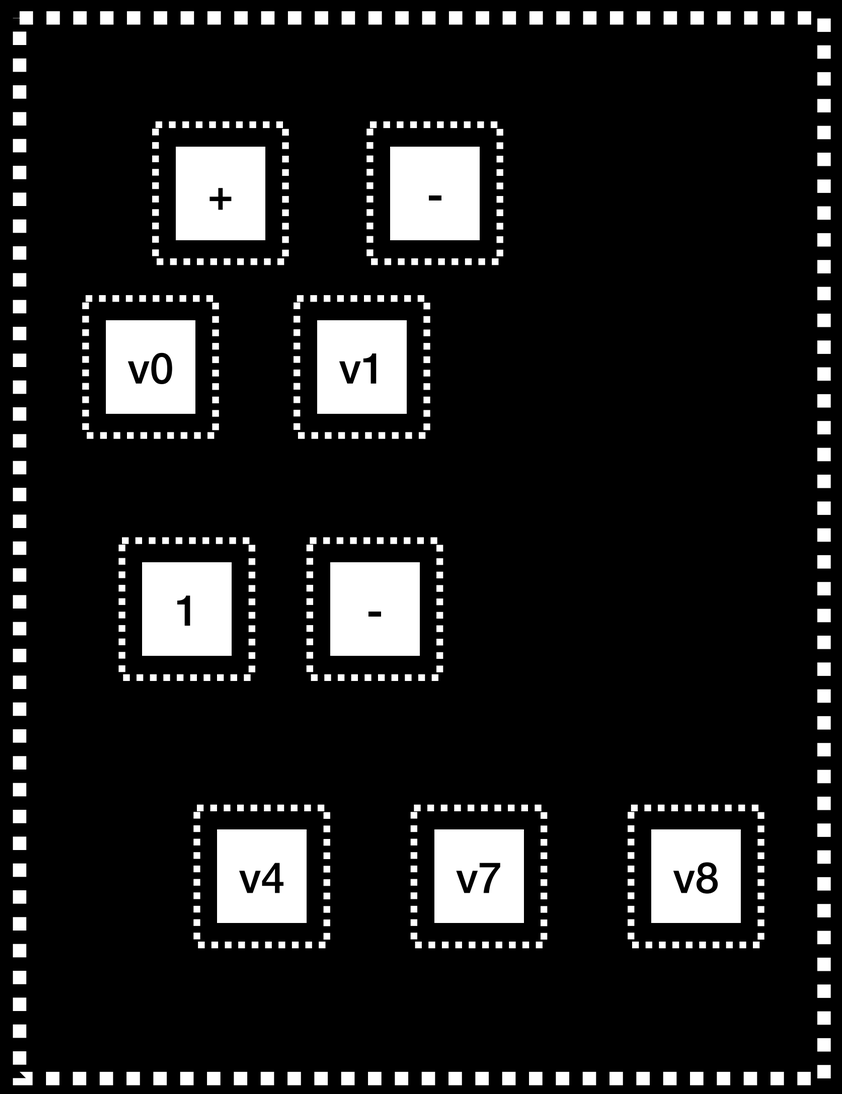
CFG skeleton contains:

- all blocks, with blockparams
- side-effecting operators
- block terminators (branches)

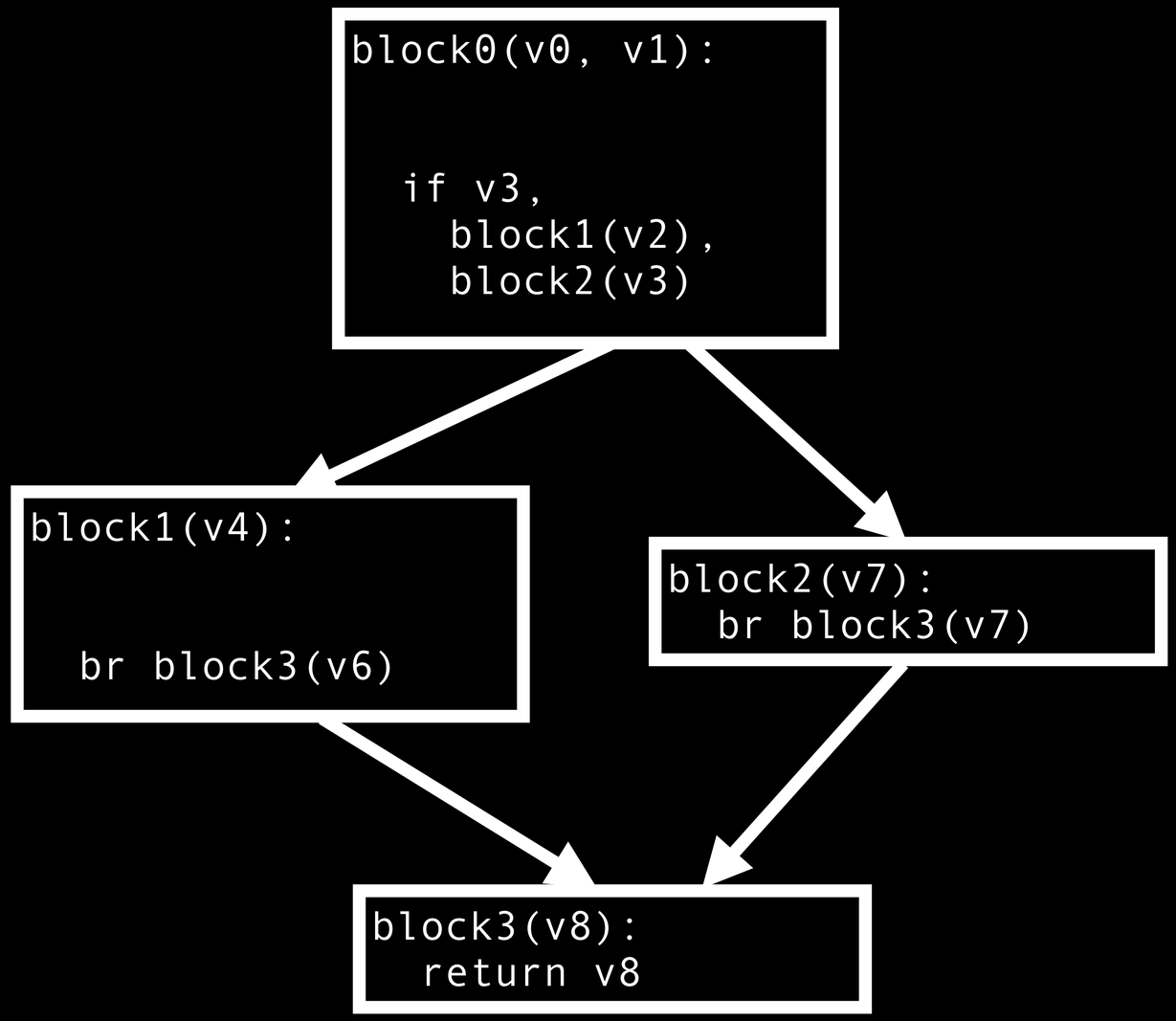
E-graph + CFG == ???

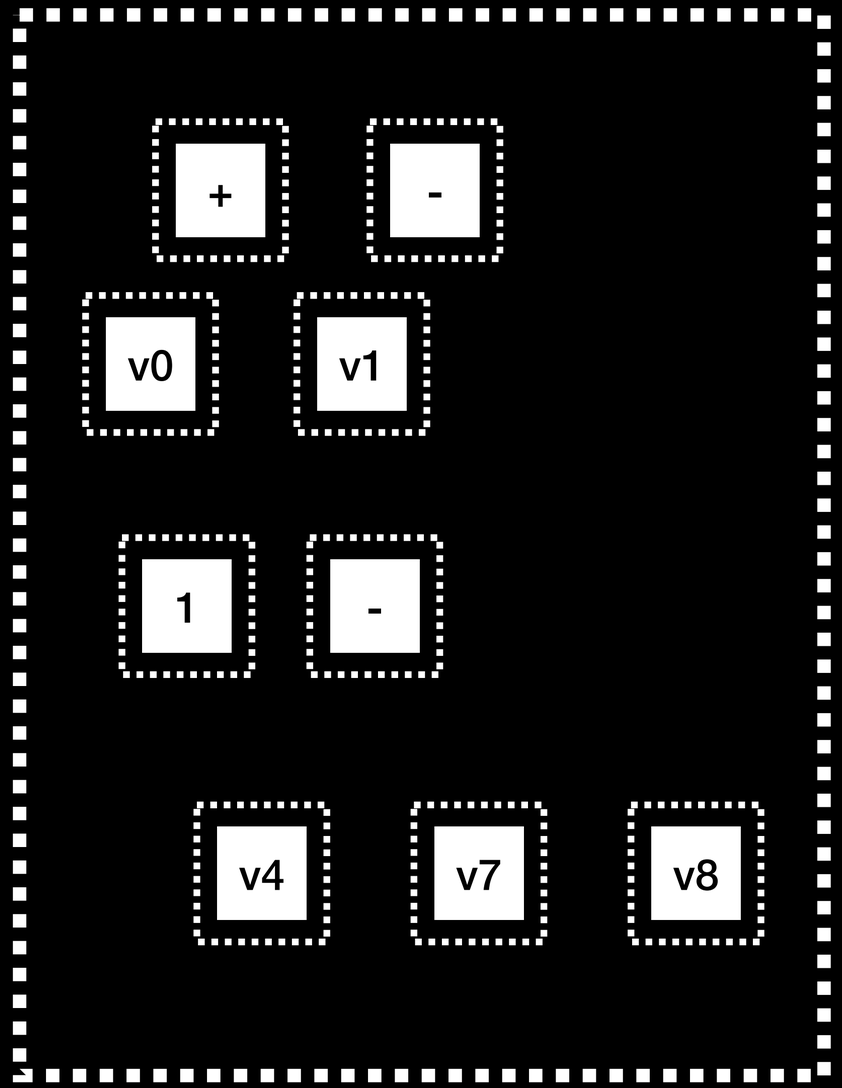
egraph contains:

- blockparam values, as terminals
- all pure operators,
  without associated location



E-graph + CFG == ???


block0(v0, v1):
if v3,
block1(v2),
block2(v3)
block1(v4):
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8


+
-
v0
v1
1
-
v4
v7
v8

E-graph + CFG == ???

egraph with CFG skeleton:

+ cheap to convert to/from CFG
  + algorithmically *and* in implementation
+ optimizations across function scope (mostly)

- harder to express rewrites that alter side-effects
- need special support for “seeing through” blockparams

E-graph + CFG == ???

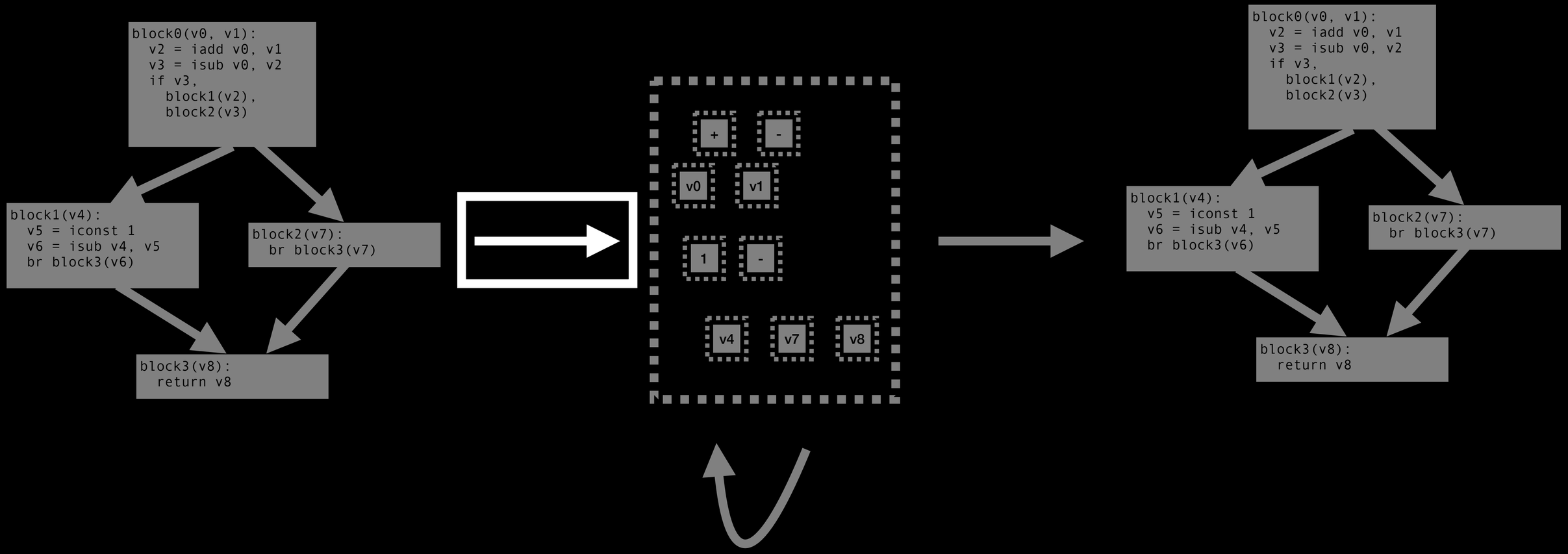
egraph with CFG skeleton:

\+ cheap to convert to/from CFG
  \+ algorithmically *and* in implementation
\+ optimizations across function scope (mostly)

\- harder to express rewrites that alter side-effects
\- need special support for “seeing through” blockparams

we do this; downsides are open questions to solve

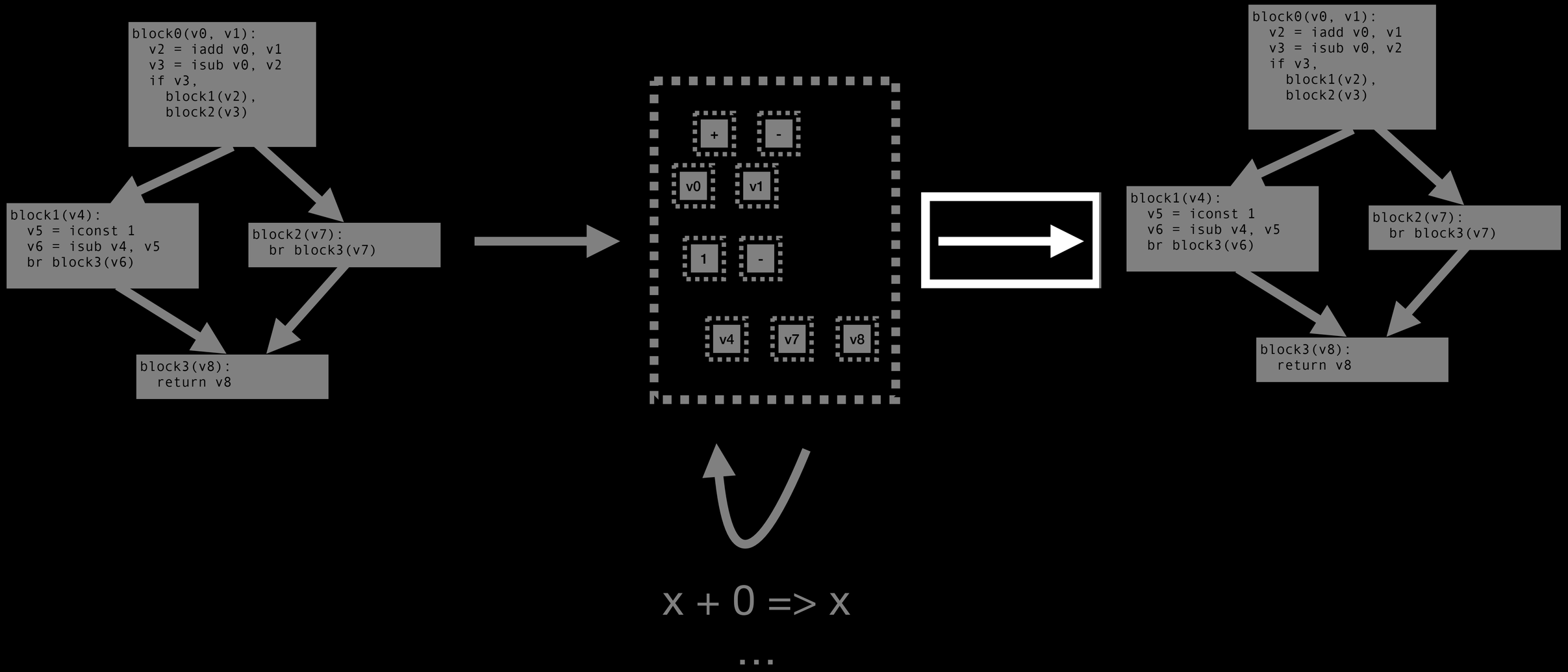
Optimization pipeline



x + 0 => x

...

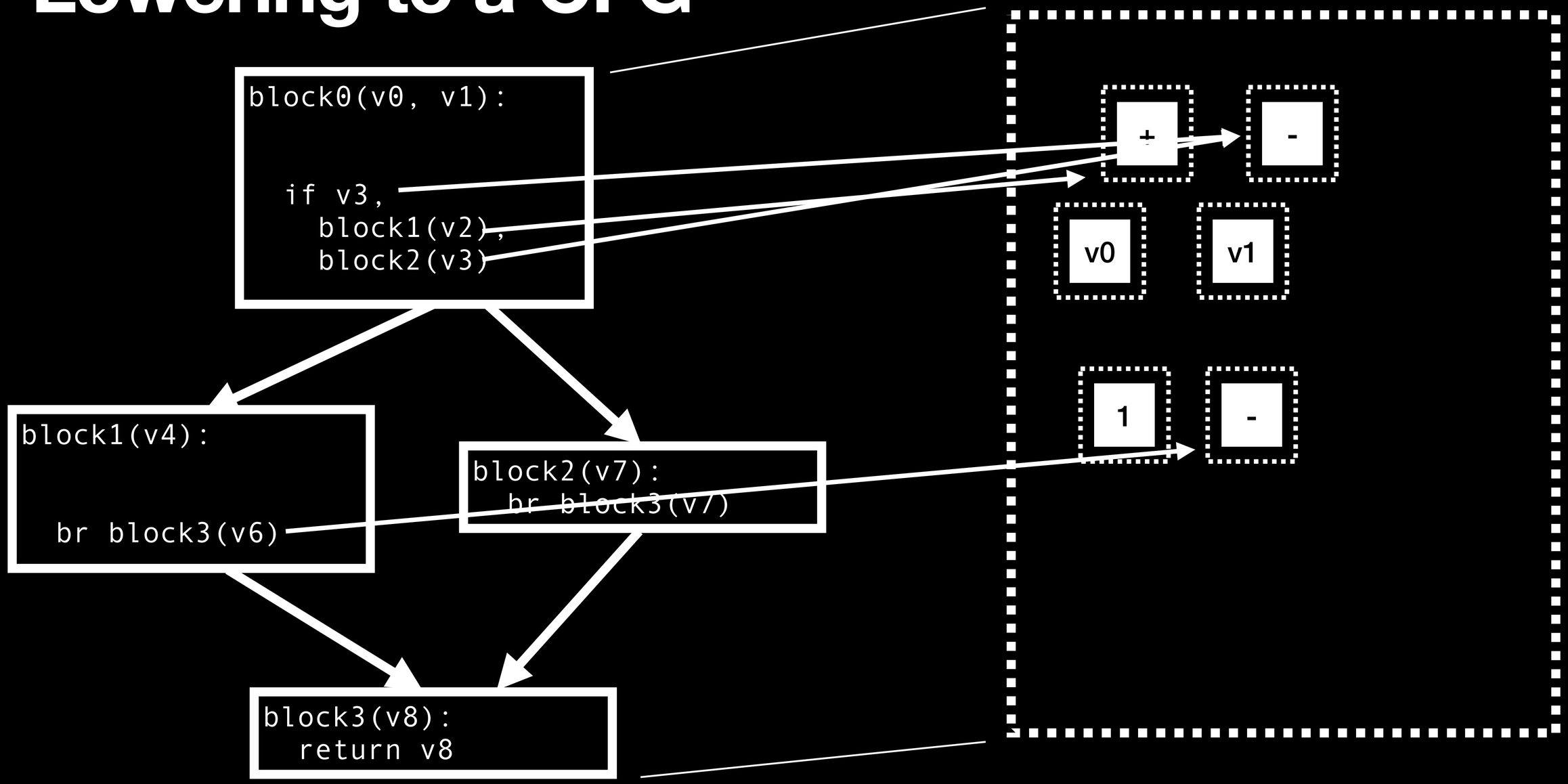
Optimization pipeline



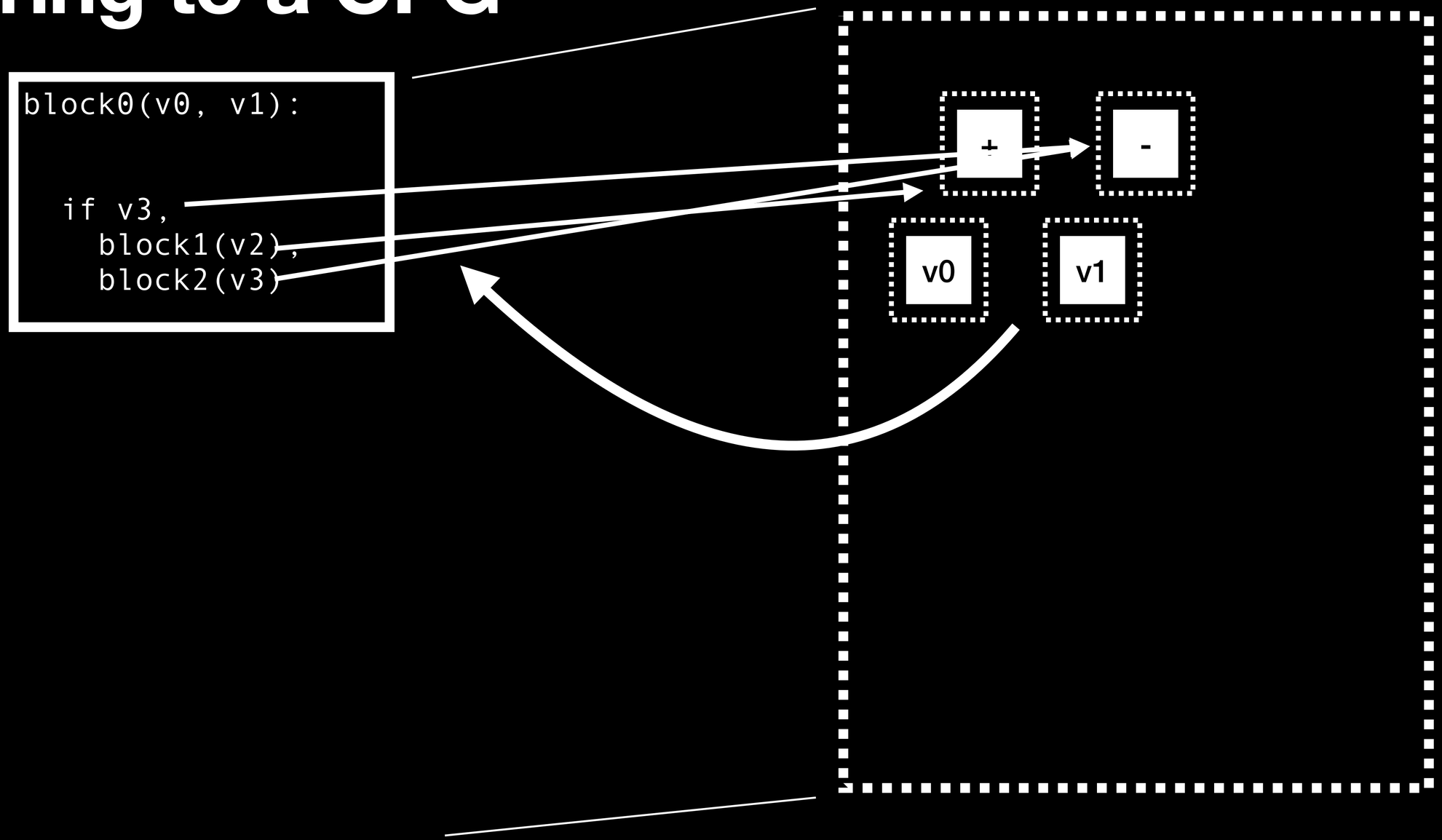
x + 0 => x

...

Lowering to a CFG

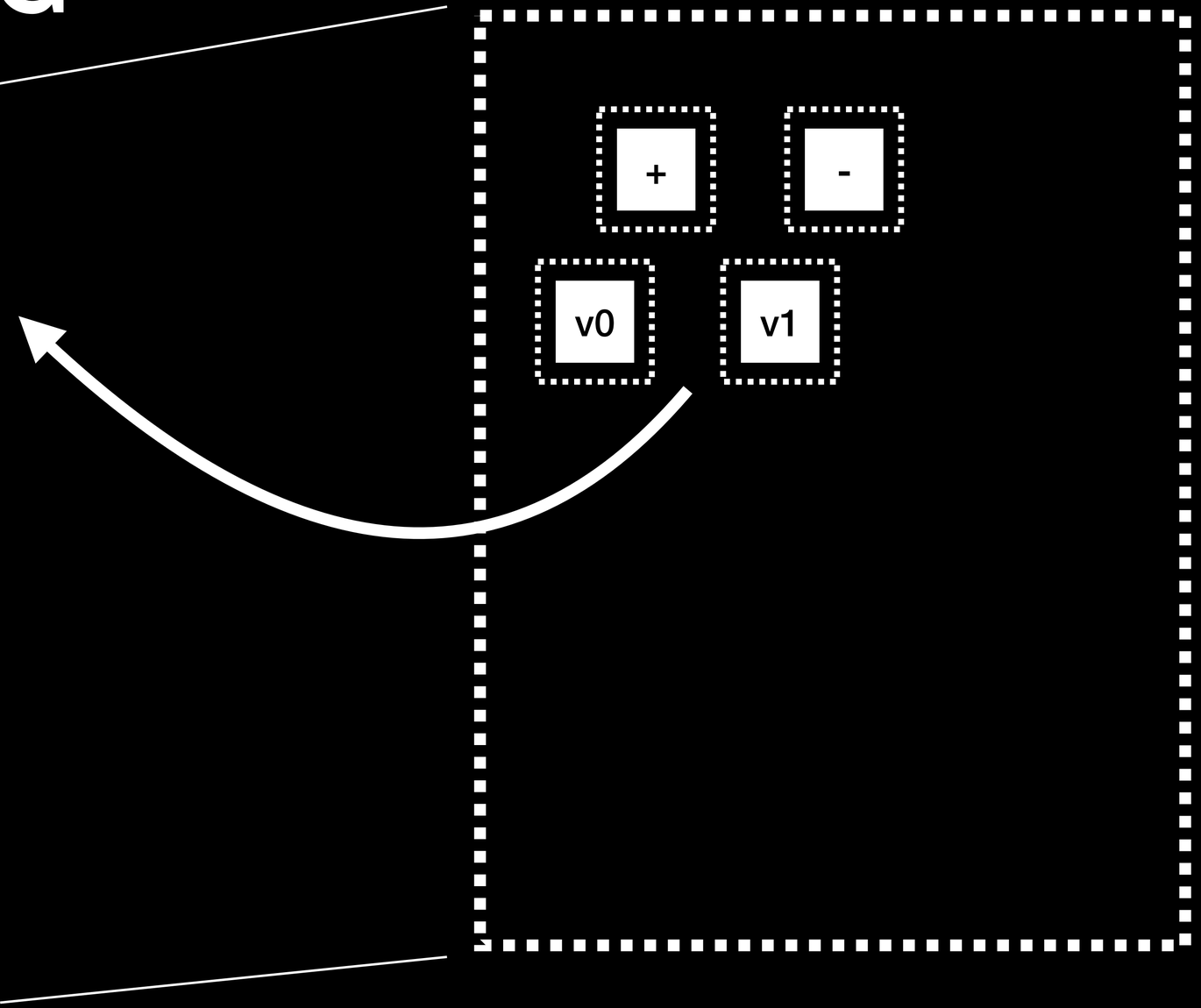


Lowering to a CFG



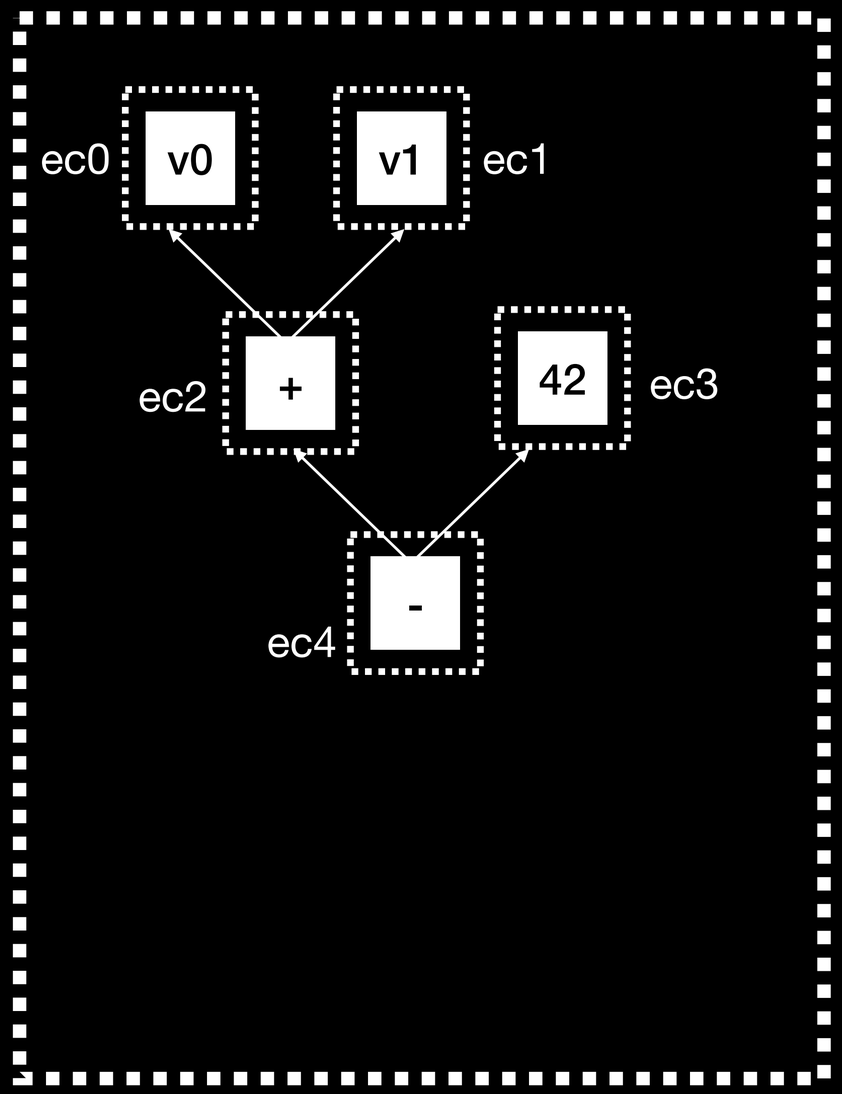
Lowering to a CFG

```
block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)
```

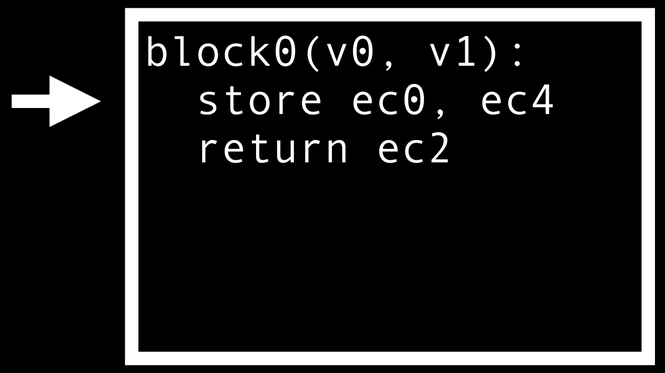


Elaboration

```
block0(v0, v1):
  store ec0, ec4
  return ec2
```

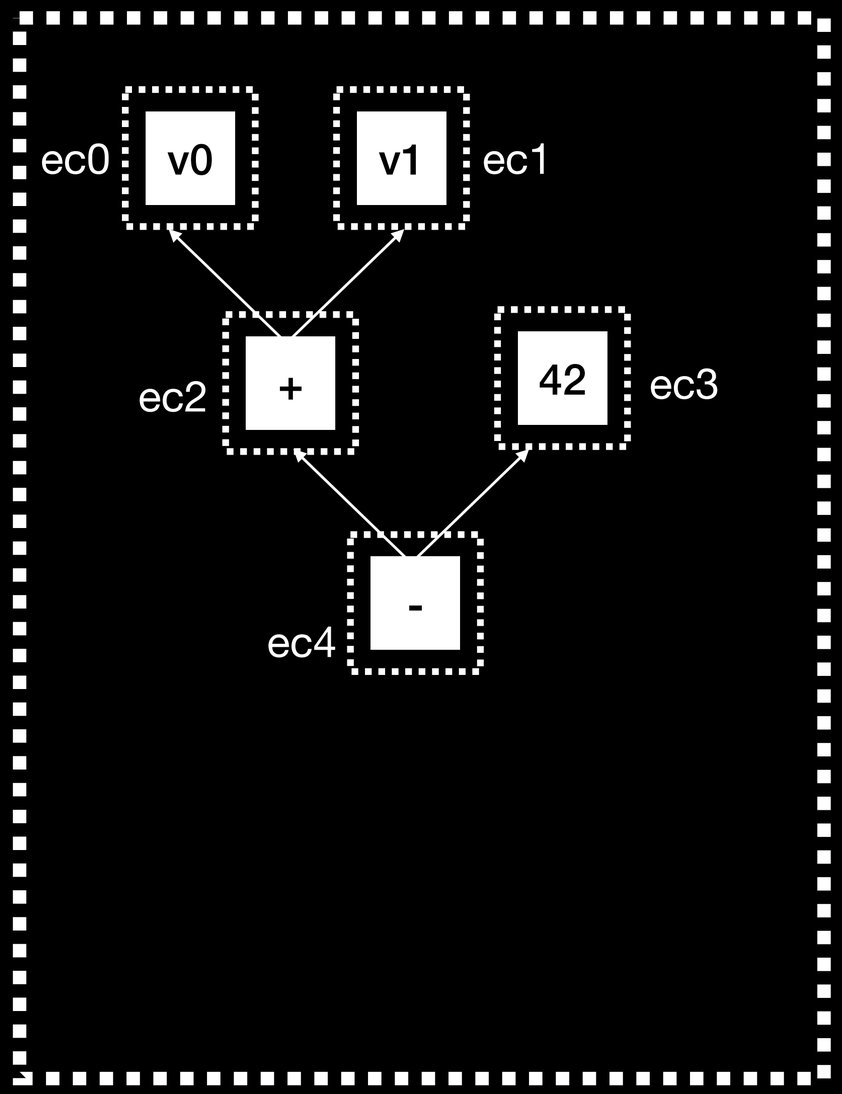


Elaboration



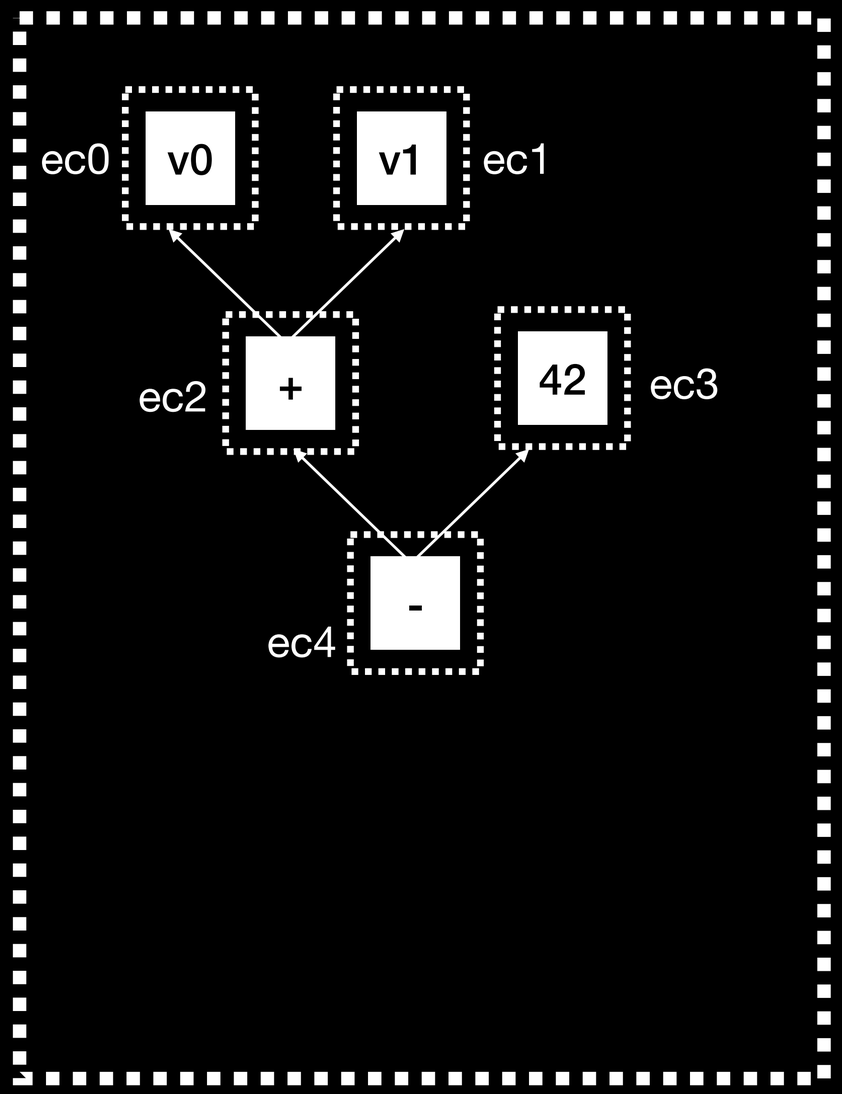
```
block0(v0, v1):
  store ec0, ec4
  return ec2
```

eclass	elaborated
ec0	v0
ec1	v1



Elaboration

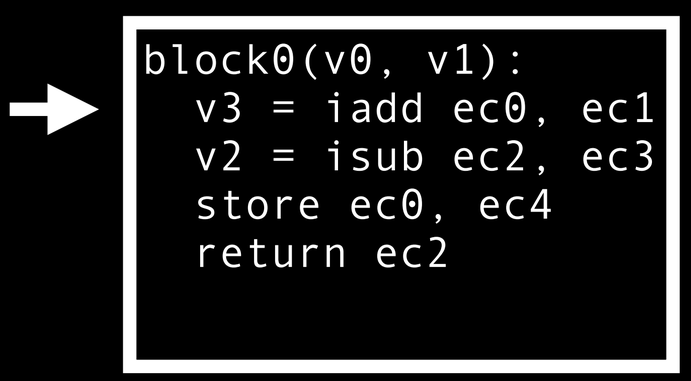
```
block0(v0, v1):
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
```



eclass	elaborated
ec0	v0
ec1	v1
ec4	v2

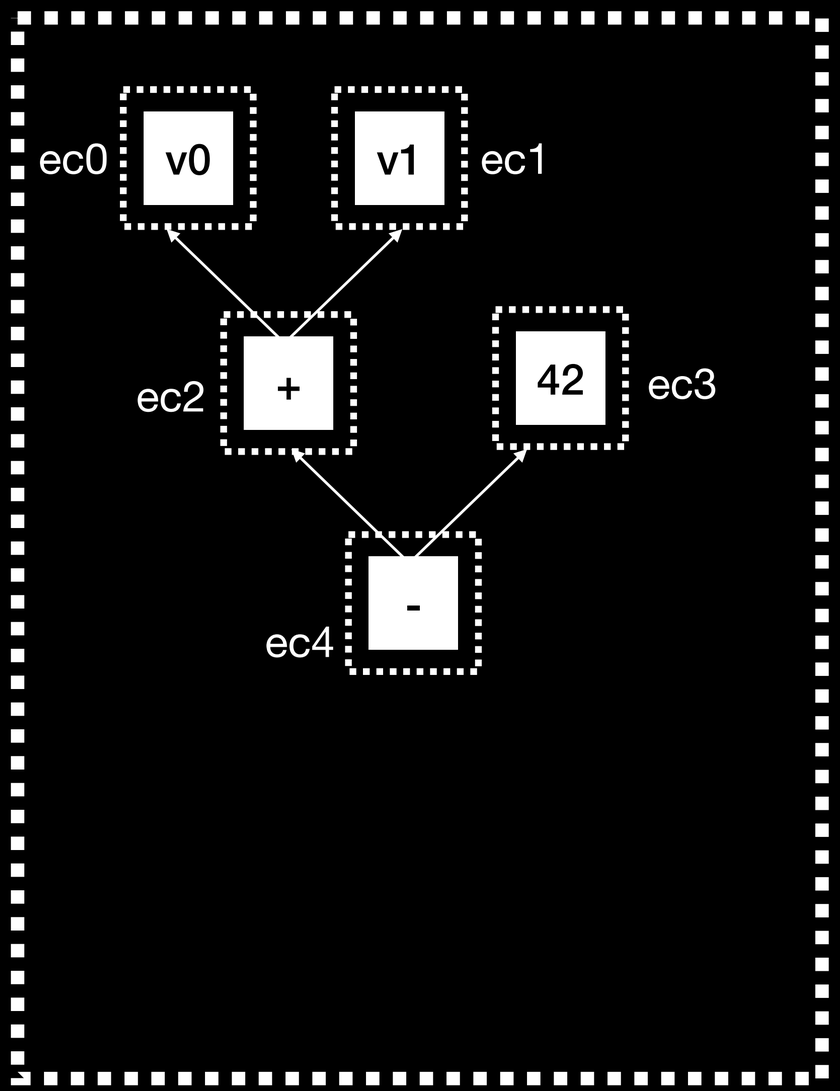
** Note: assume extraction (node selection) is done already!*

Elaboration

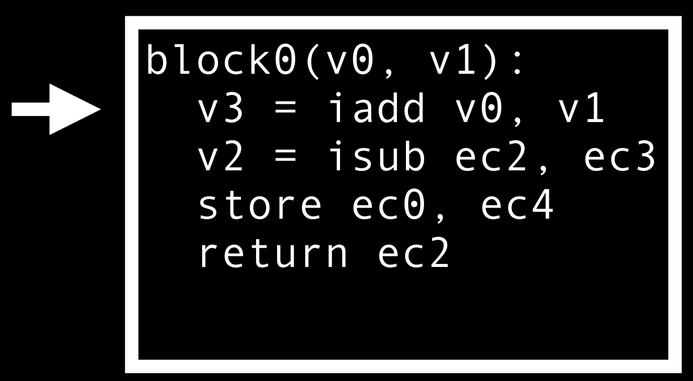


```
block0(v0, v1):
  v3 = iadd ec0, ec1
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
```

eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec4	v2

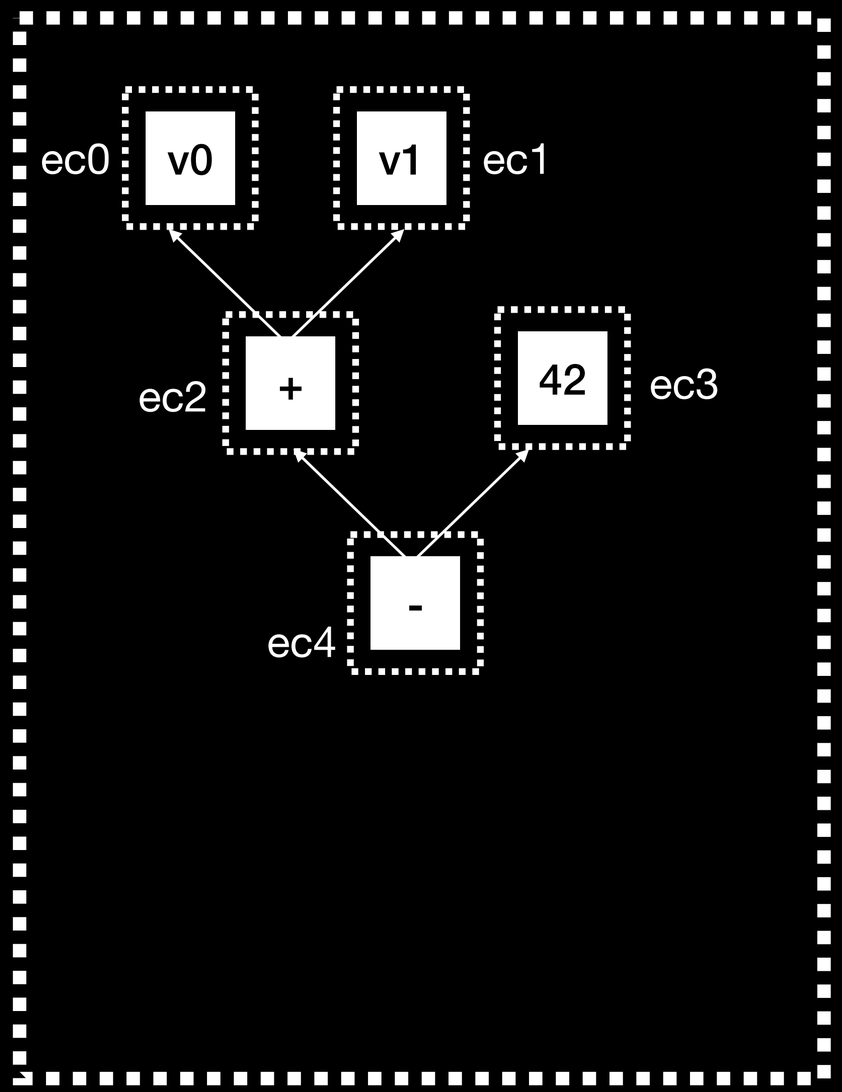


Elaboration

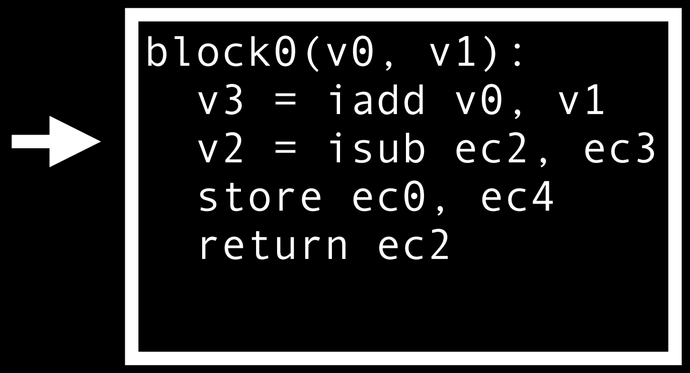


```
block0(v0, v1):
  v3 = iadd v0, v1
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
```

eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec4	v2

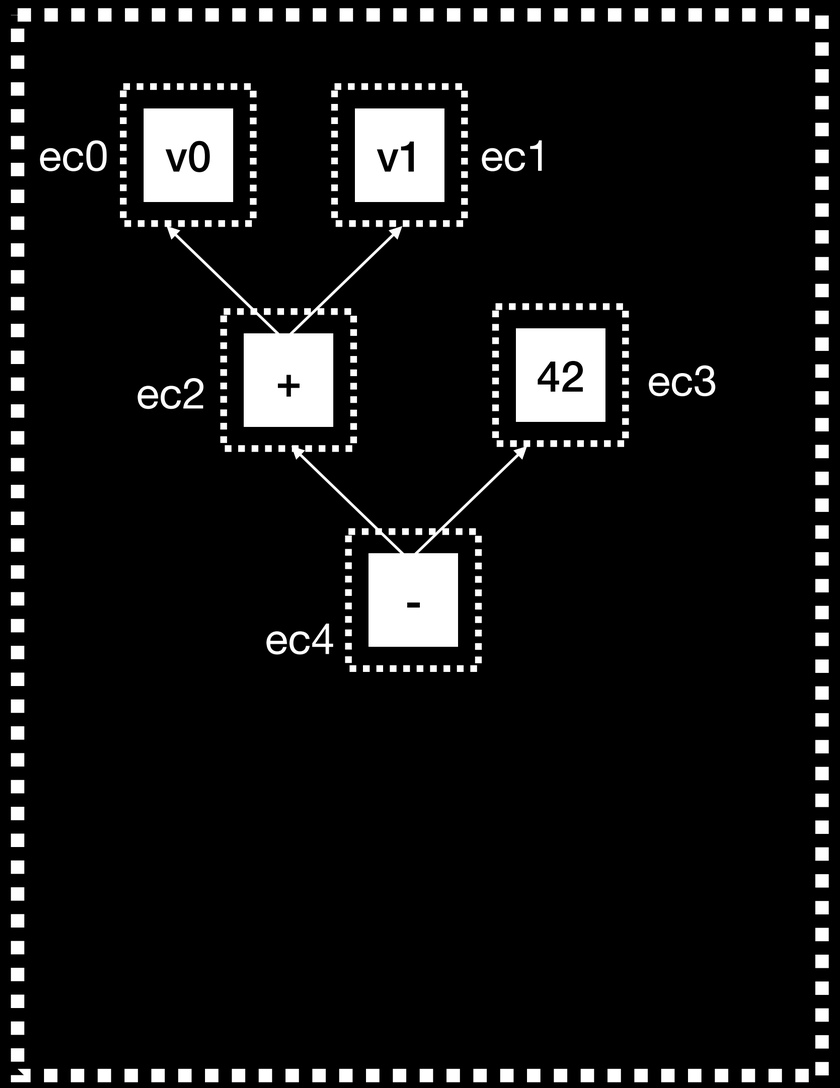


Elaboration



```
block0(v0, v1):
  v3 = iadd v0, v1
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
```

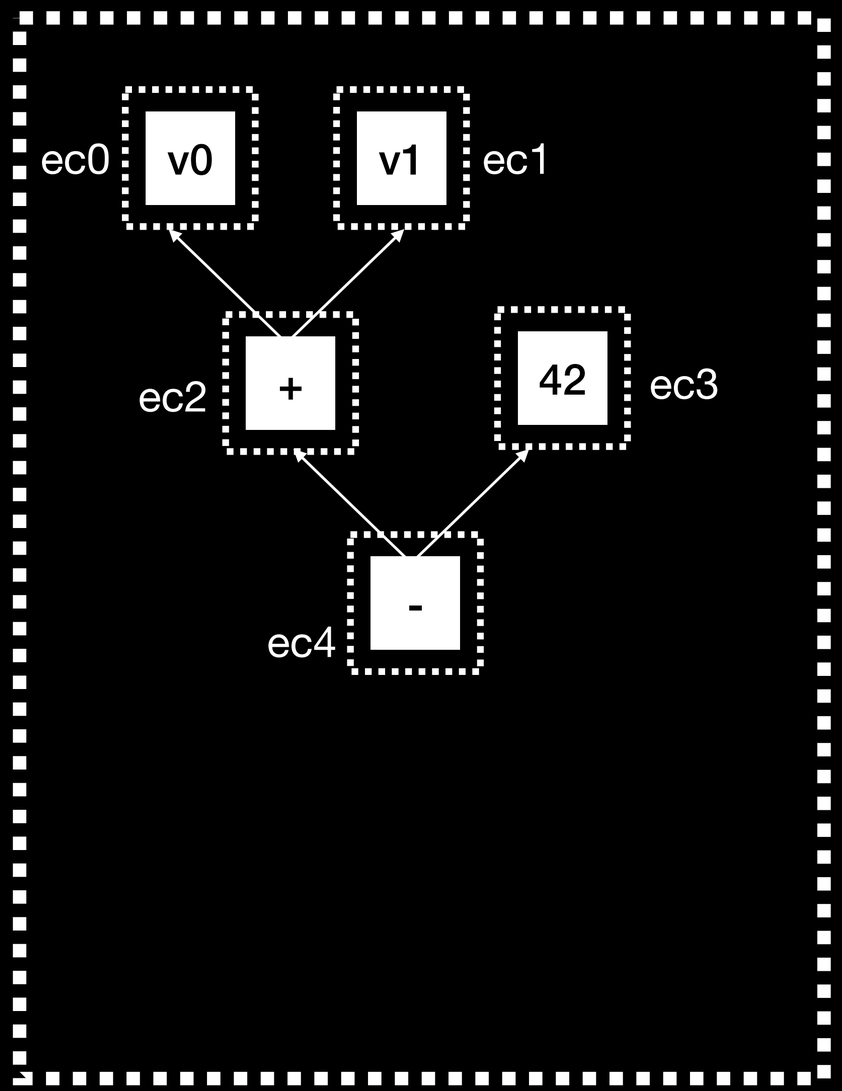
eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec4	v2



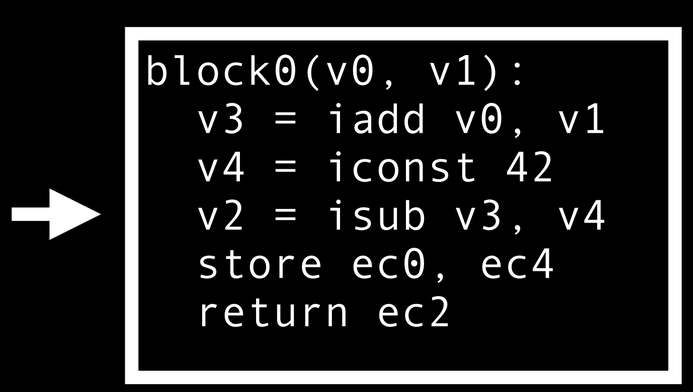
Elaboration

```
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
```

eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec3	v4
ec4	v2

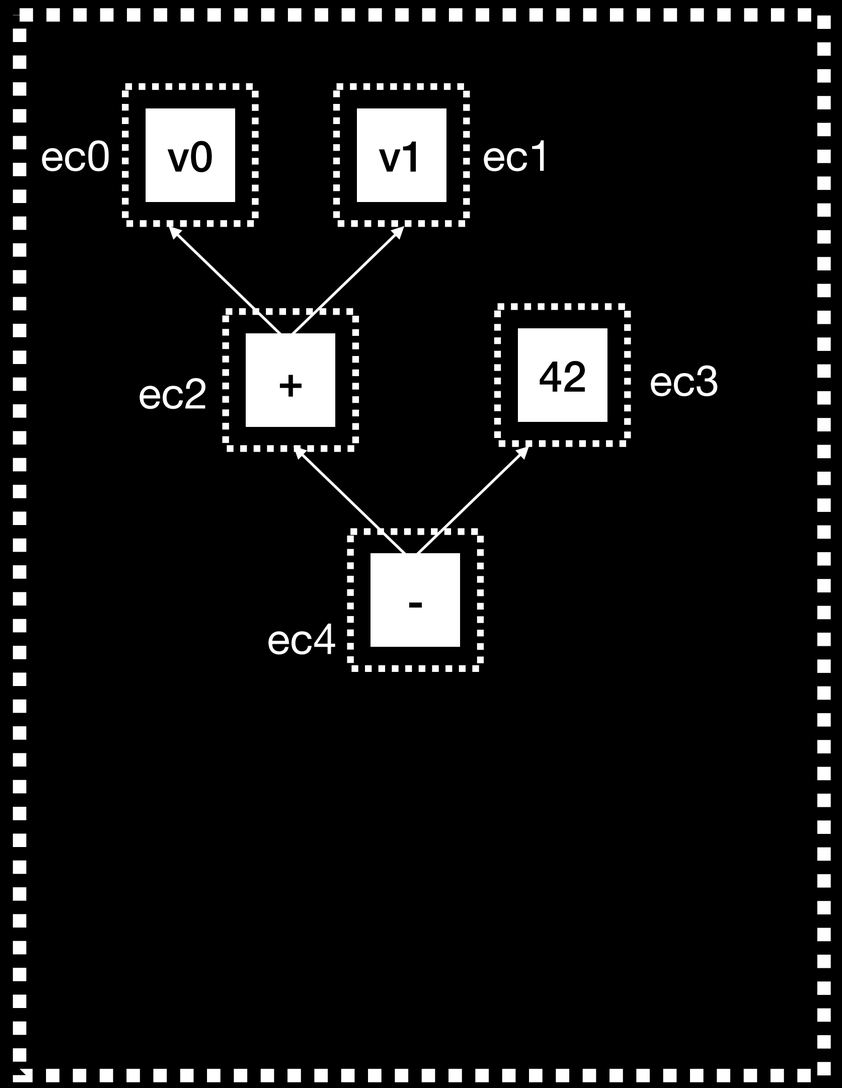


Elaboration

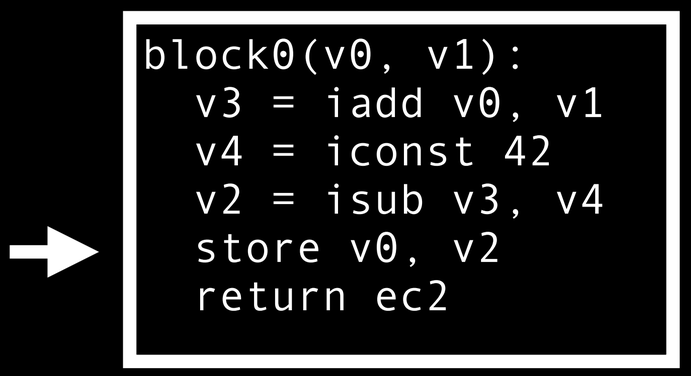


```
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub v3, v4
  store ec0, ec4
  return ec2
```

eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec3	v4
ec4	v2

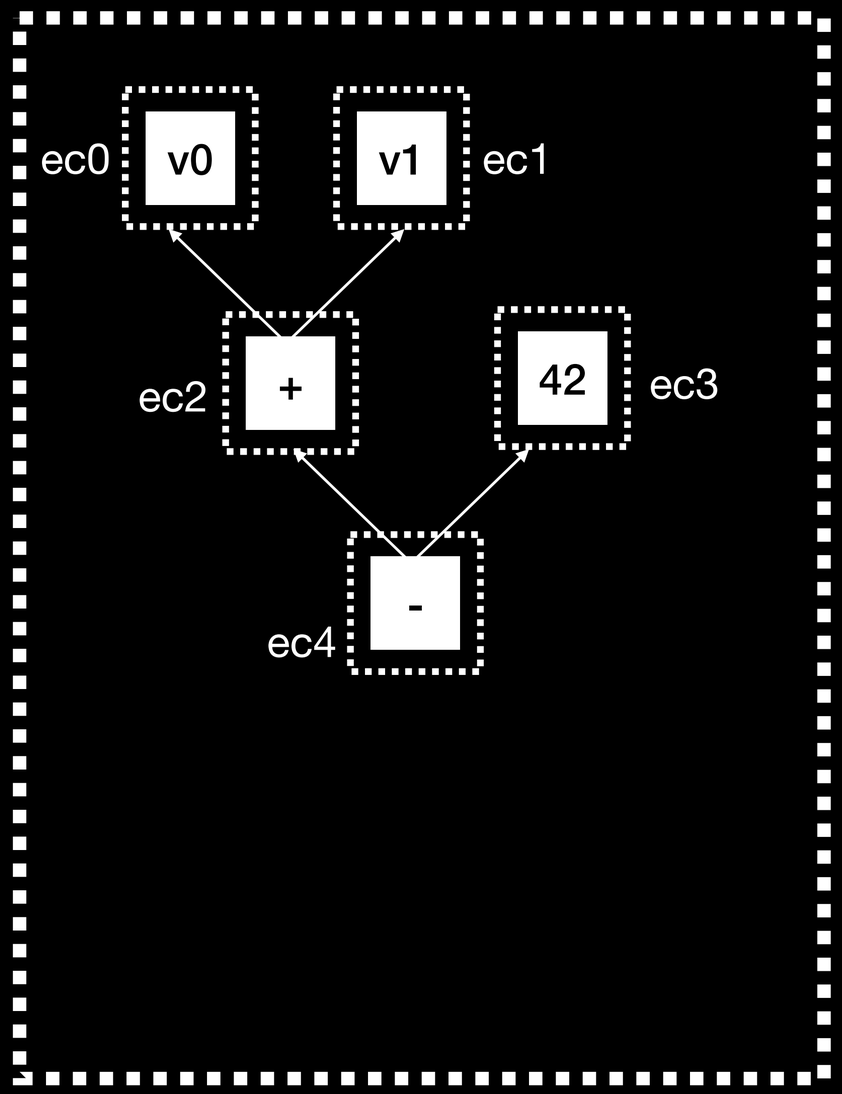


Elaboration

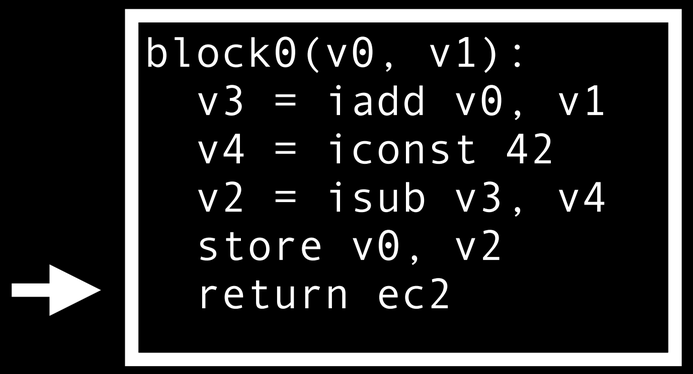


```
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub v3, v4
  store v0, v2
  return ec2
```

eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec3	v4
ec4	v2

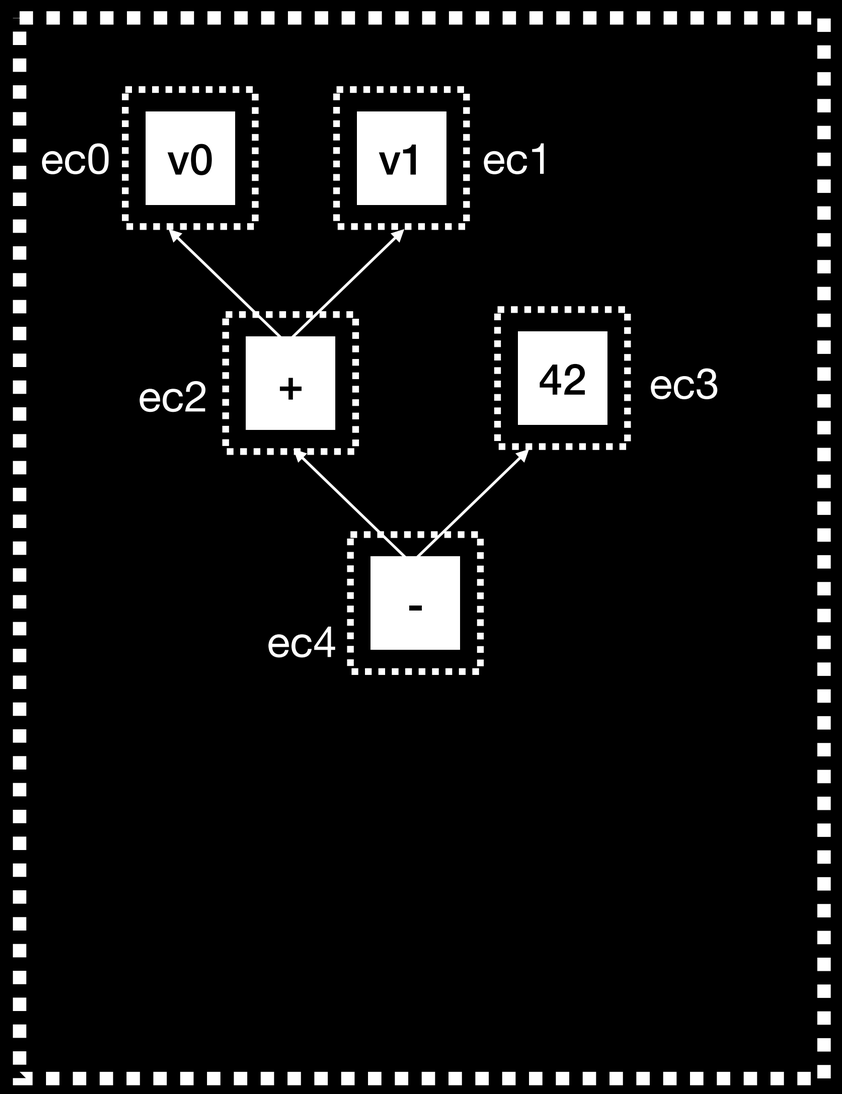


Elaboration

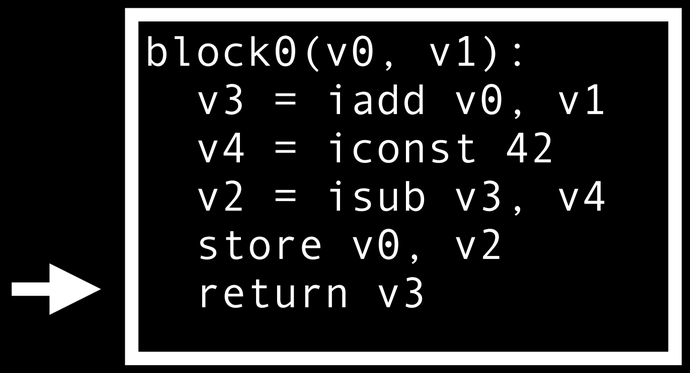


```
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub v3, v4
  store v0, v2
  return ec2
```

eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec3	v4
ec4	v2

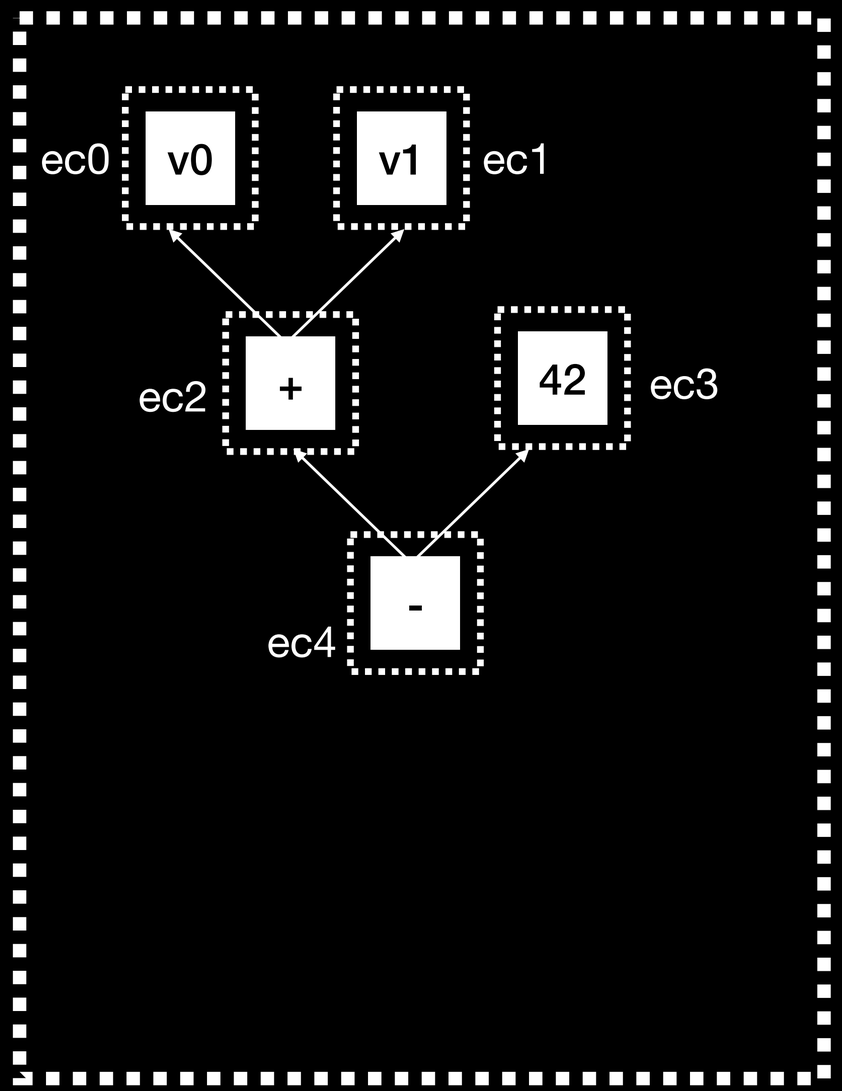


Elaboration

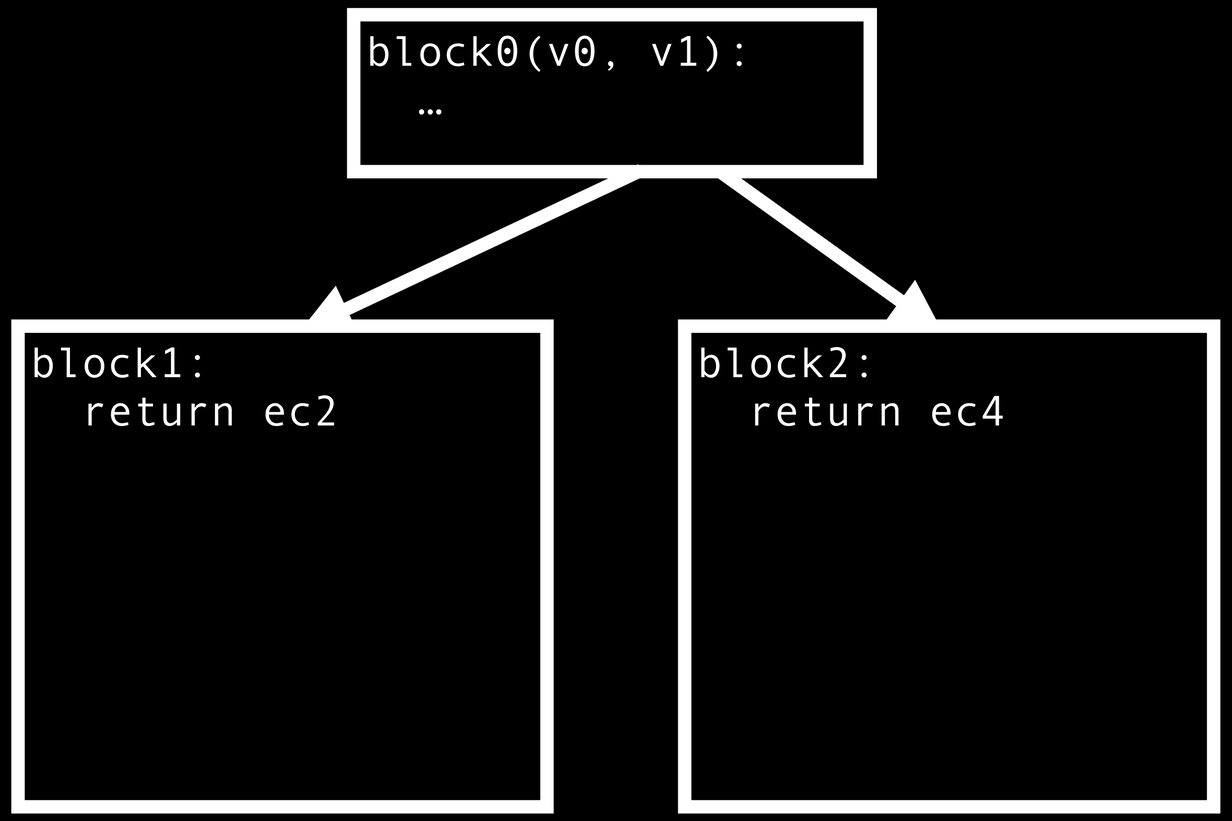


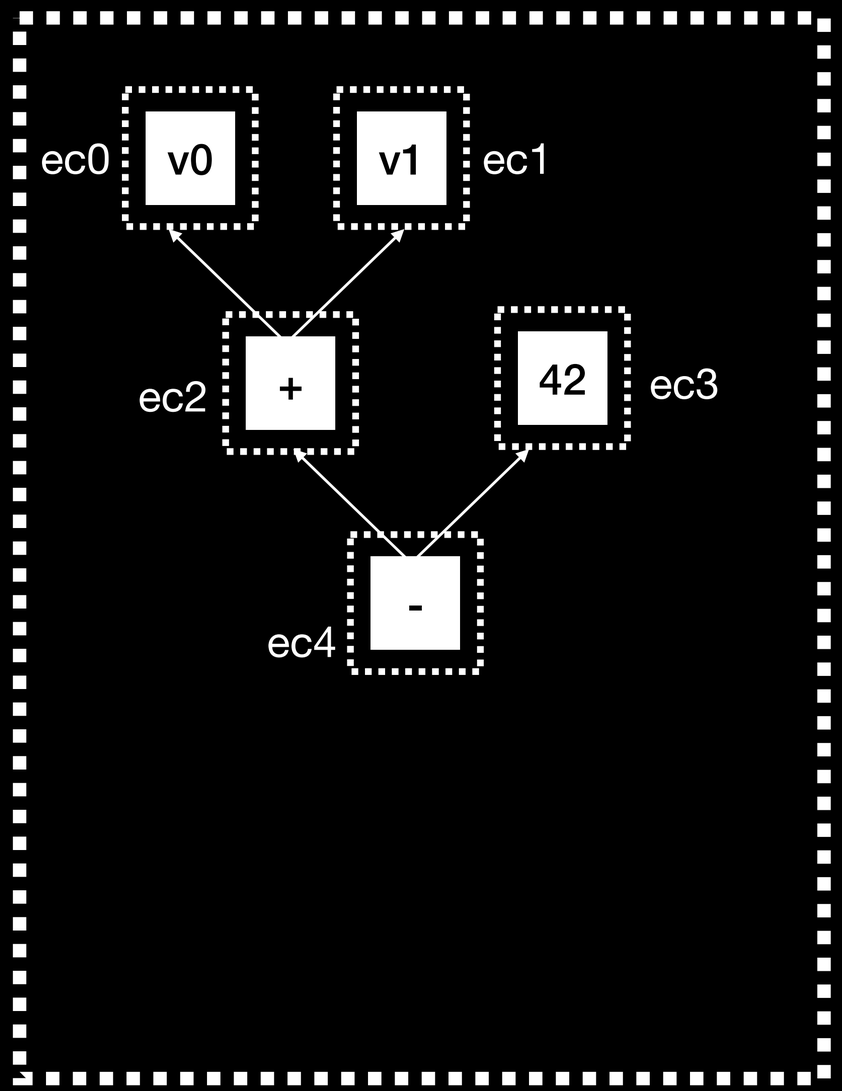
```
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub v3, v4
  store v0, v2
  return v3
```

eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec3	v4
ec4	v2

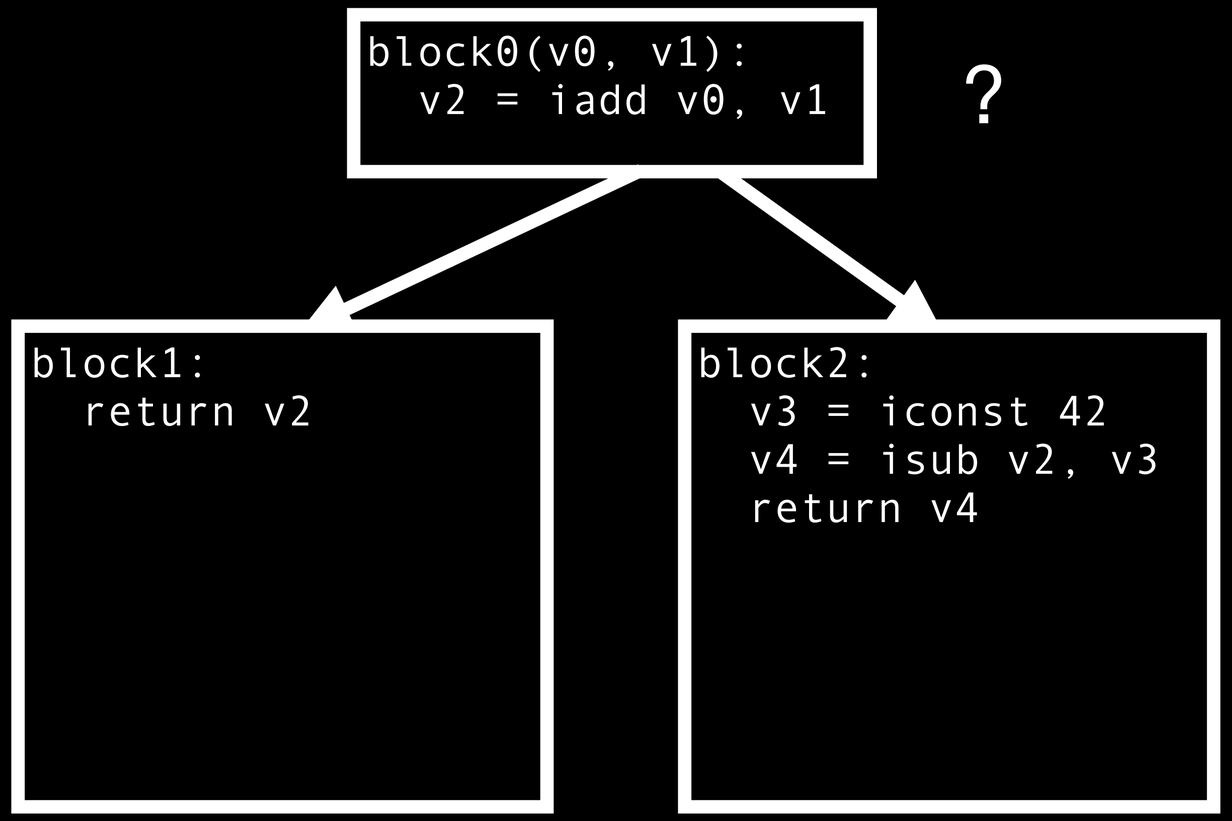


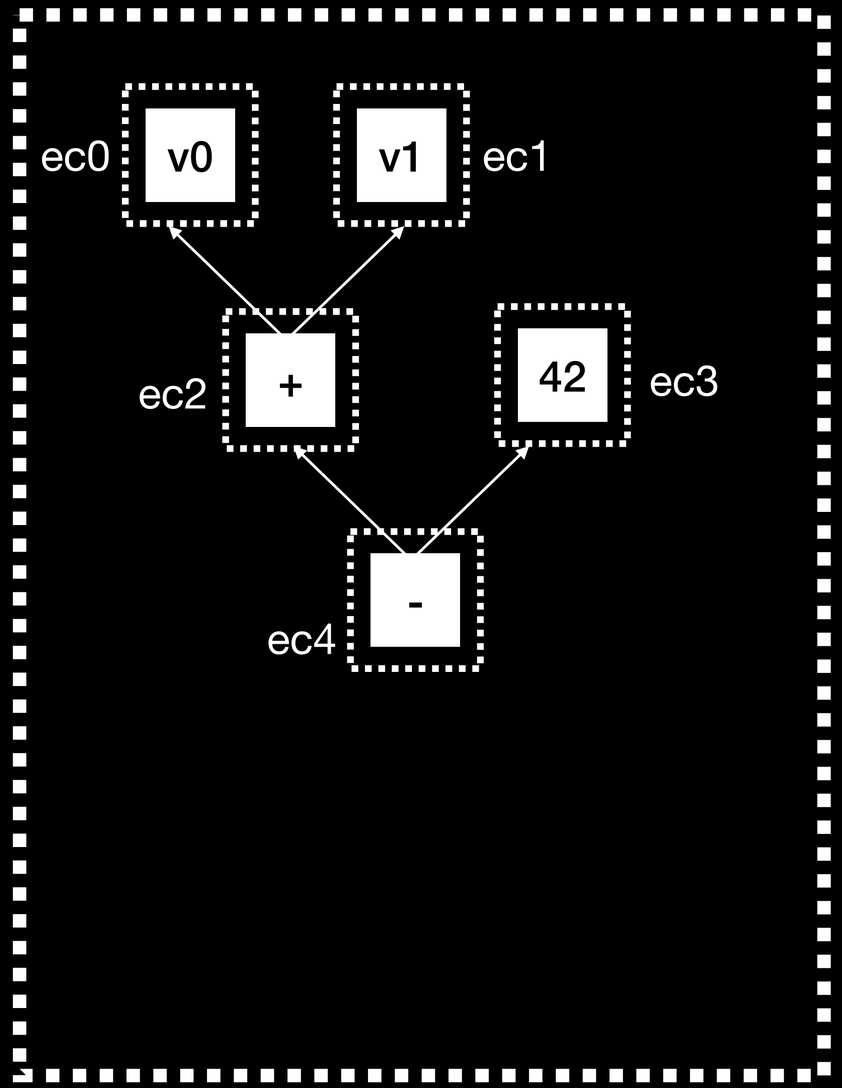
Elaboration… twice?



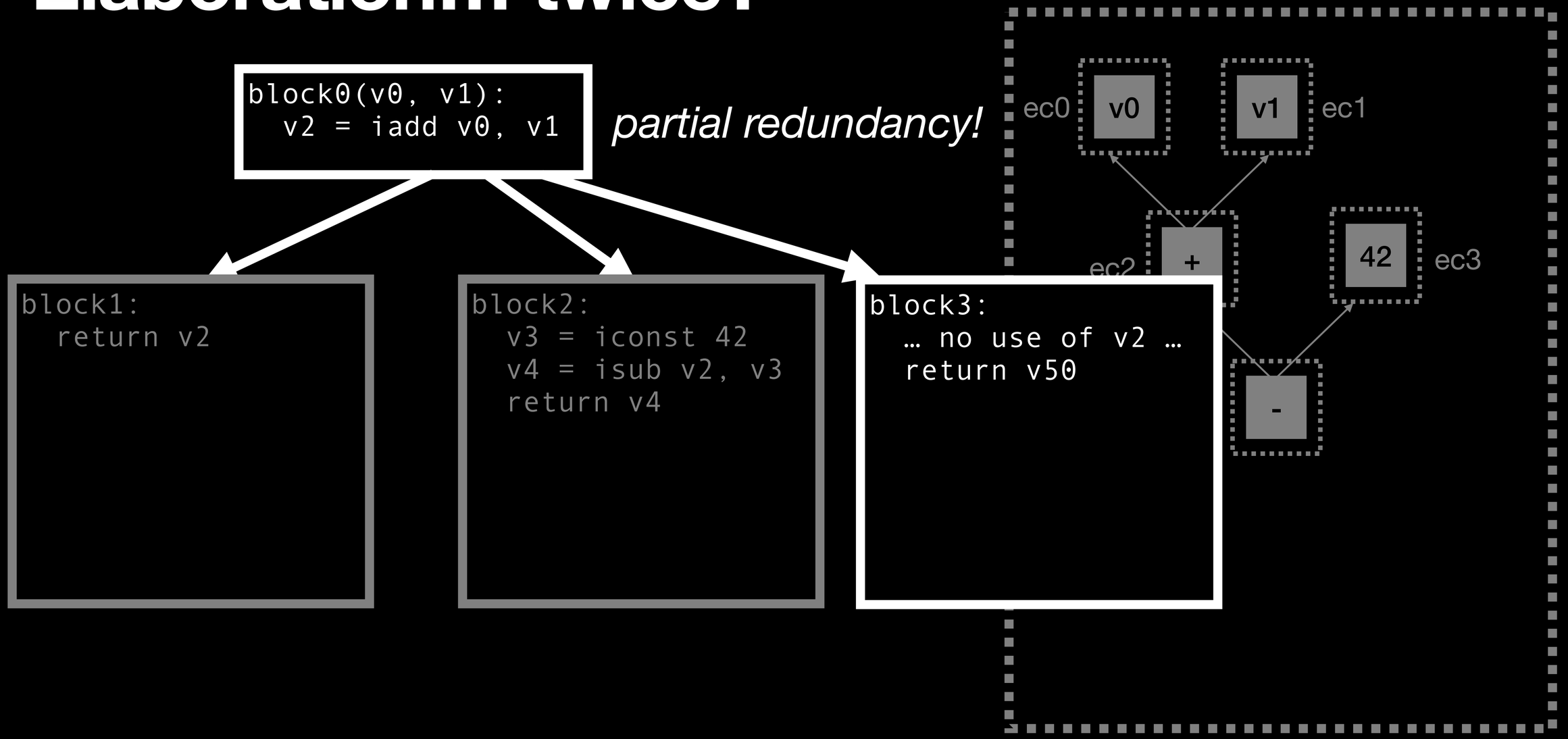


Elaboration… twice?

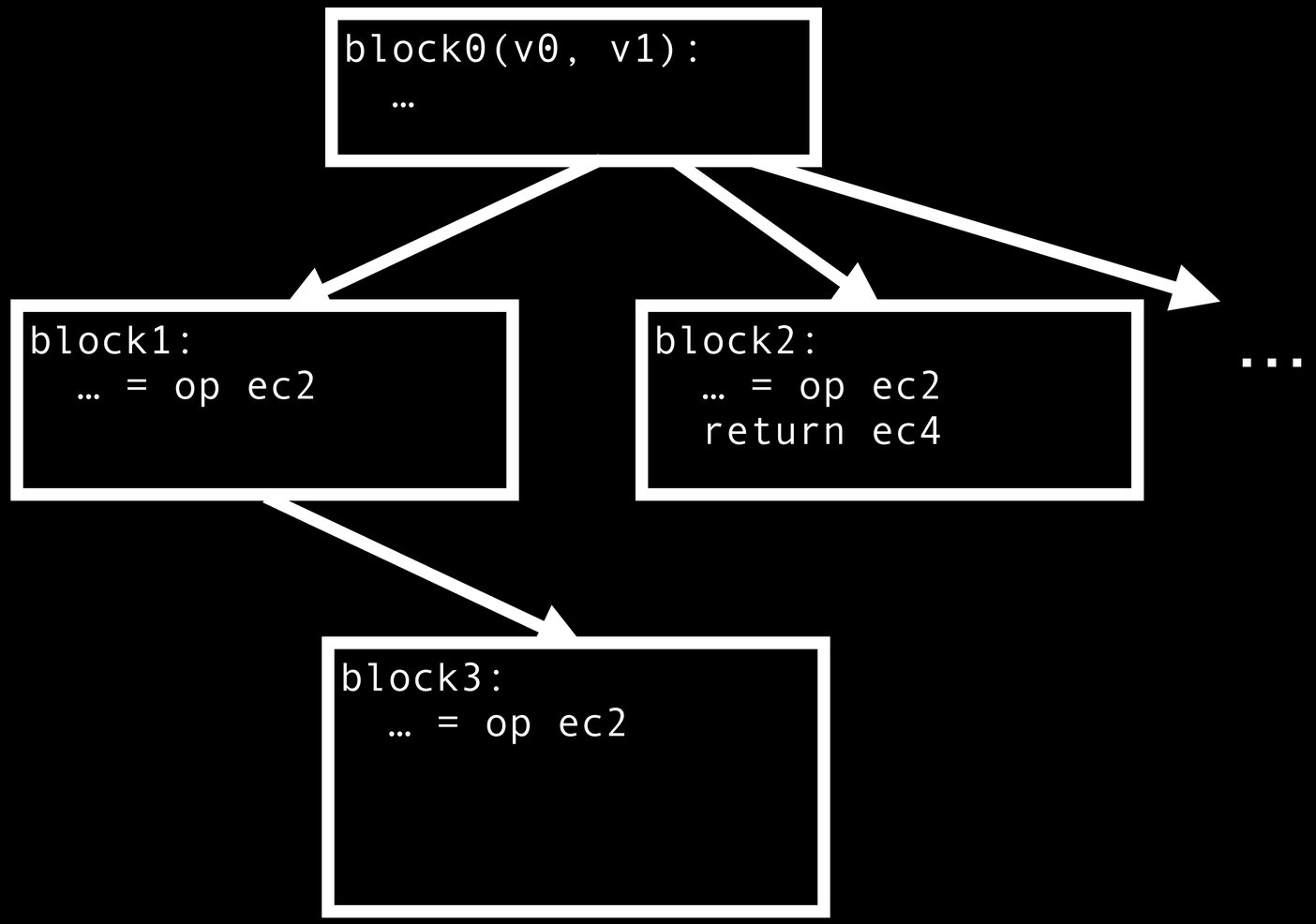




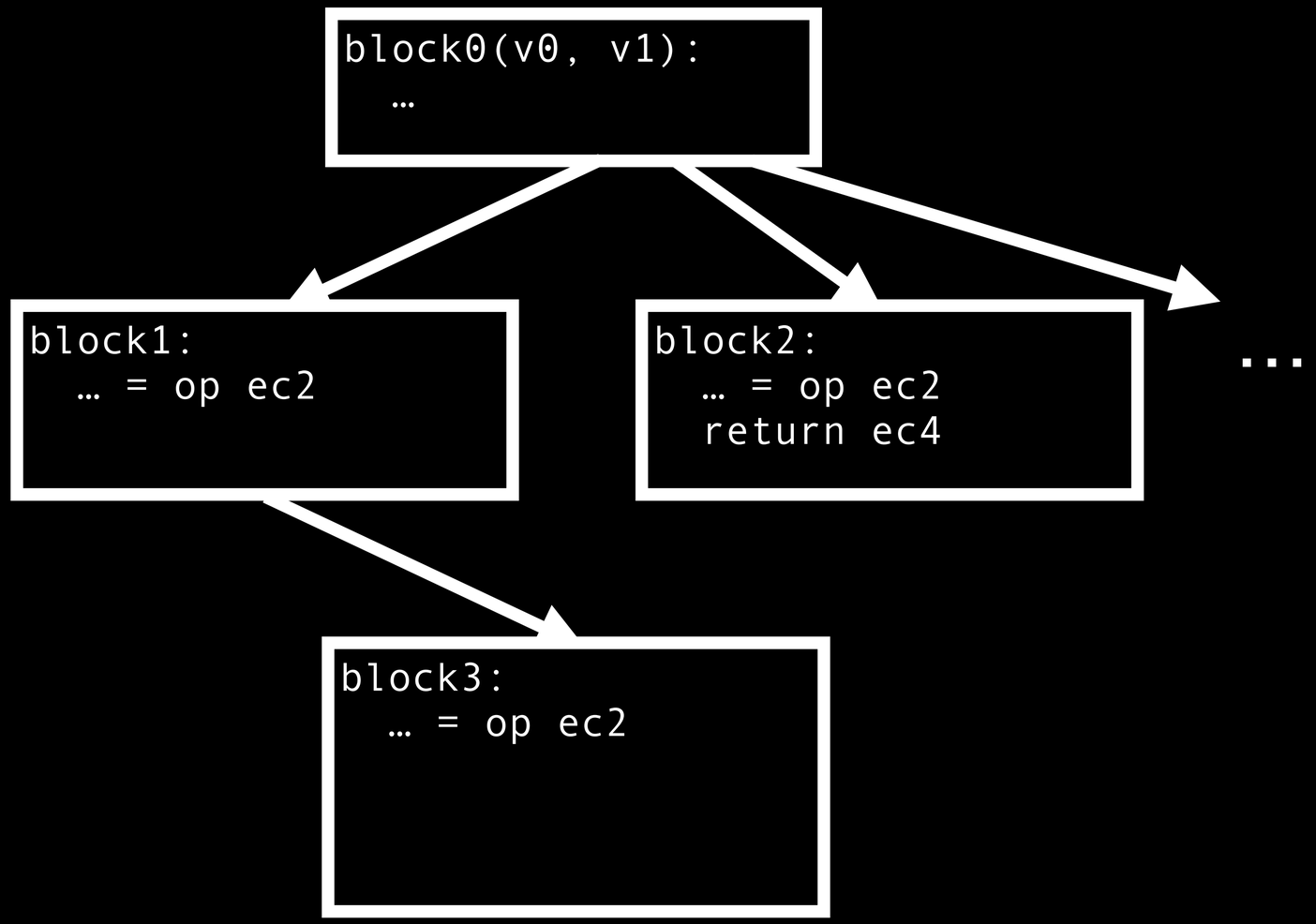
Elaboration… twice?

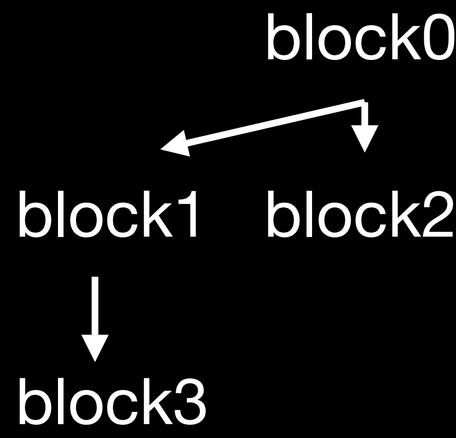


Scoped Elaboration

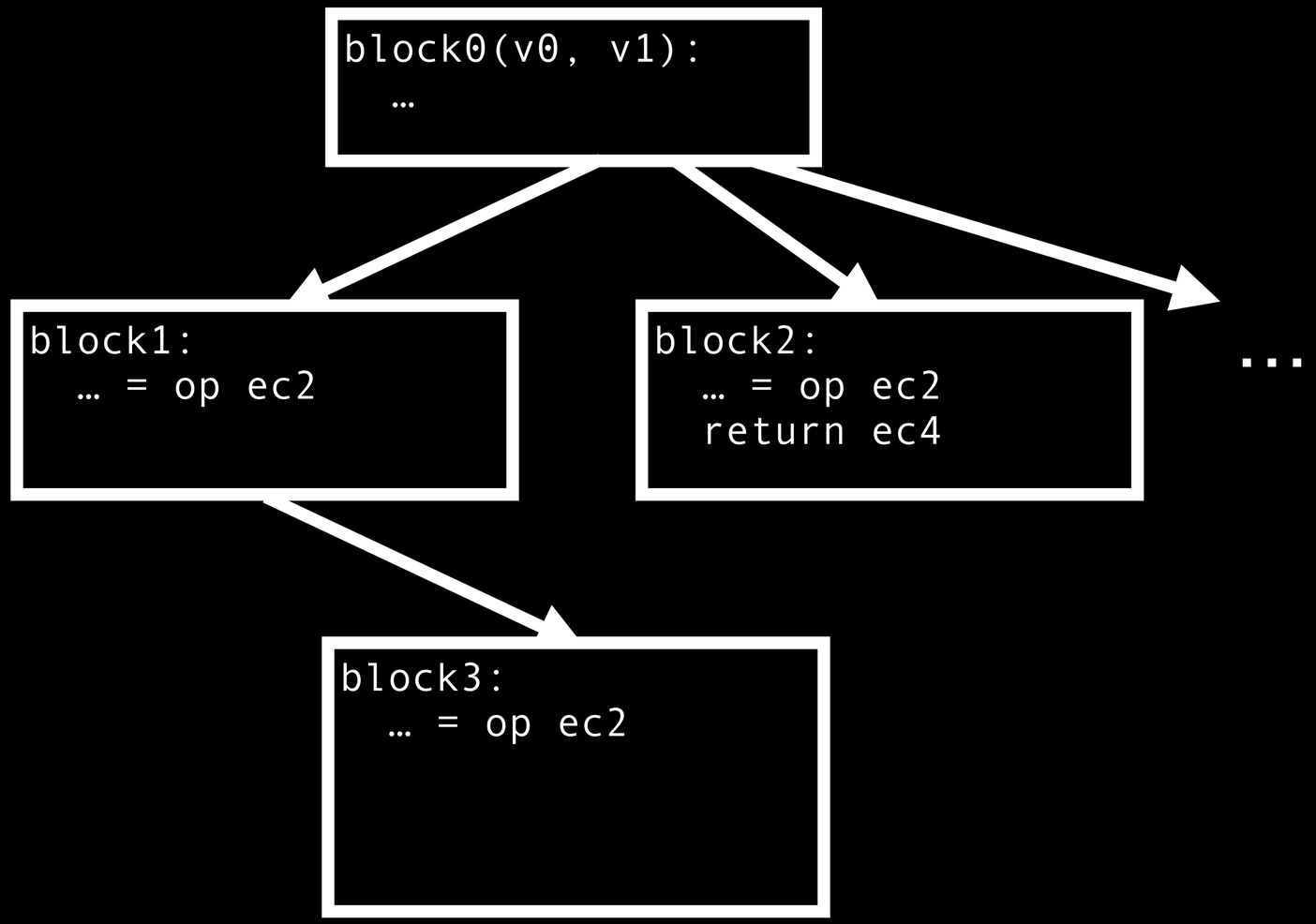


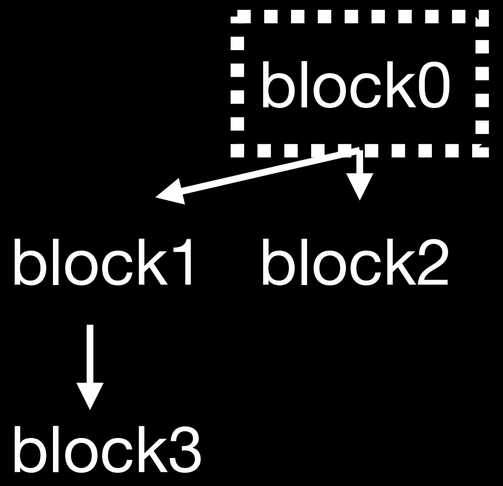
Scoped Elaboration



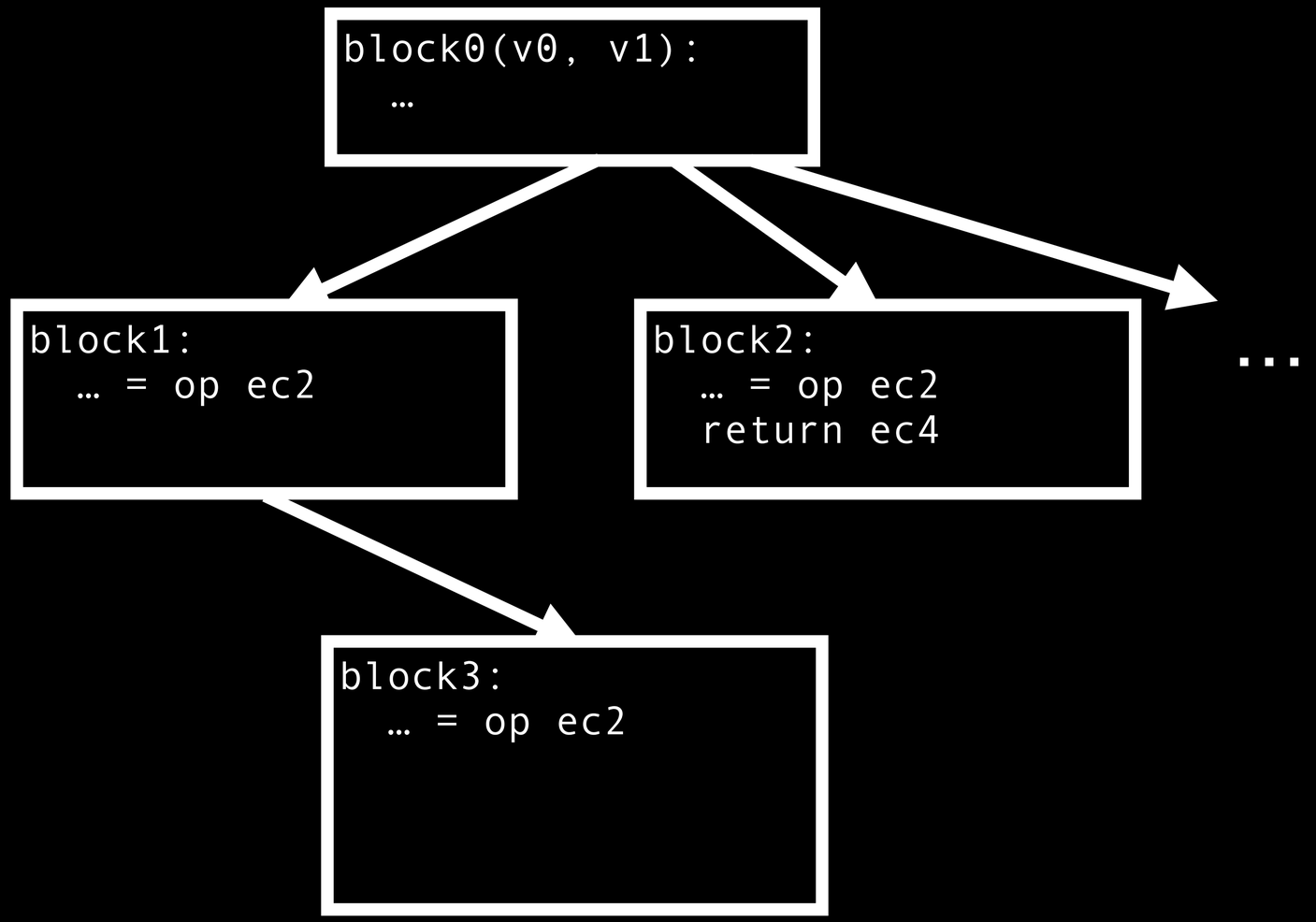


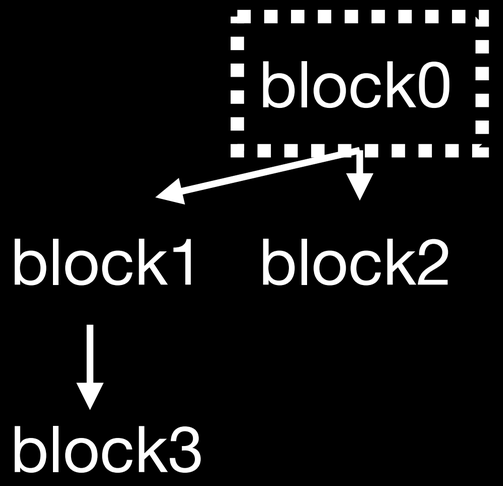
Scoped Elaboration





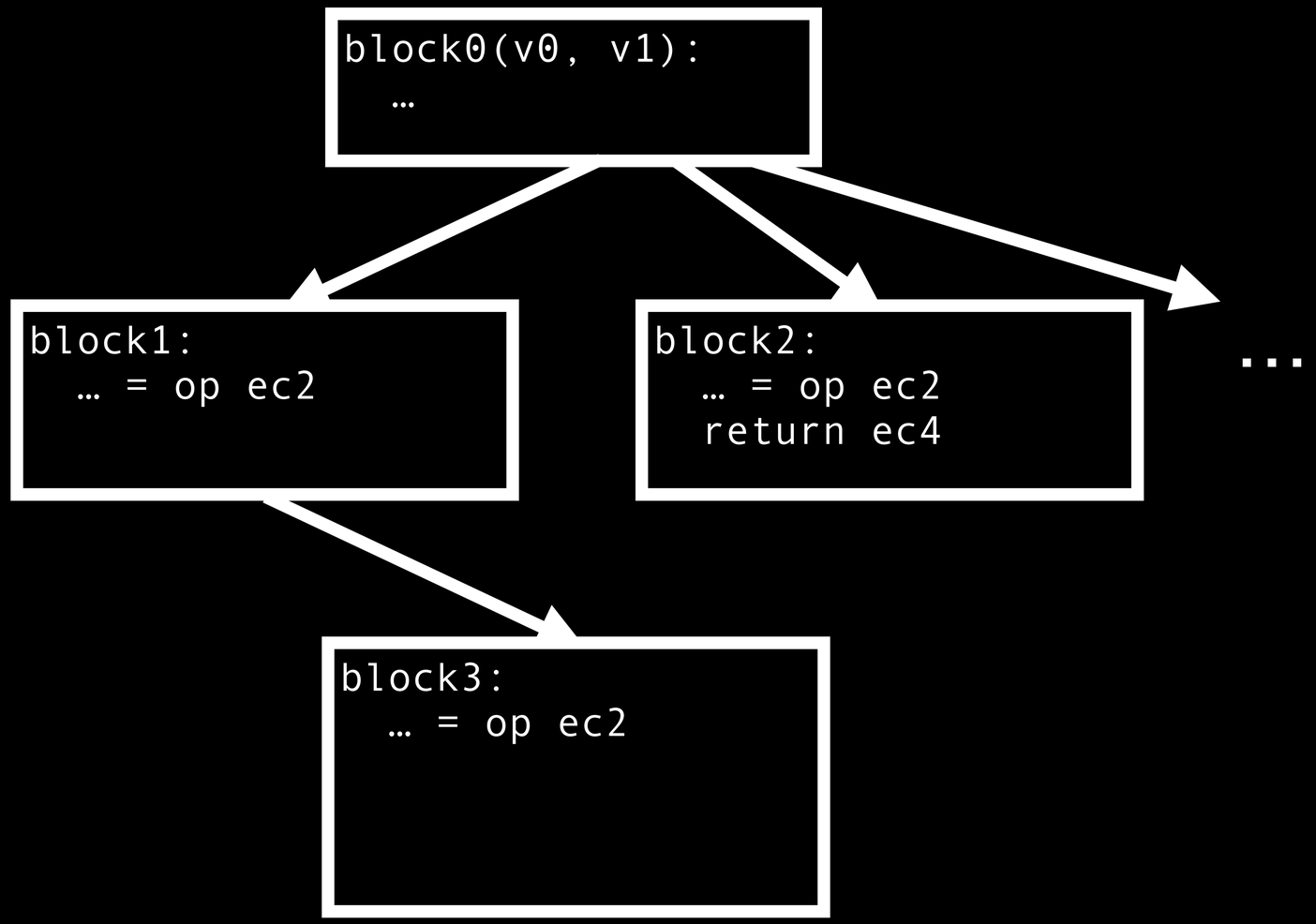
Scoped Elaboration

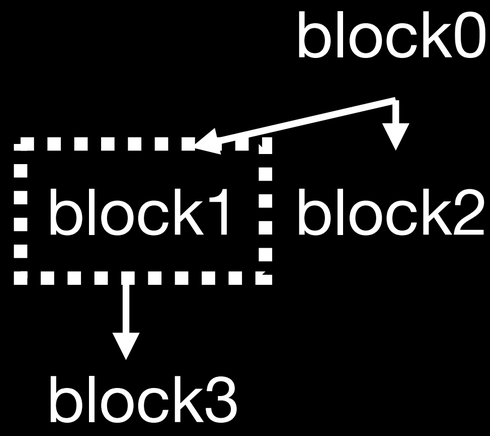




eclass	elaborated
ec0	v0
ec1	v1

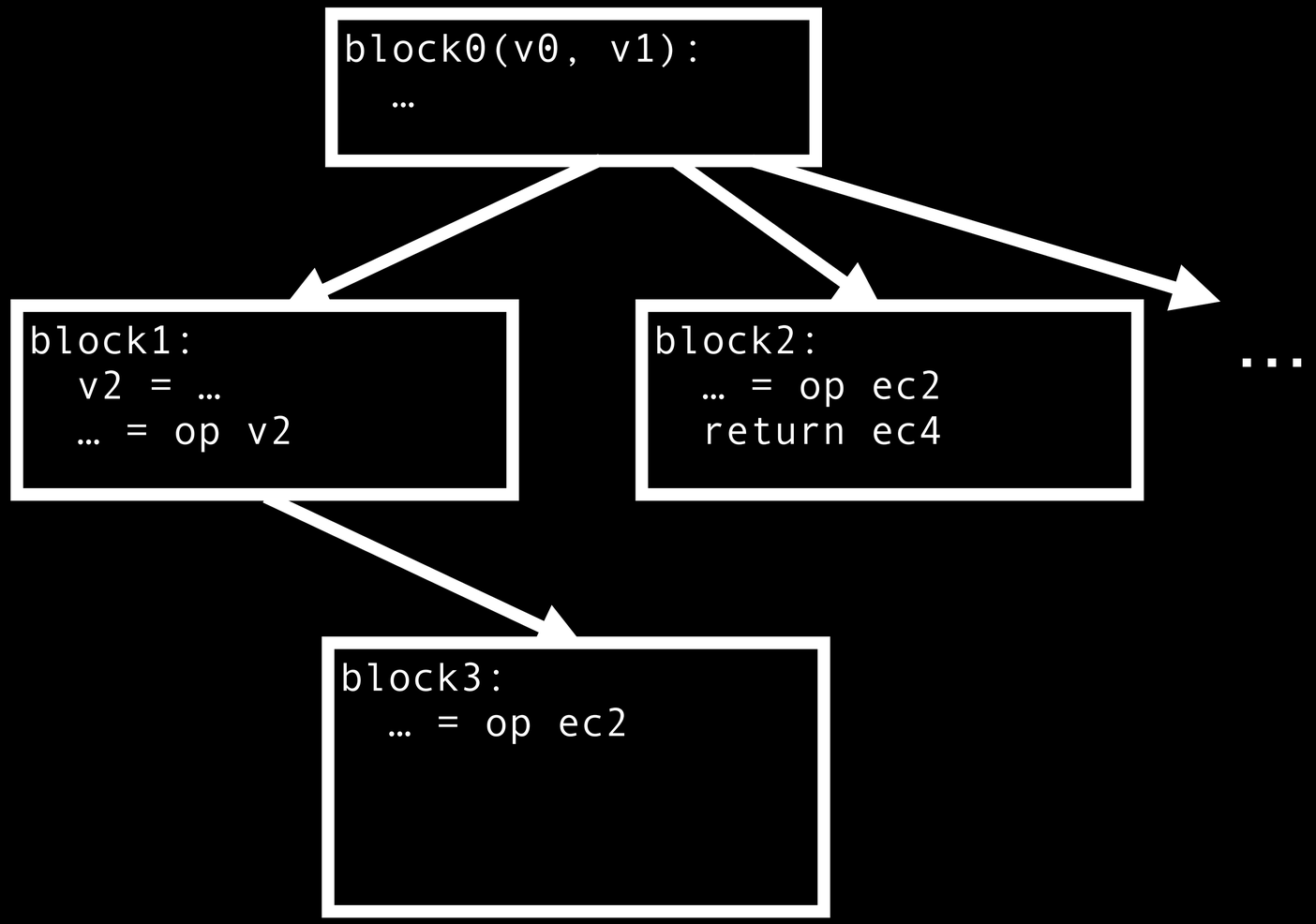
Scoped Elaboration

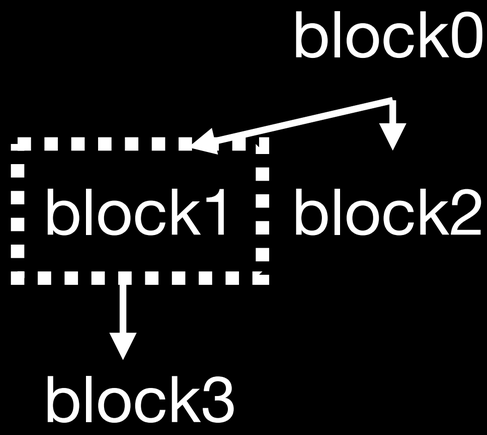




eclass	elaborated
ec0	v0
ec1	v1

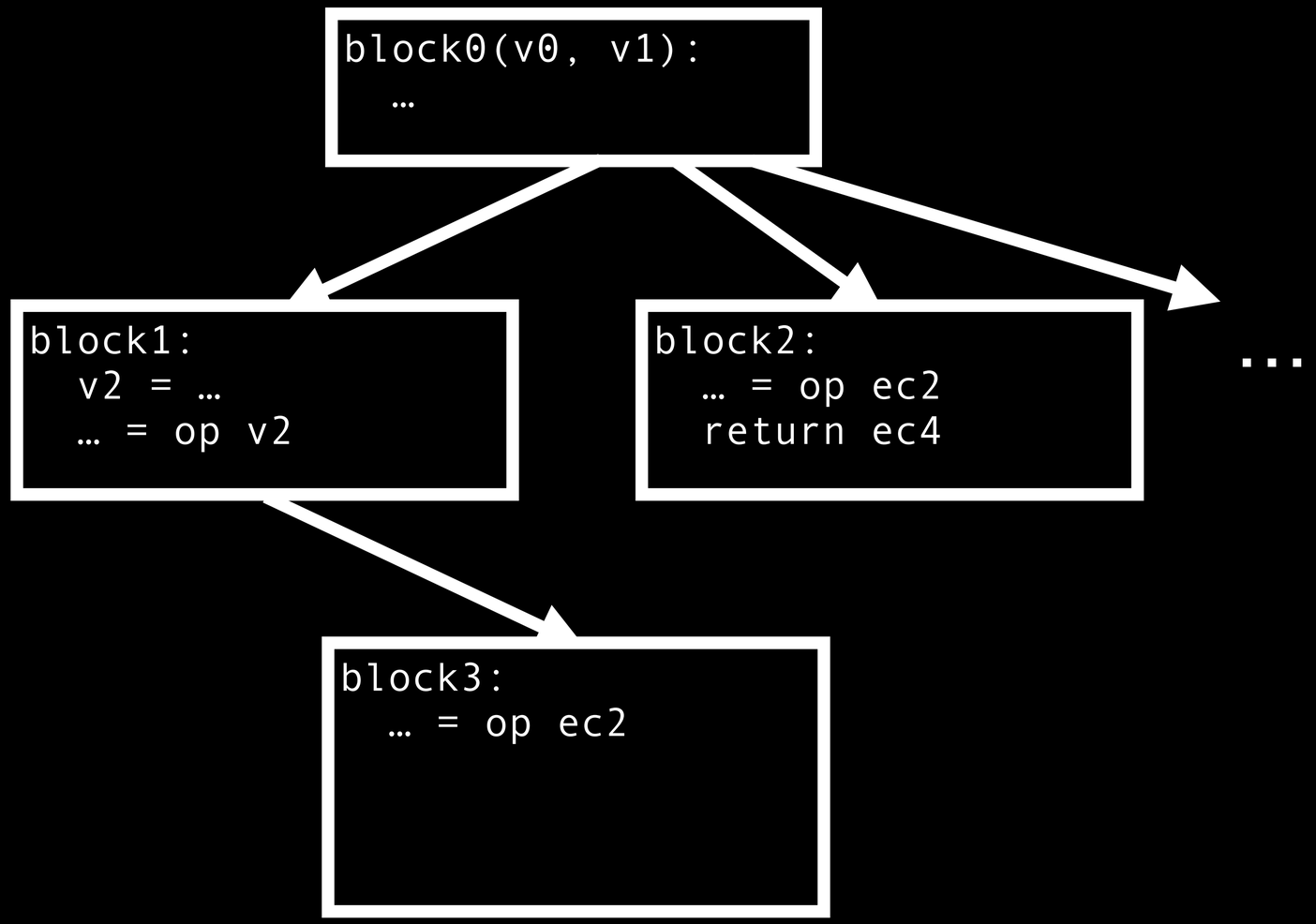
Scoped Elaboration

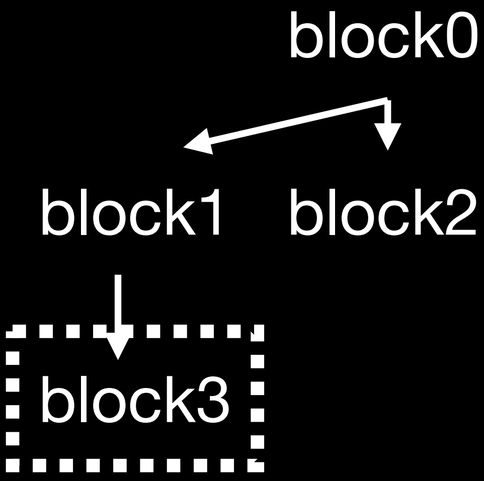




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

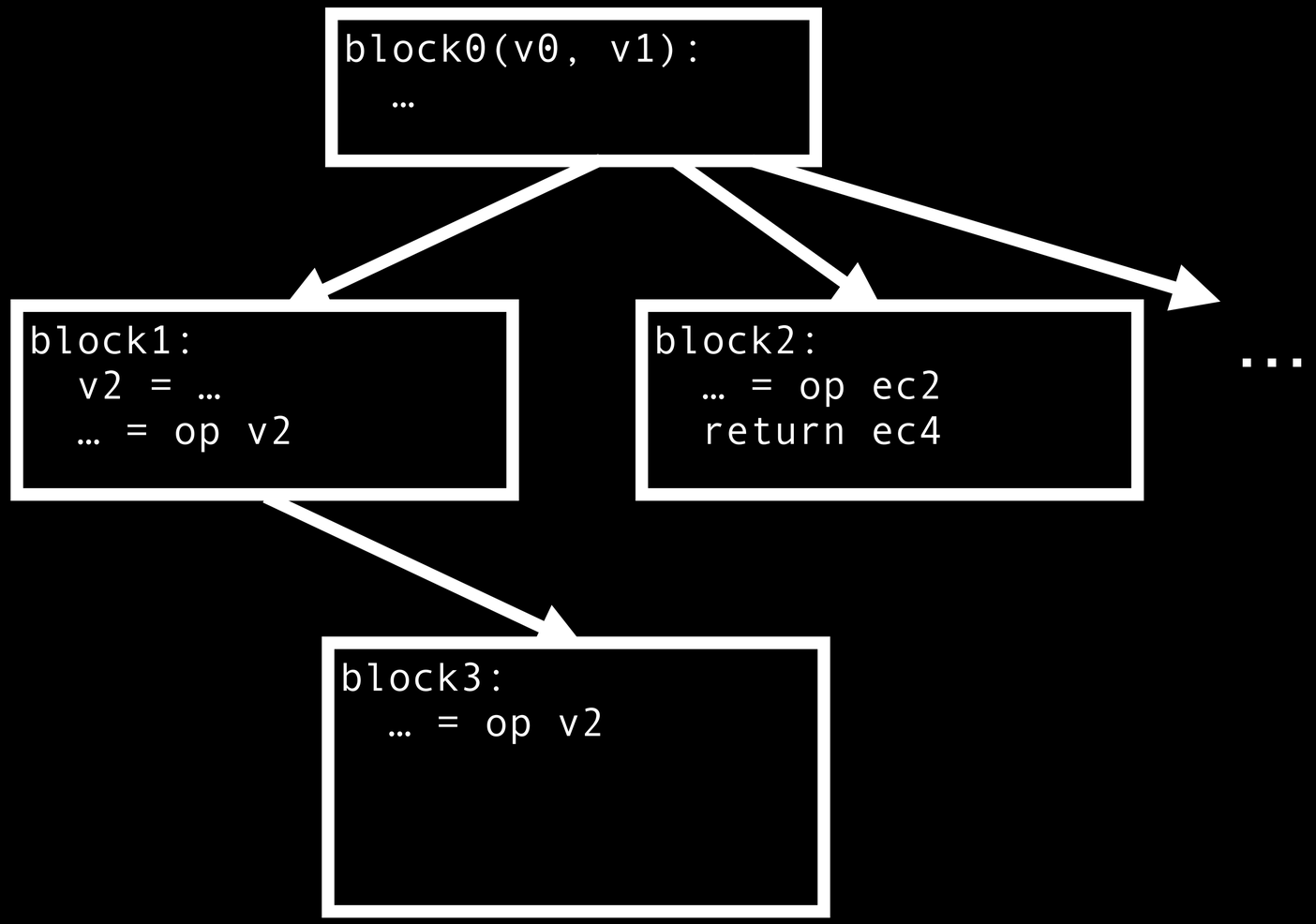
Scoped Elaboration

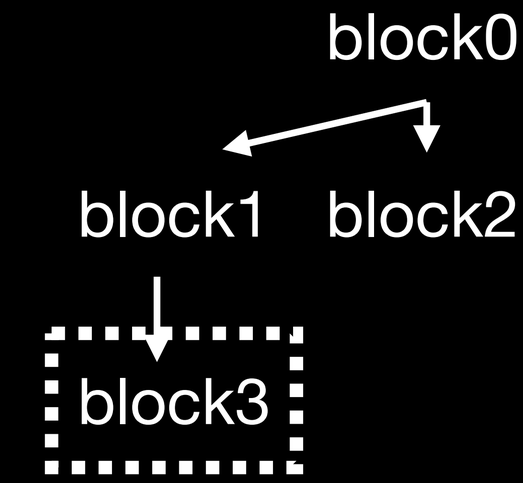




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

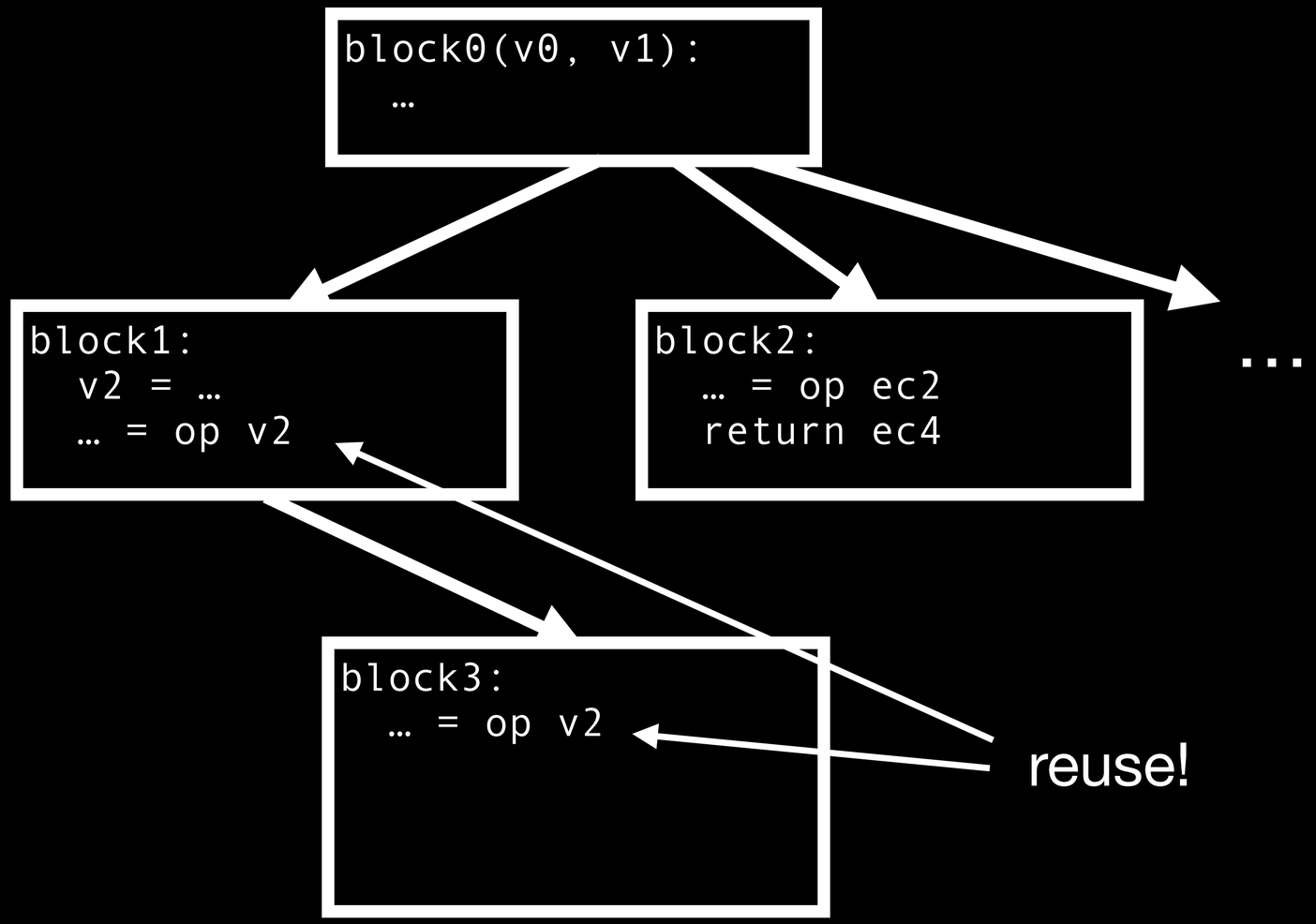
Scoped Elaboration

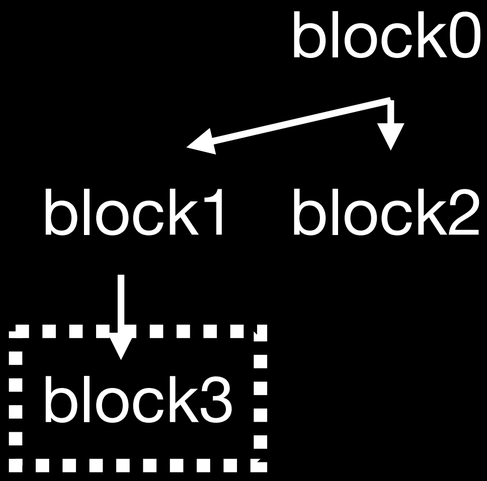




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

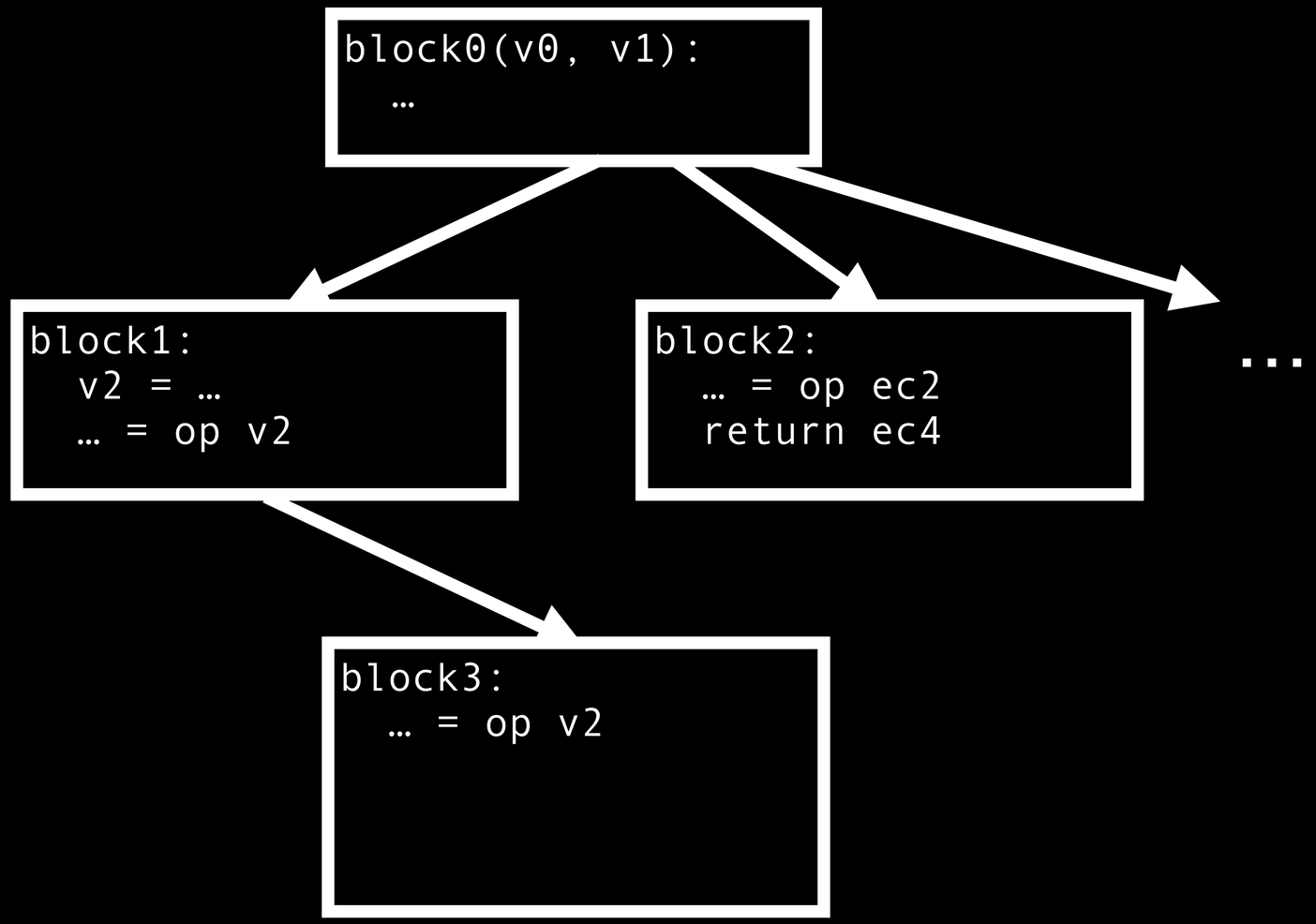
Scoped Elaboration

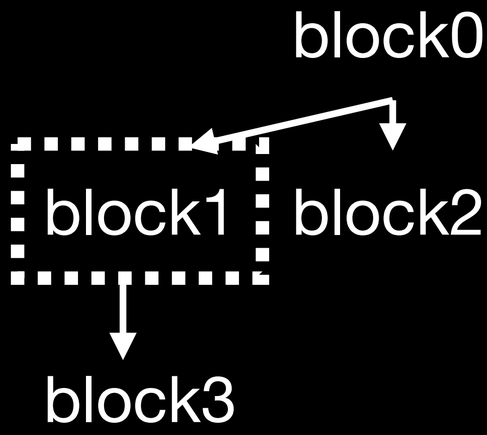




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

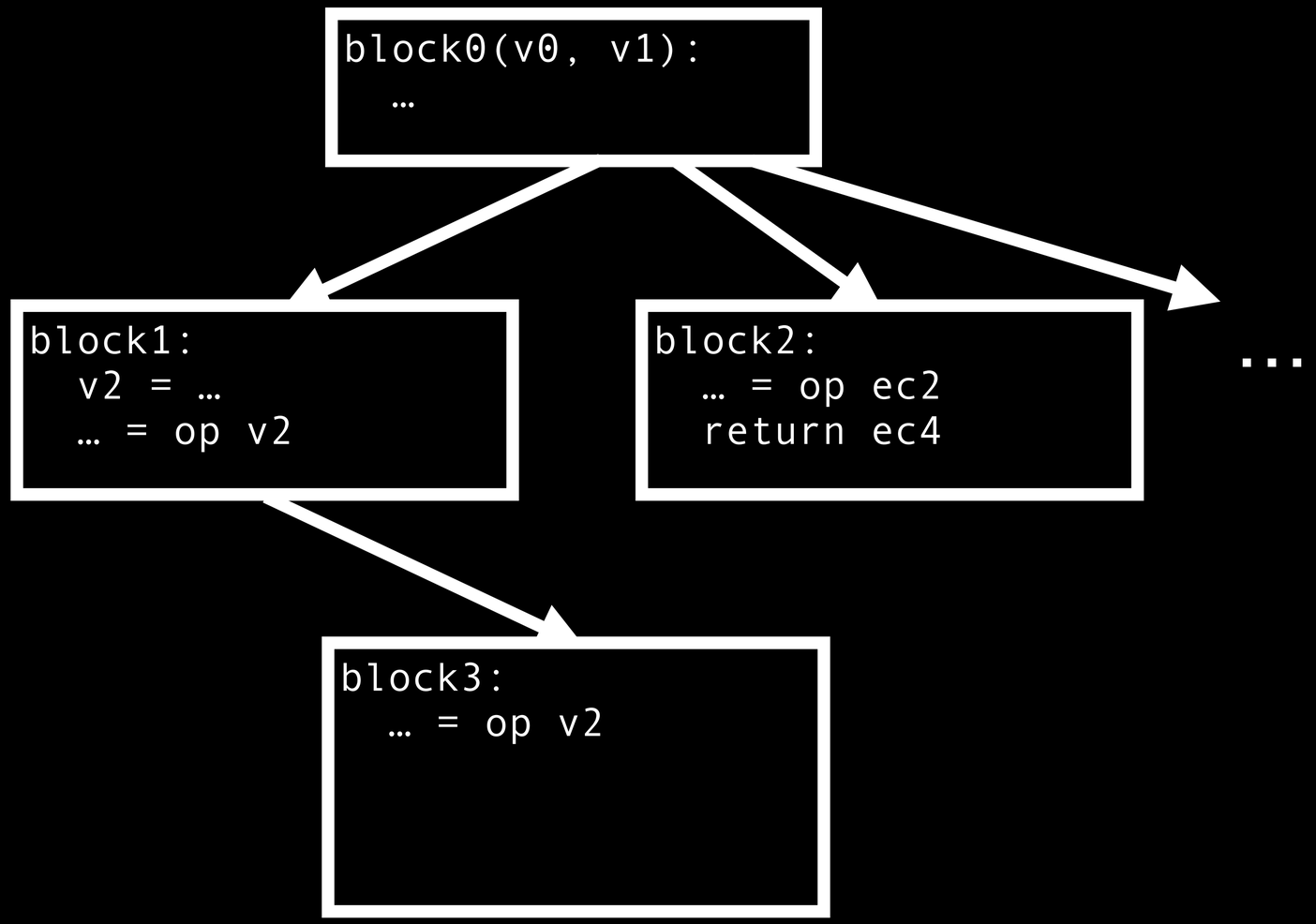
Scoped Elaboration

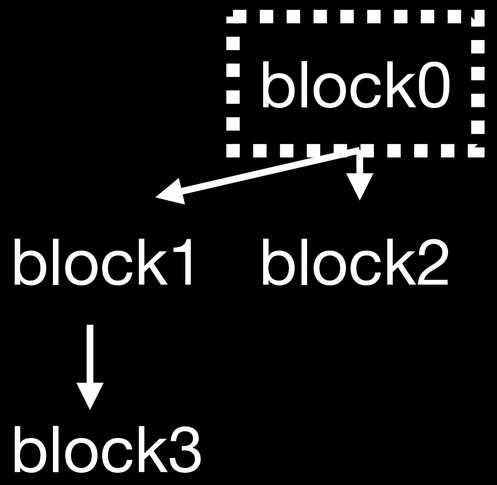




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

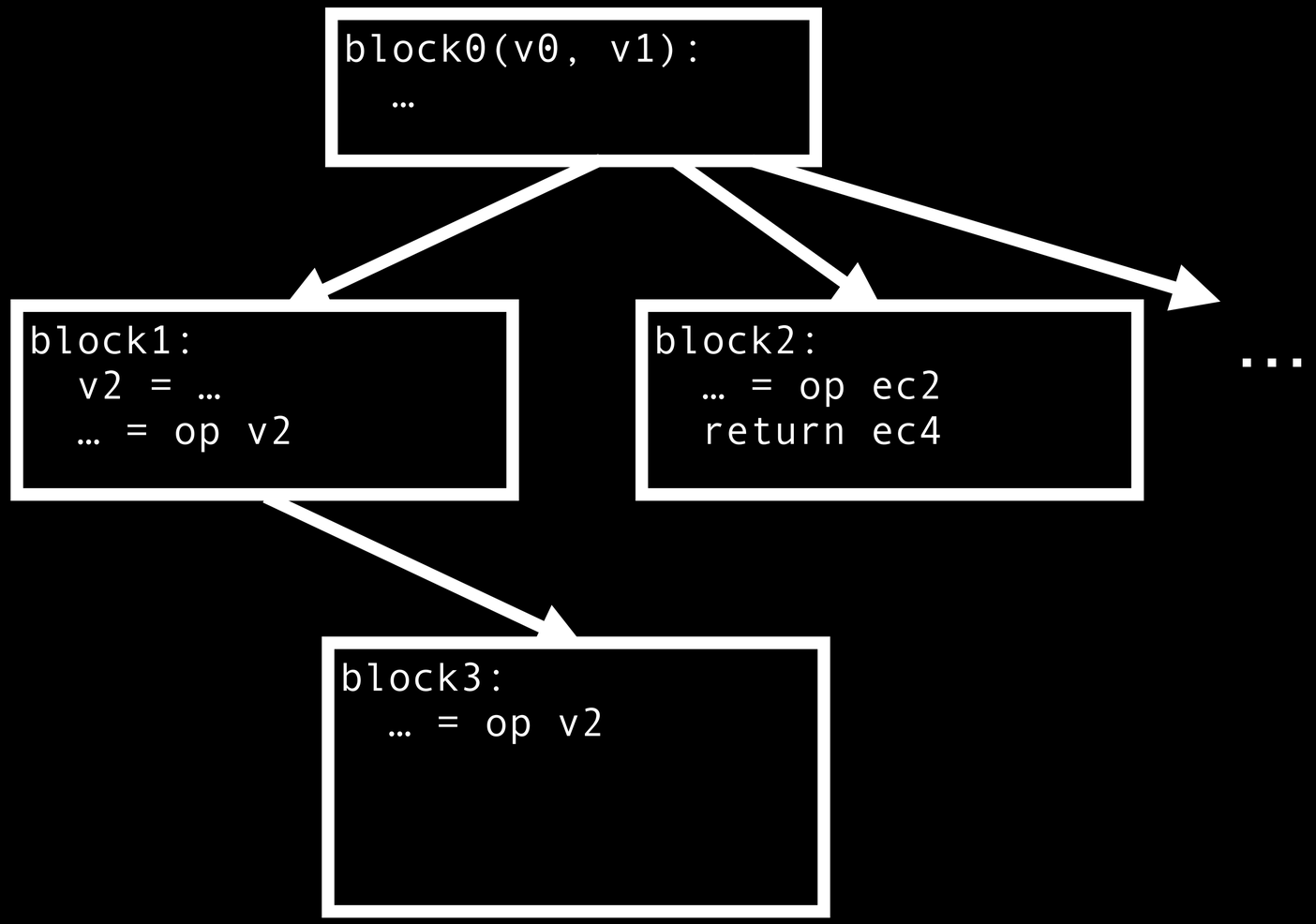
Scoped Elaboration

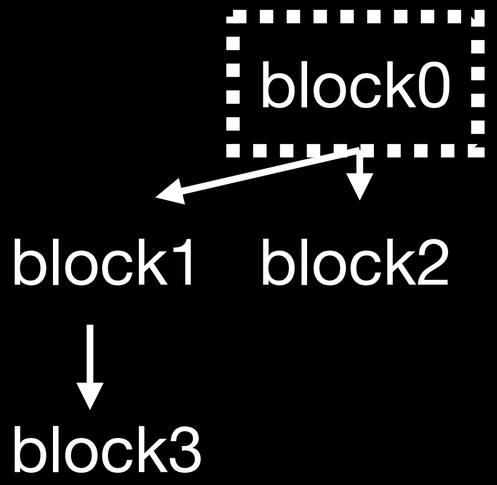




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

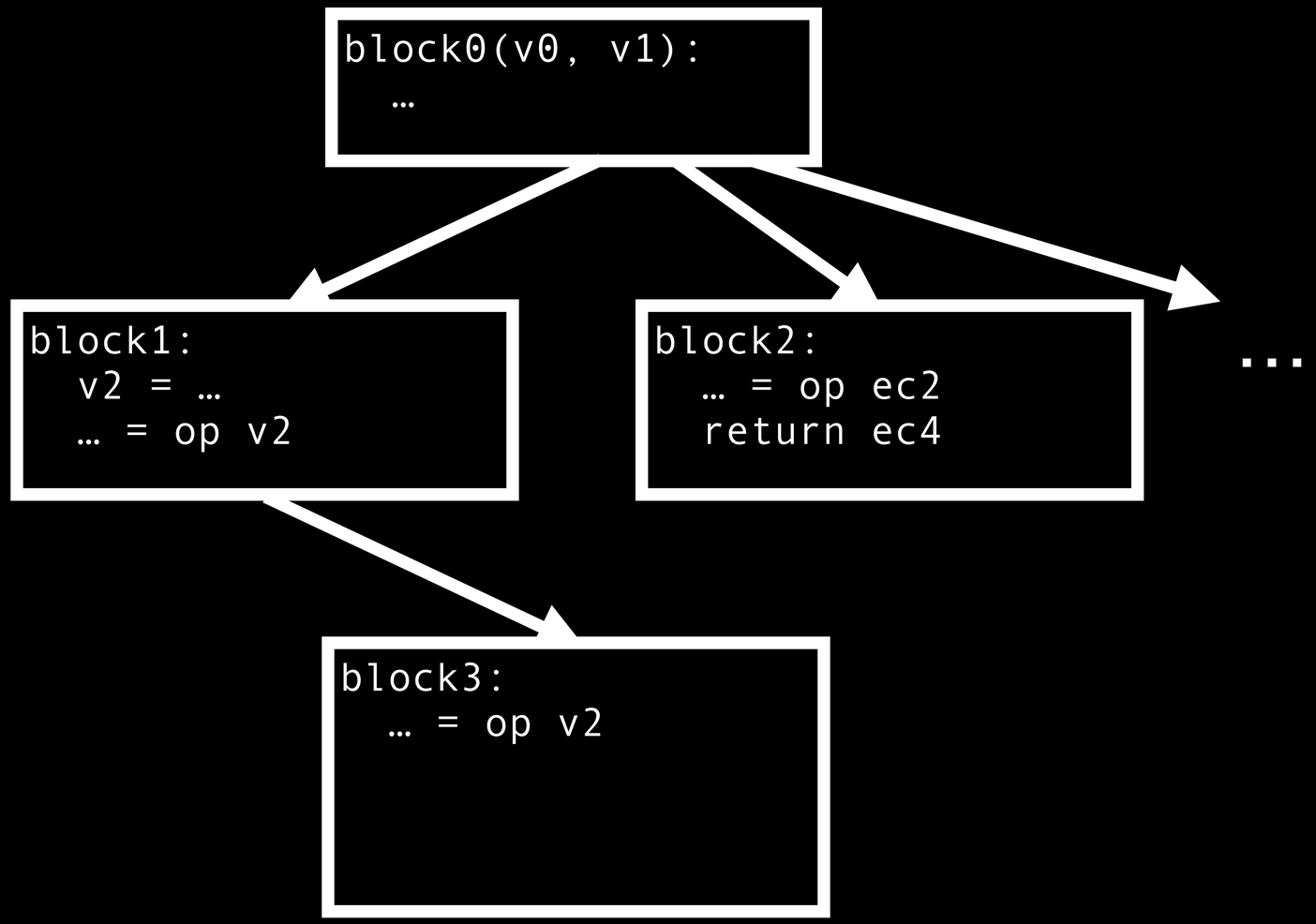
Scoped Elaboration

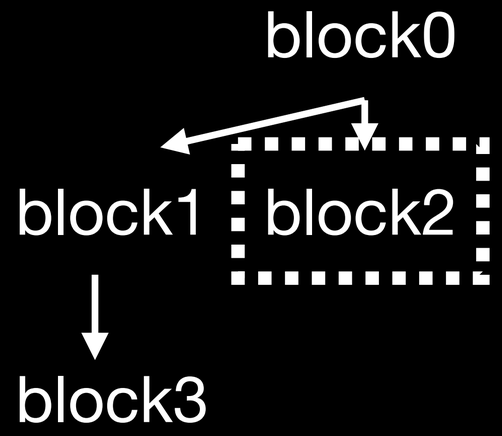




eclass	elaborated
ec0	v0
ec1	v1

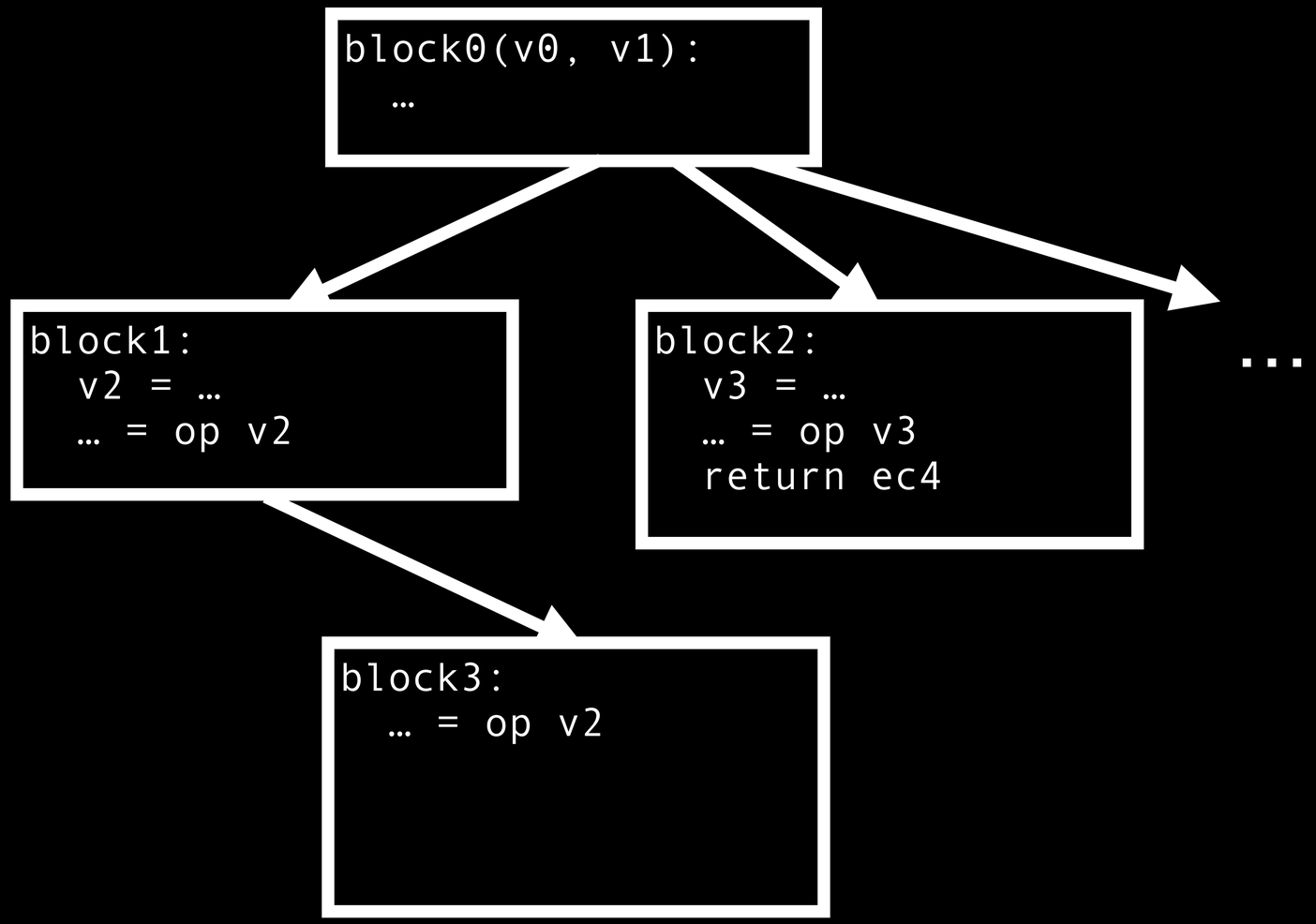
Scoped Elaboration

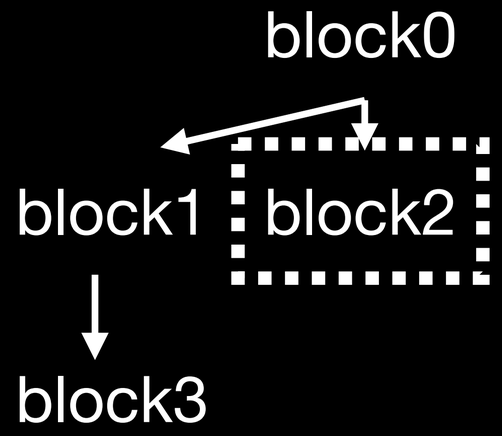




eclass	elaborated
ec0	v0
ec1	v1

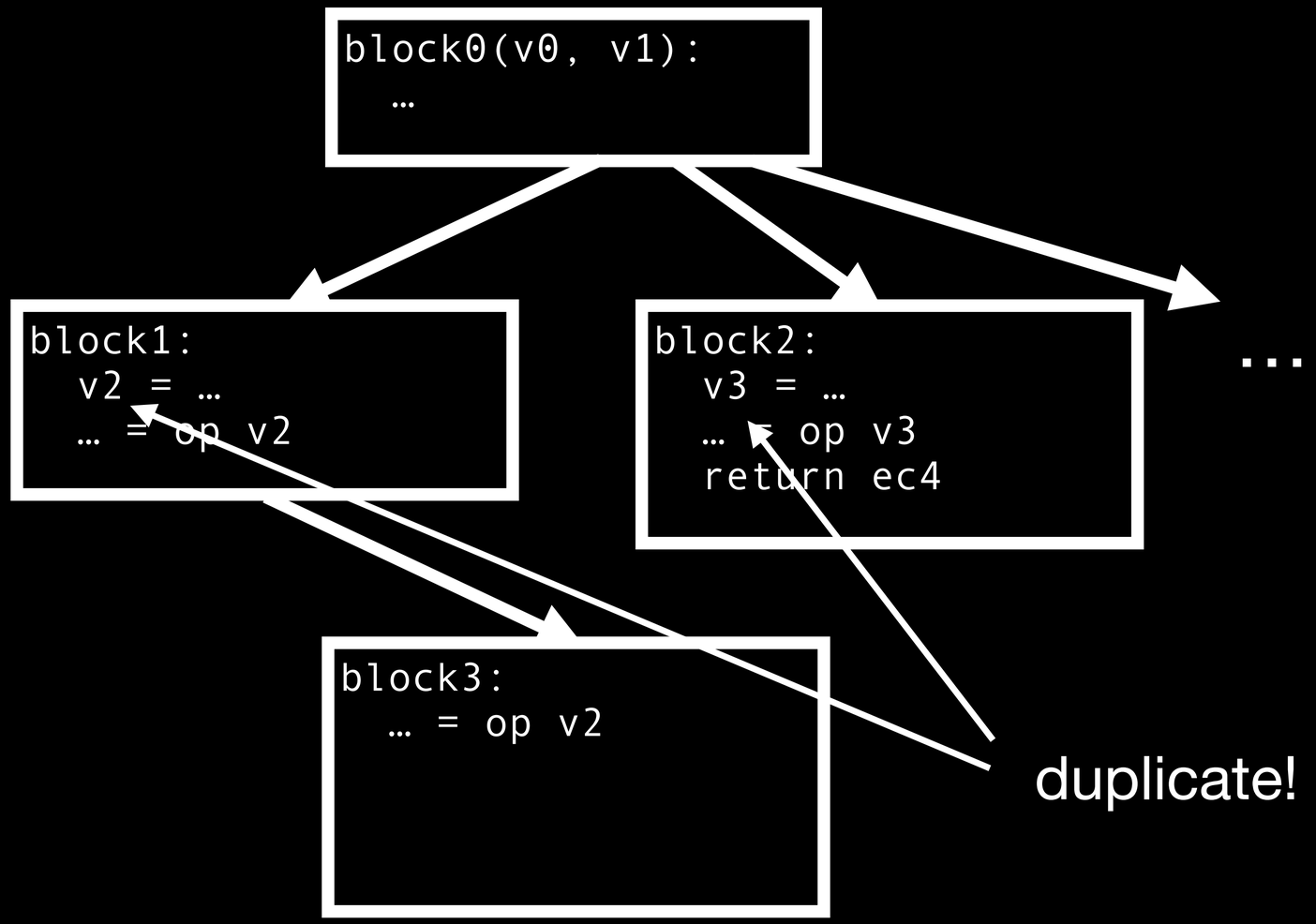
Scoped Elaboration

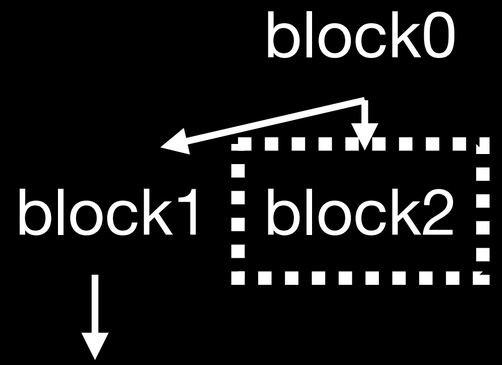




eclass	elaborated
ec0	v0
ec1	v1
ec2	v3

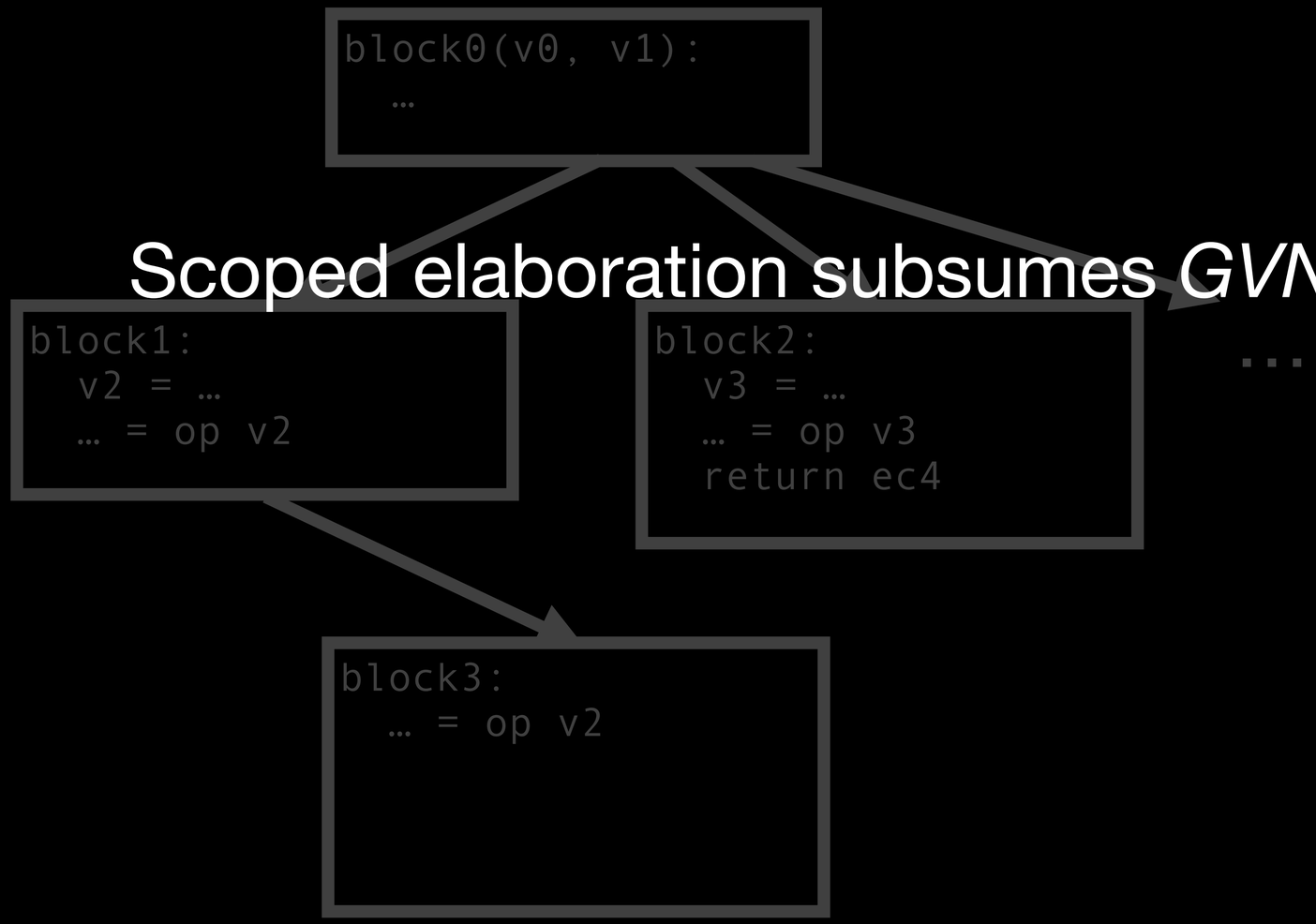
Scoped Elaboration





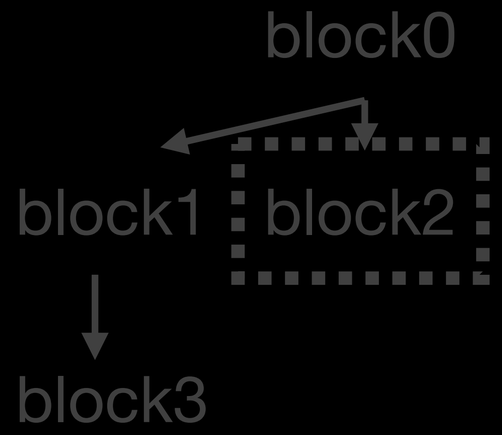
eclass	elaborated
ec0	v0
ec1	v1
ec2	v3

Scoped Elaboration



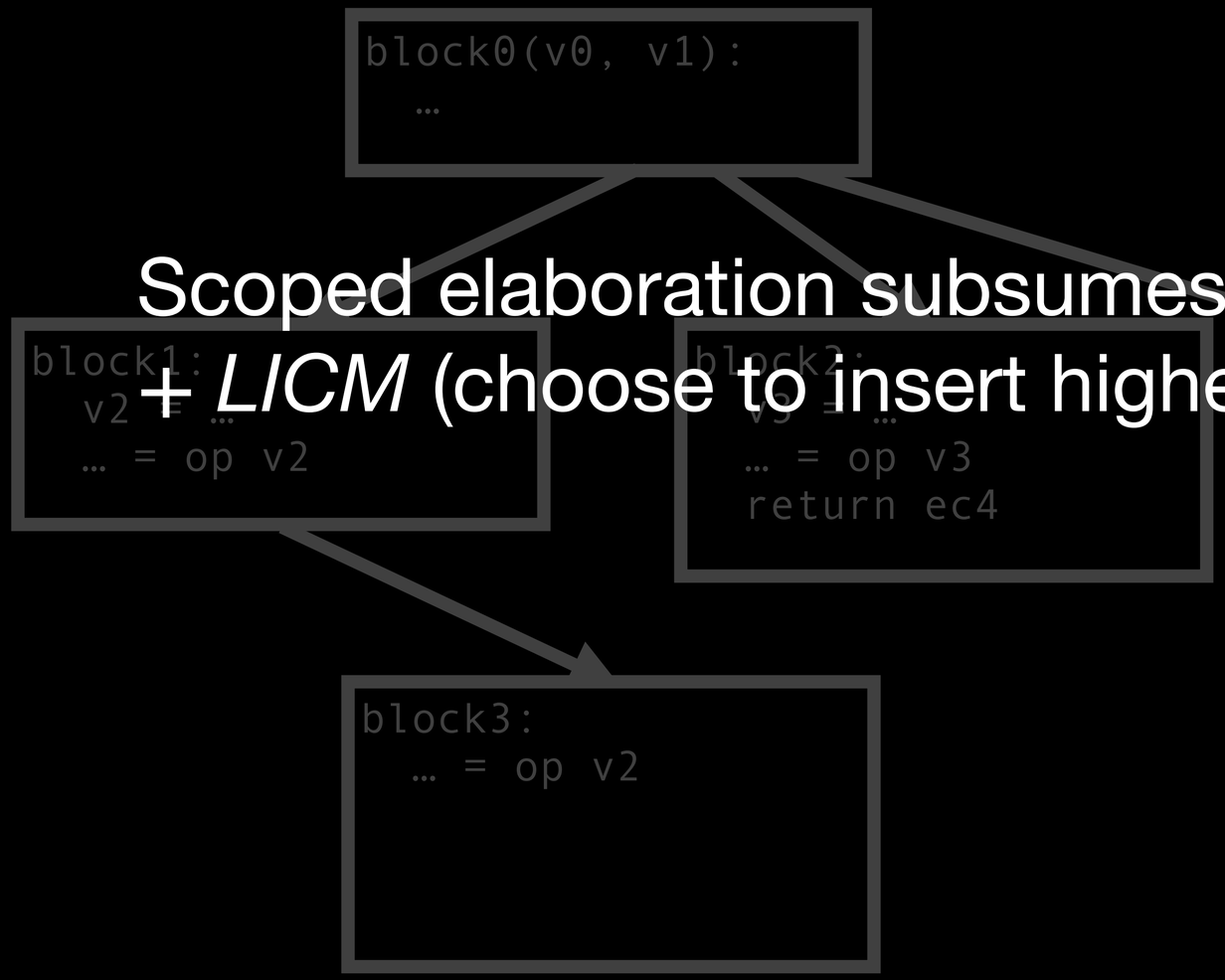
Scoped elaboration subsumes *GVN*

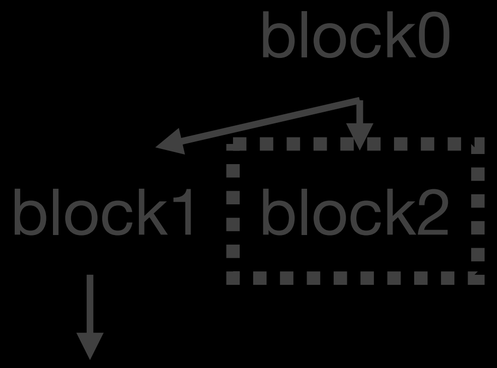
…



eclass	elaborated
ec0	v0
ec1	v1
ec2	v3

Scoped Elaboration

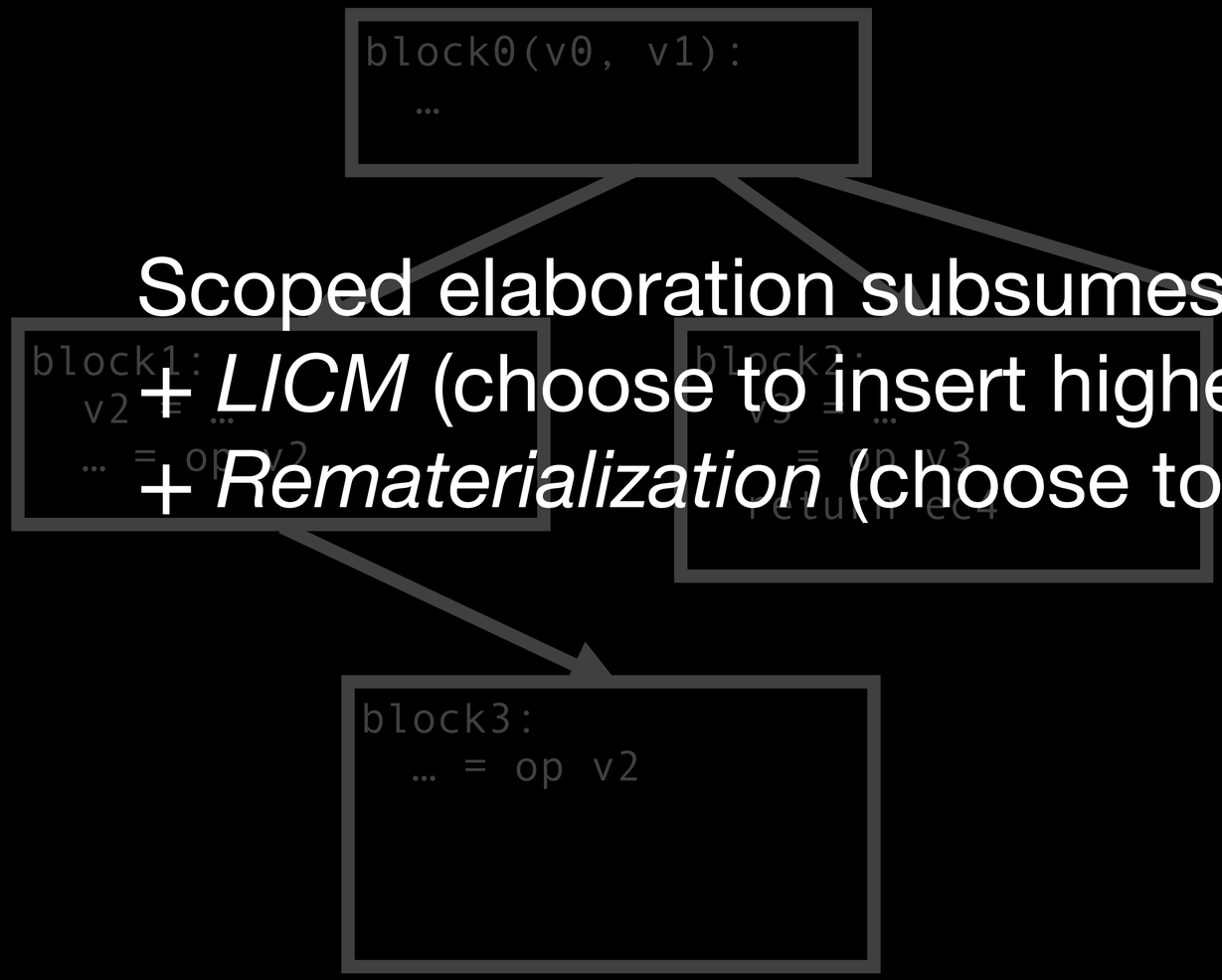


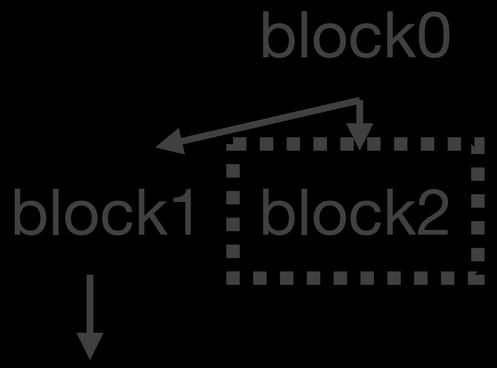


Scoped elaboration subsumes *GVN*
+ *LICM* (choose to insert higher in loopnest + scoped map)

eclass	elaborated
ec0	v0
ec1	v1
ec2	v3

Scoped Elaboration

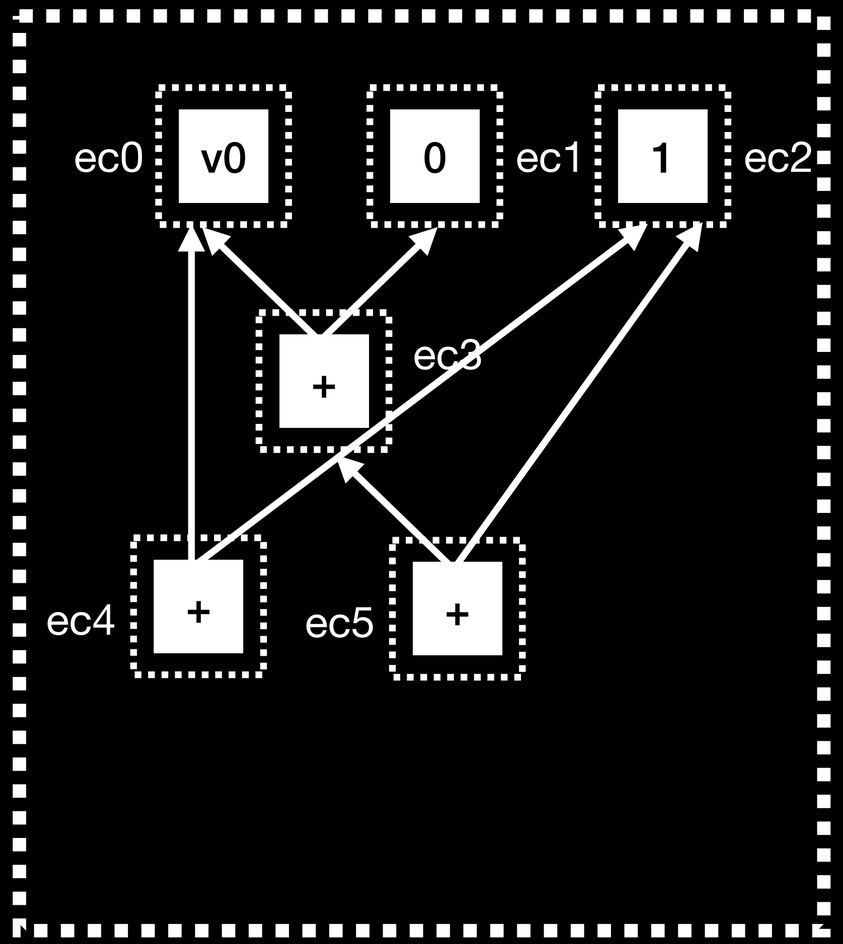




Scoped elaboration subsumes *GVN*
+ *LICM* (choose to insert higher in loopnest + scoped map)
+ *Rematerialization* (choose to create duplicate anyway)

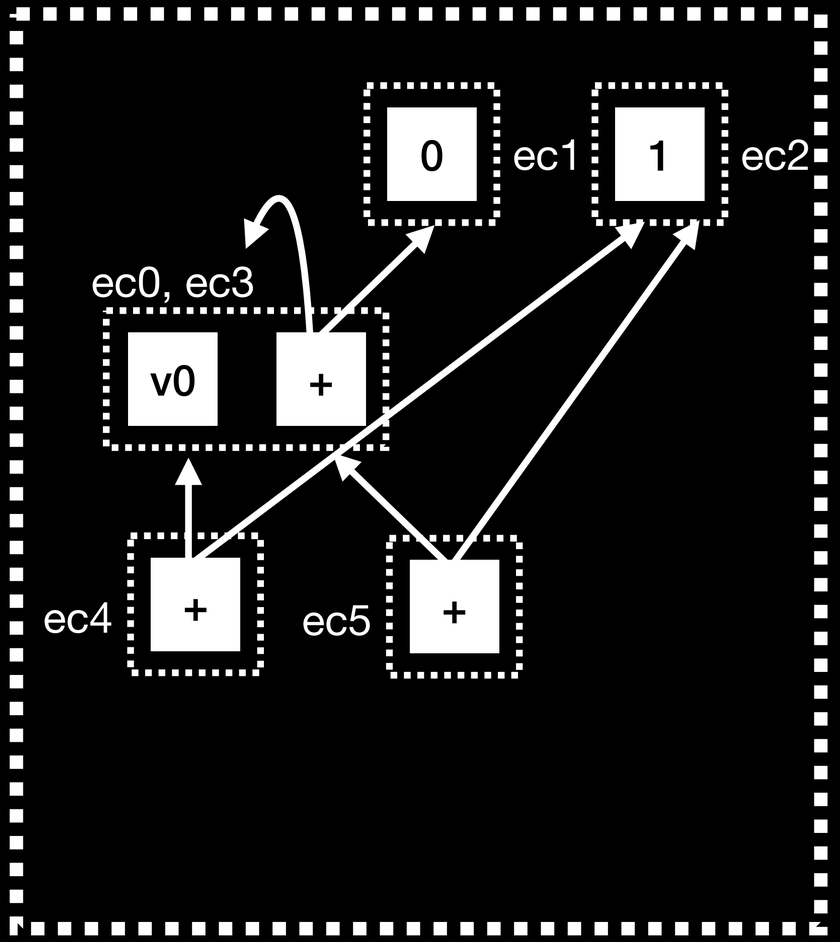
eclass	elaborated
ec0	v0
ec1	v1
ec2	v3

Rewrites and Repair



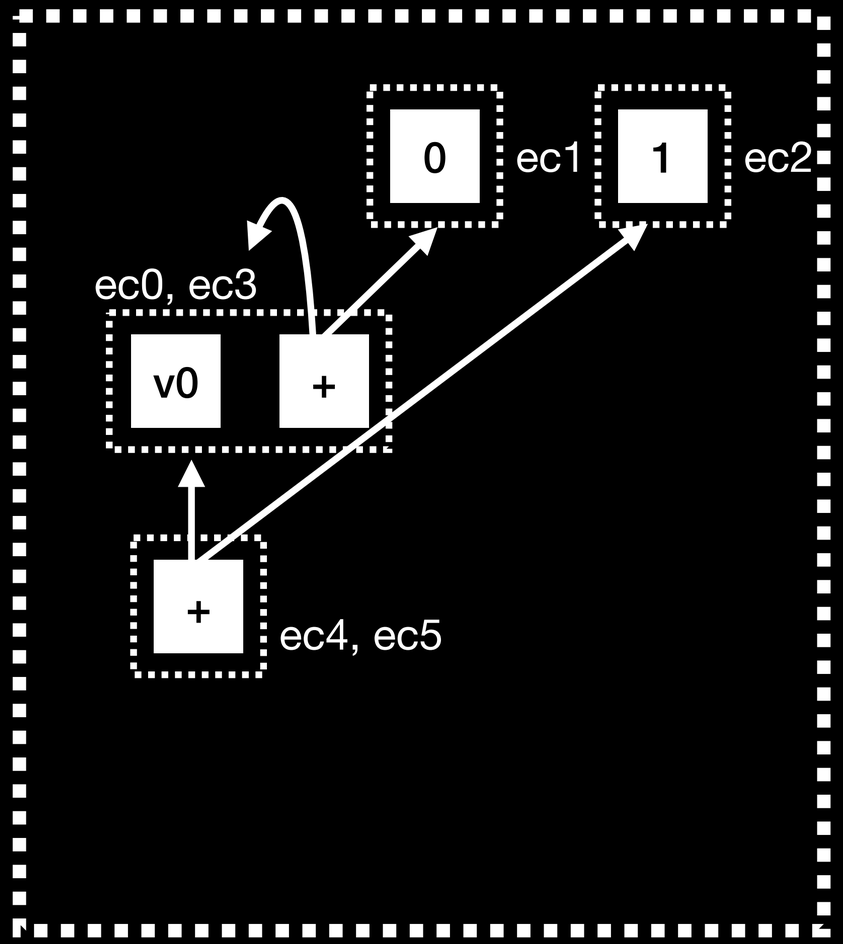
Rewrite: x + 0 => x

Rewrites and Repair



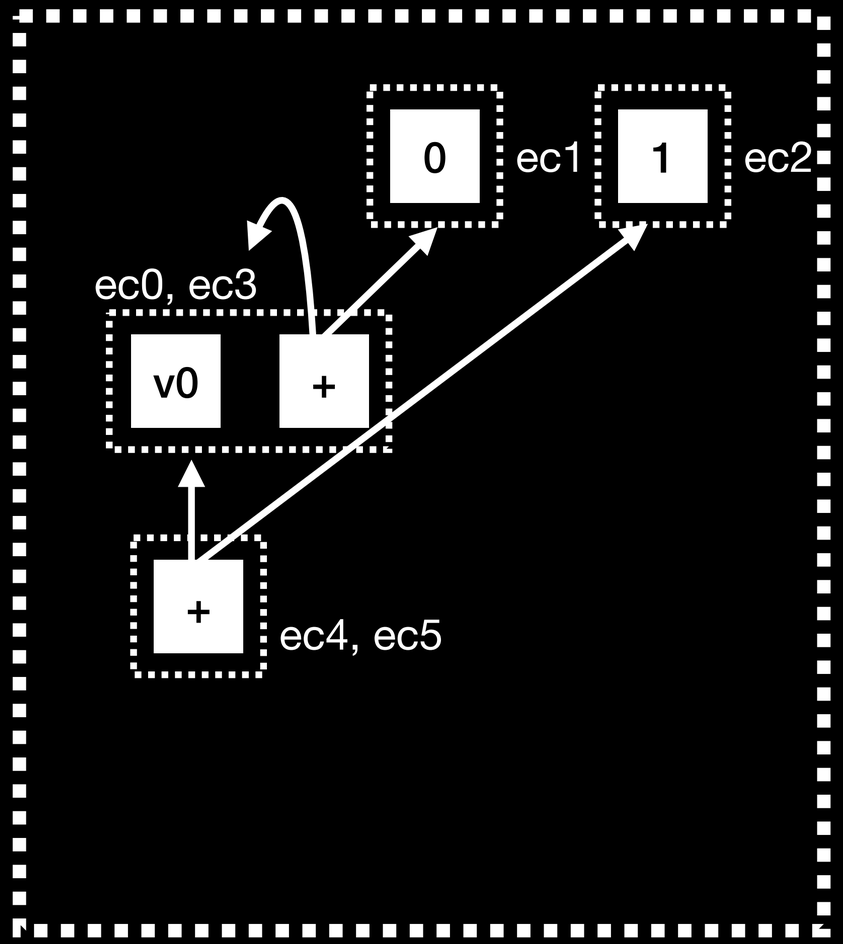
Rewrite: x + 0 => x

Rewrites and Repair



Rewrite: x + 0 => x

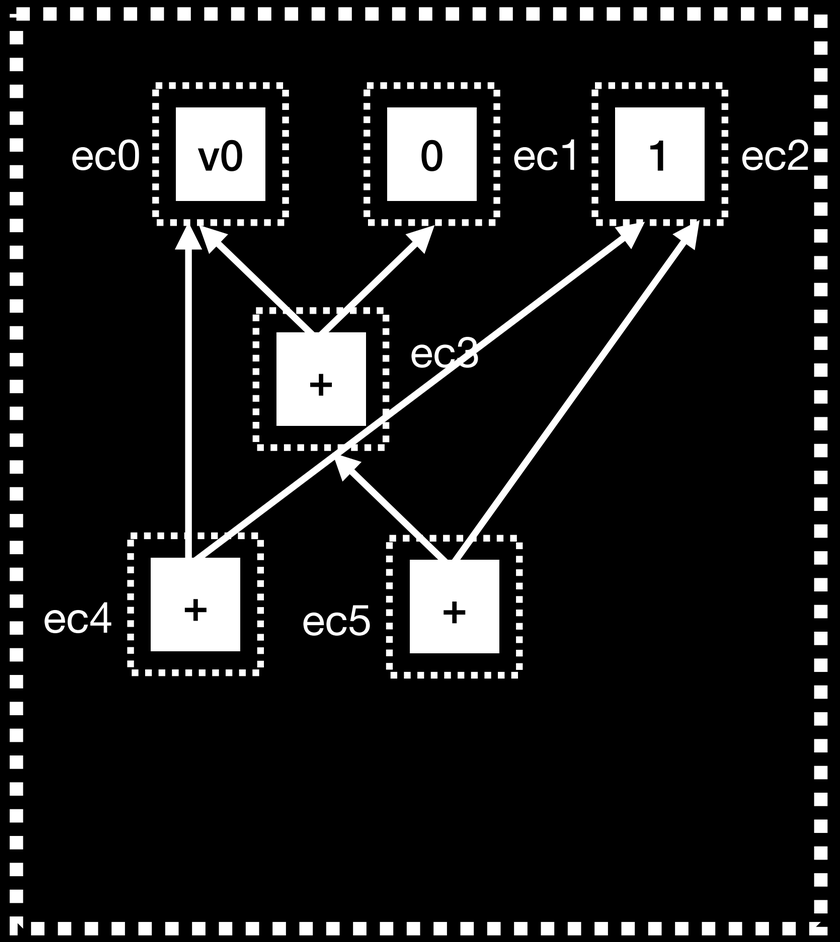
Rewrites and Repair



Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

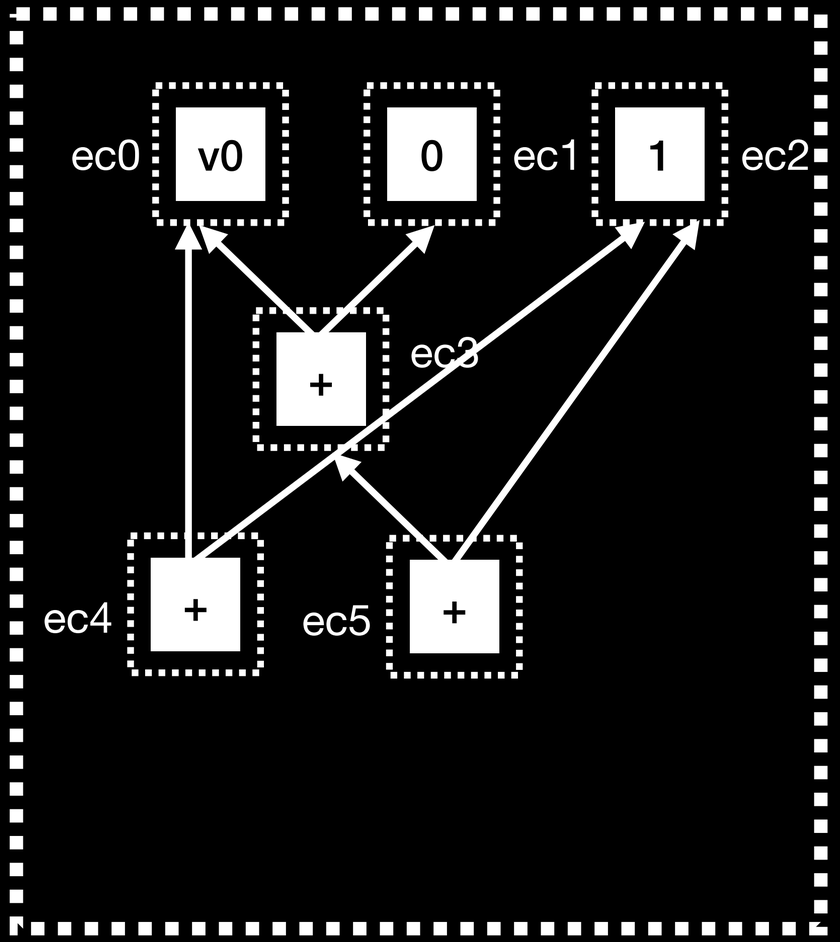
Rewrites and Repair



Rewrite: x + 0 => x

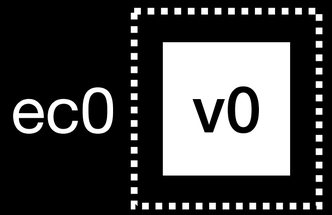
Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

Rewrites and Repair

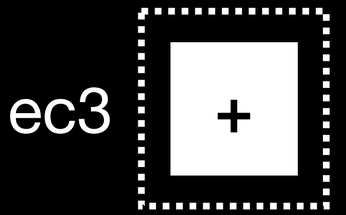


Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

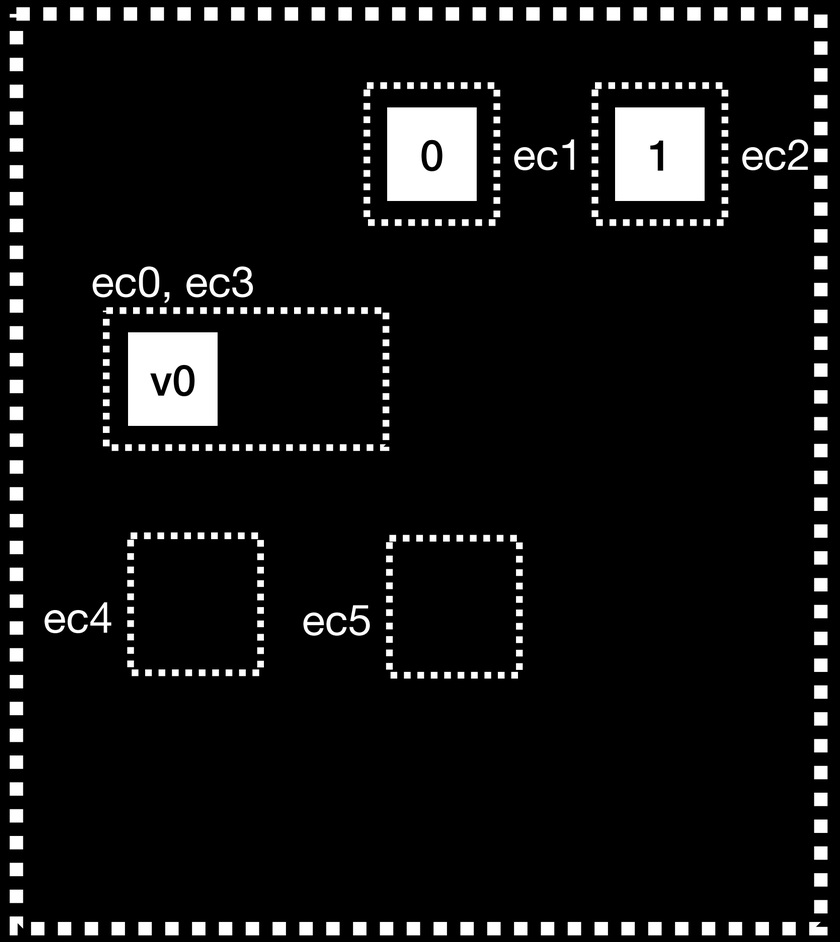


Parents:
{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2)}



Parents:
{ec5 := (+ ec3 ec2)}

Rewrites and Repair

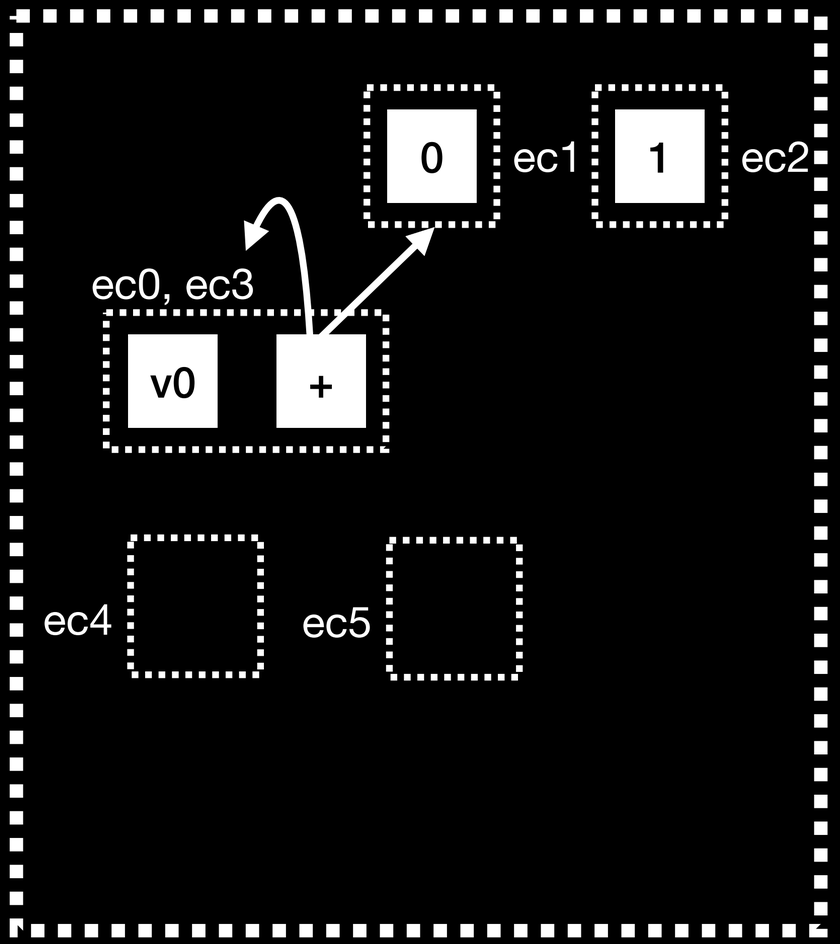


Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

Parents:
{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2),
ec5 := (+ ec3 ec2)}

Rewrites and Repair



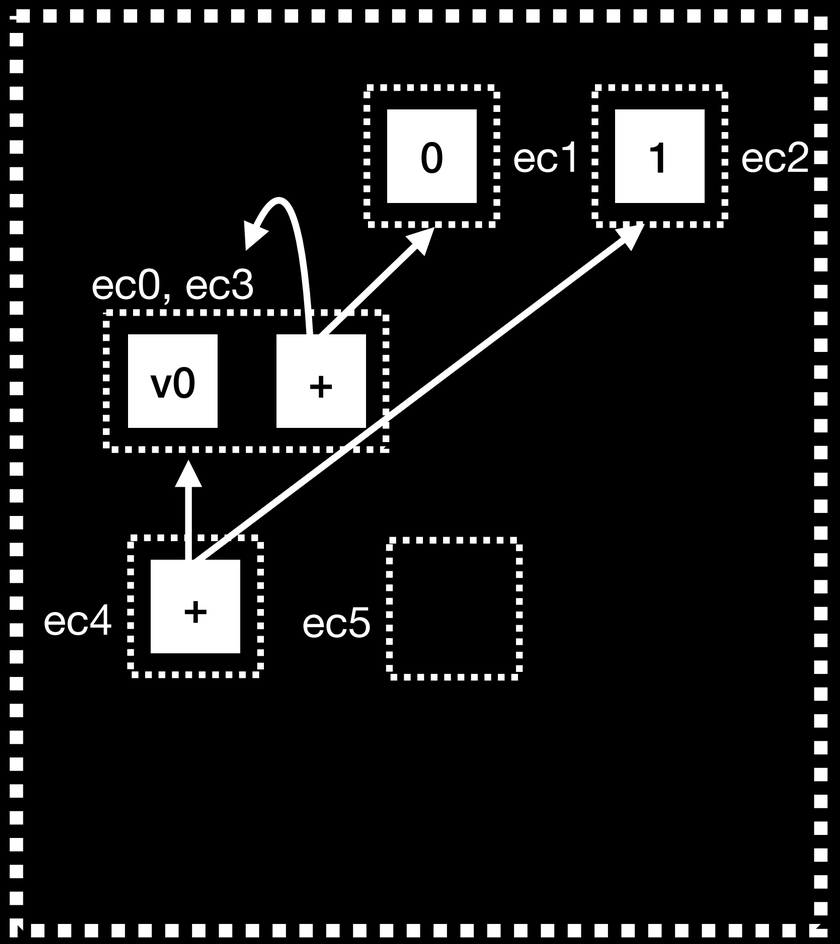
Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

Parents:
{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2),
ec5 := (+ ec3 ec2)}

Re-intern
ec3 := (+ ec0 ec1)

Rewrites and Repair



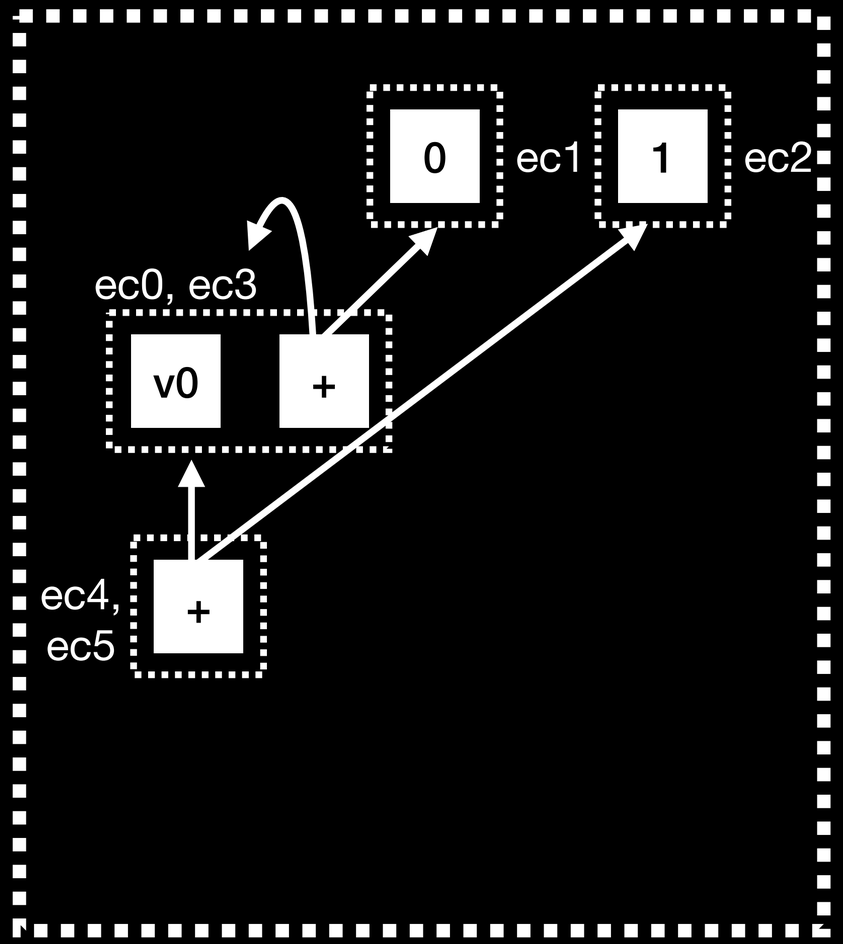
Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

Parents:
{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2),
ec5 := (+ ec3 ec2)}

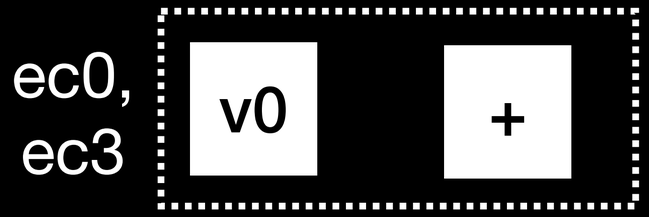
Re-intern
ec4 := (+ ec0 ec2)

Rewrites and Repair



Rewrite: x + 0 => x

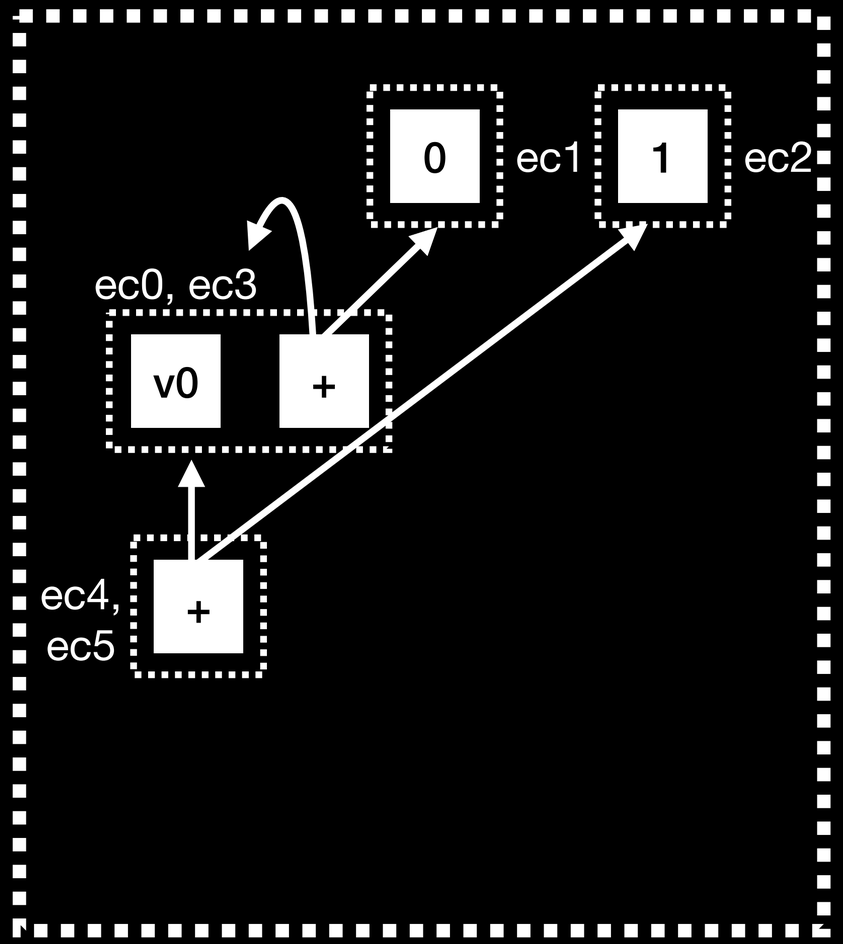
Fixup requires backlinks (parent pointers) and re-interning, which are *costly*



Parents:
{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2),
ec5 := (+ ec3 ec2)}

Re-intern
ec5 := (+ ec3 ec2)
= (+ ec0 ec2)
= ec4

Rewrites and Repair



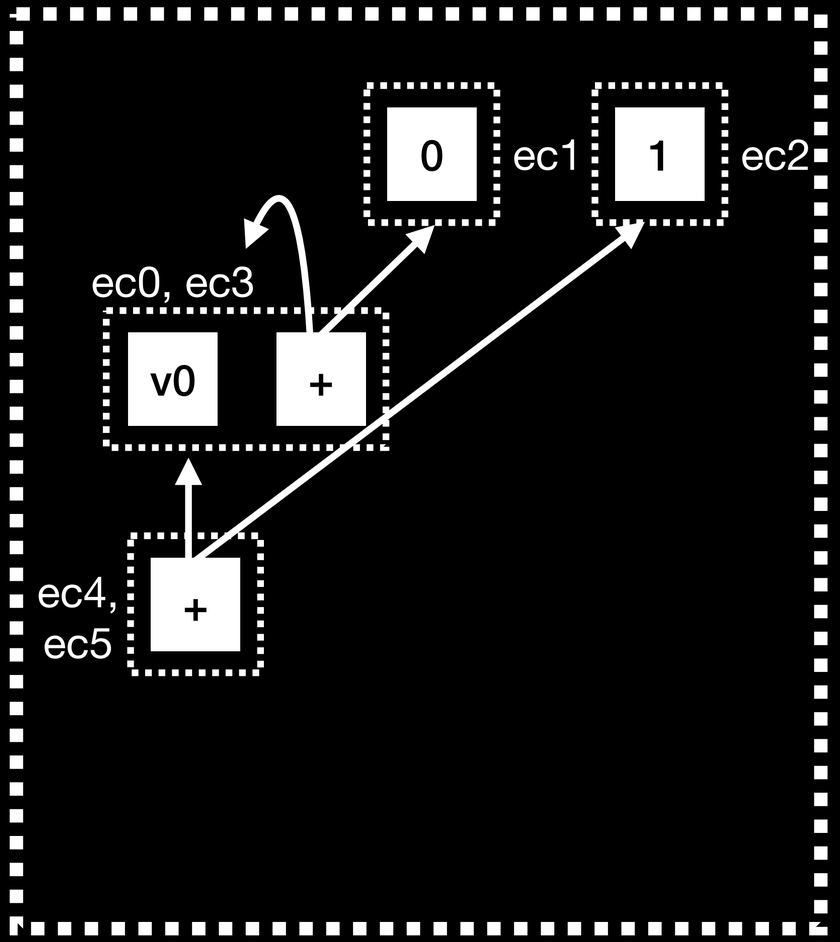
Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

Eliminate:

- Parent lists?
- Duplicated storage of nodes?
- Merging of parent lists, with dedup'ing?

Rewrites and Repair



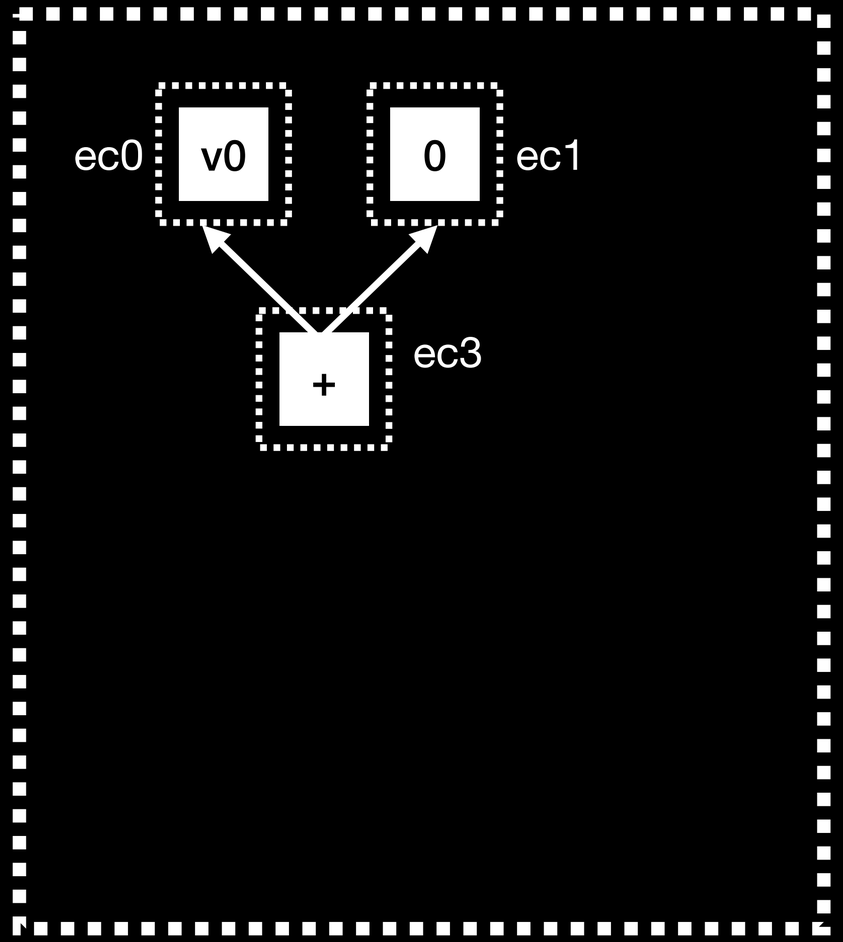
Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

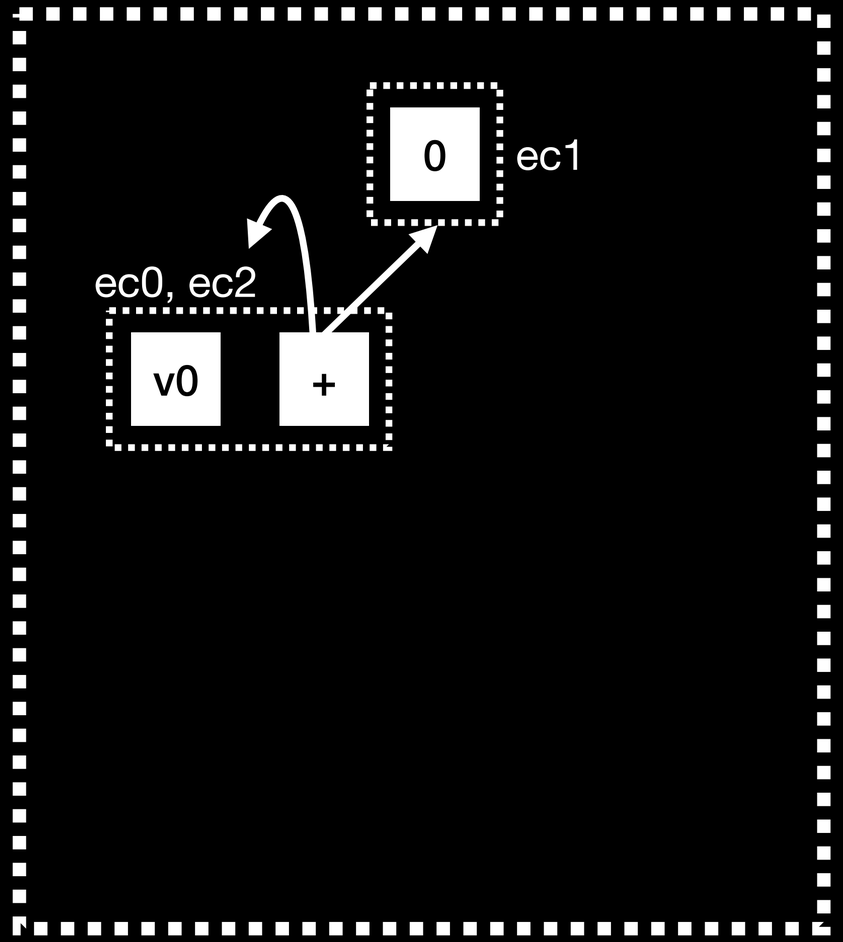
Eliminate:
- Parent lists?
- Duplicated storage of nodes?
- Merging of parent lists, with dedup'ing?

—> compile + memory overhead too high
vs. traditional compiler pipeline

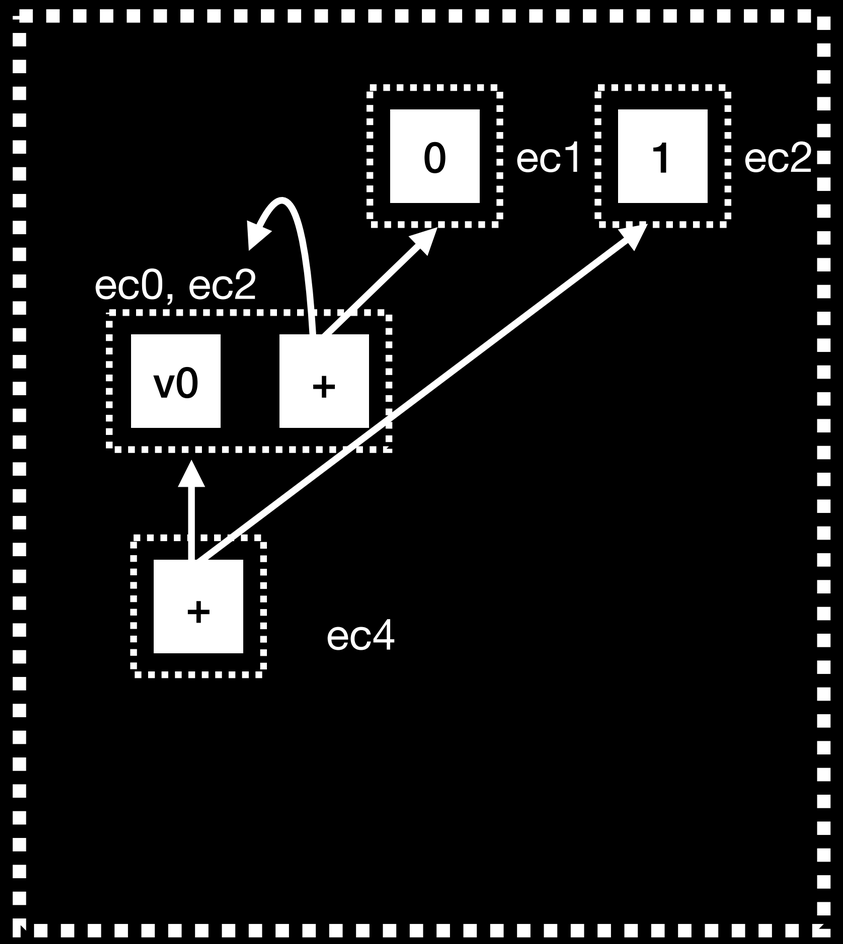
Rewrite eagerly?!



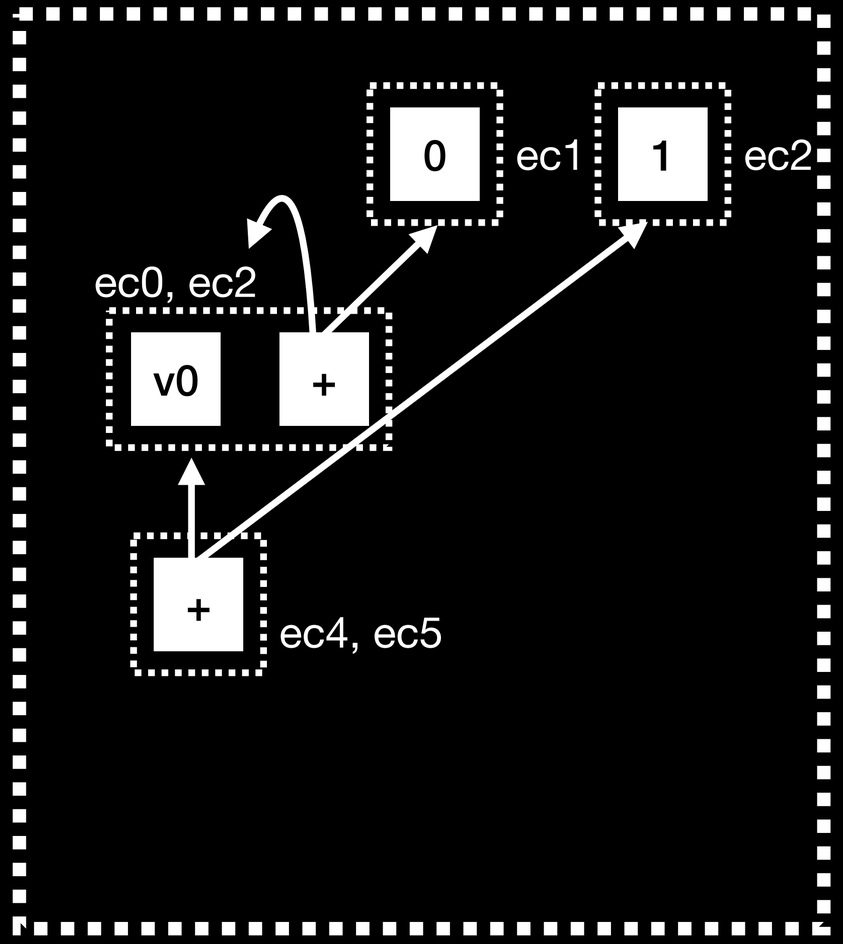
Rewrite eagerly?!



Rewrite eagerly?!



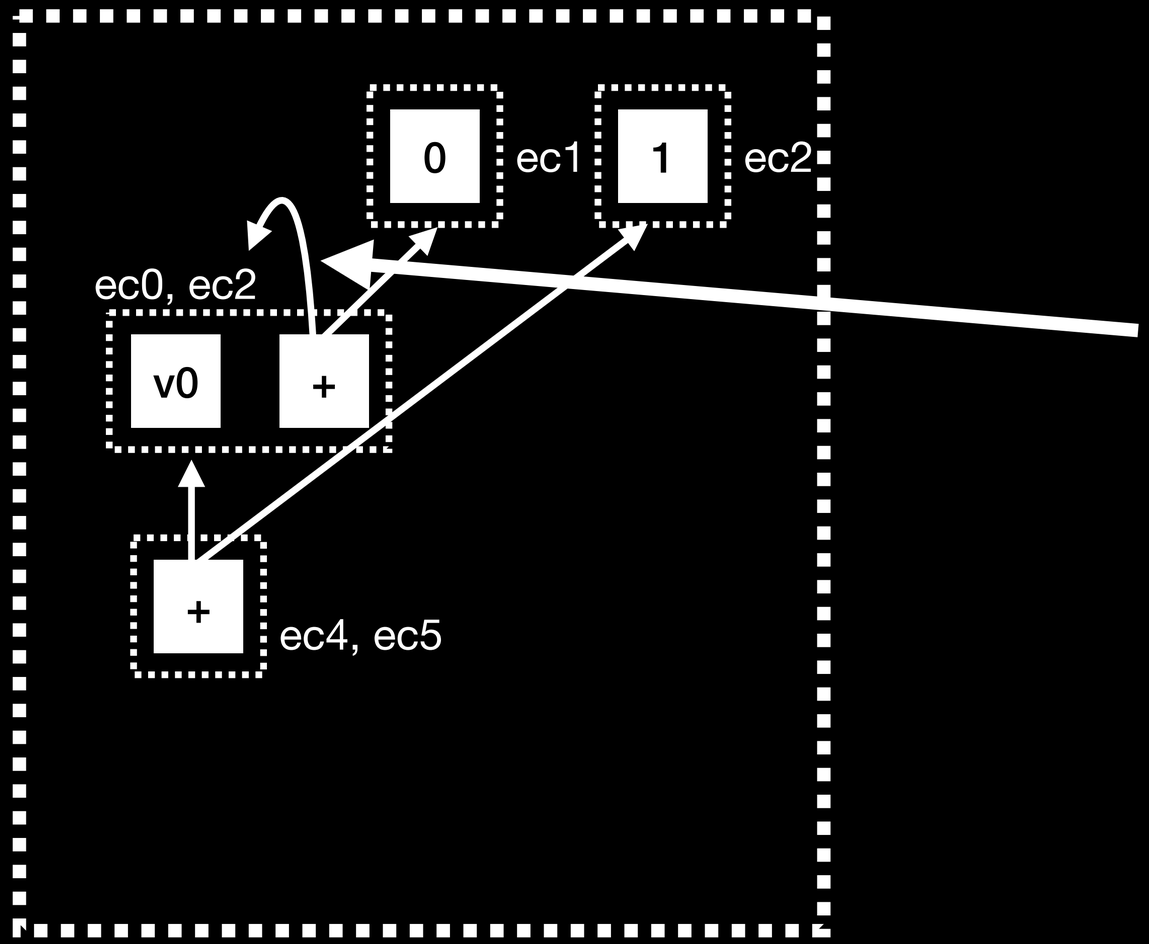
Rewrite eagerly?!



Rewrite already occurred

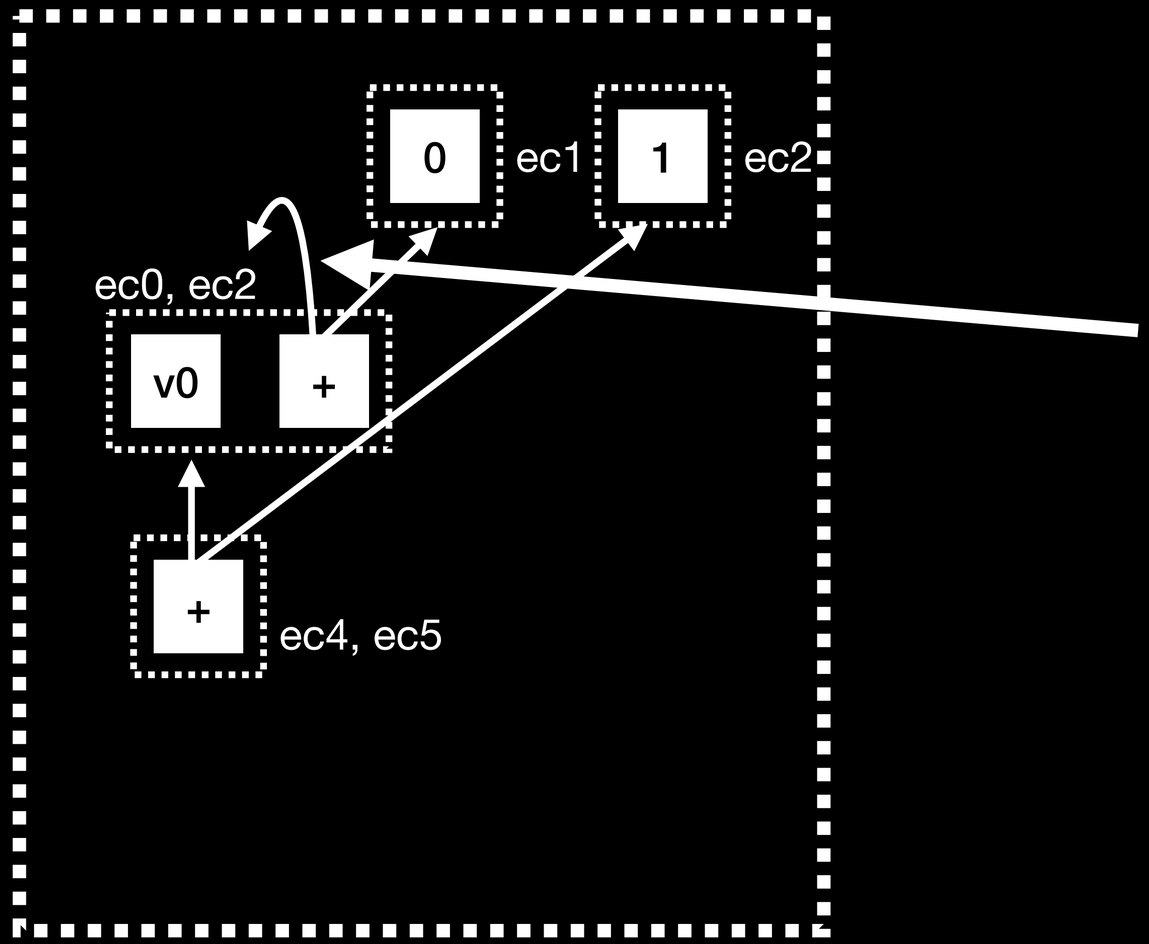
➡ ec5 hash-conses to ec4

Rewrite eagerly?!



How do we handle this cycle?

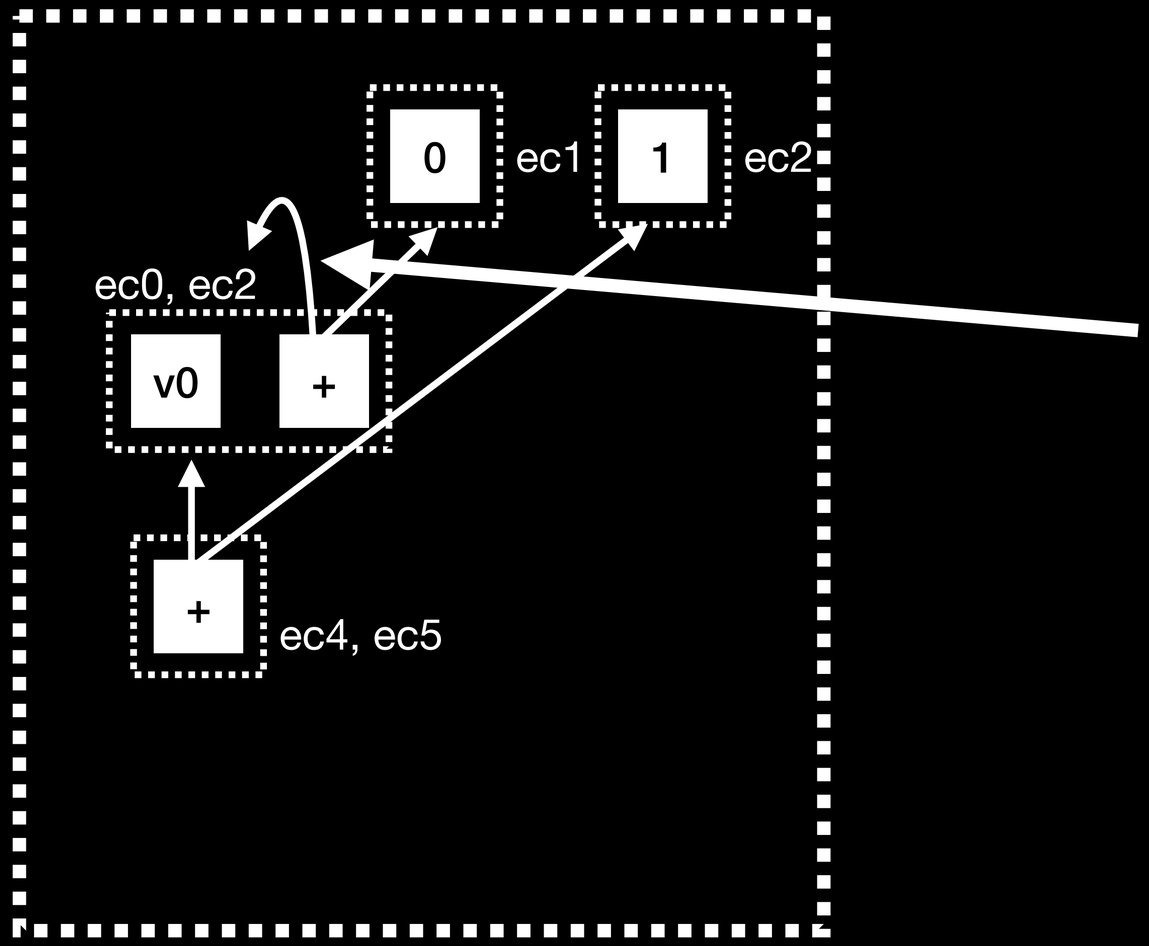
Rewrite eagerly?!



How do we handle this cycle?

- Cycles preclude single pass (imply fixpoint algorithm)

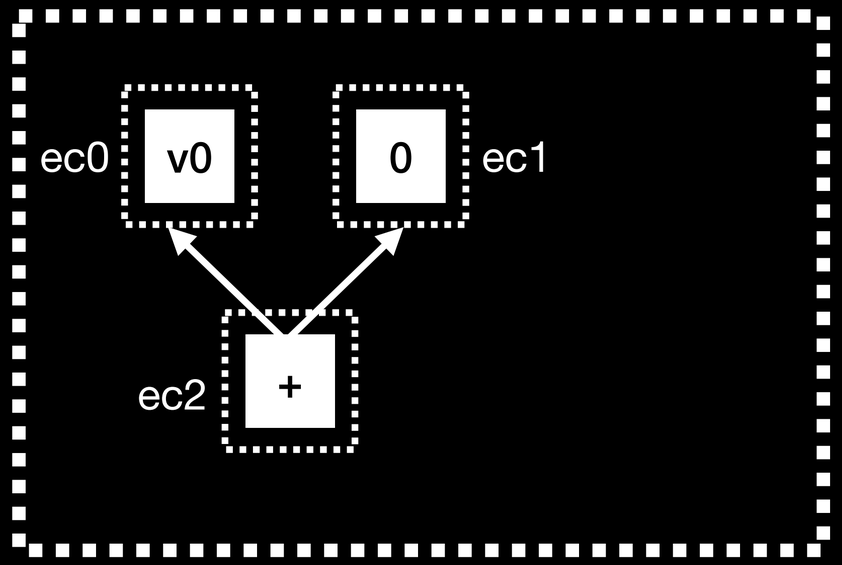
Rewrite eagerly?!



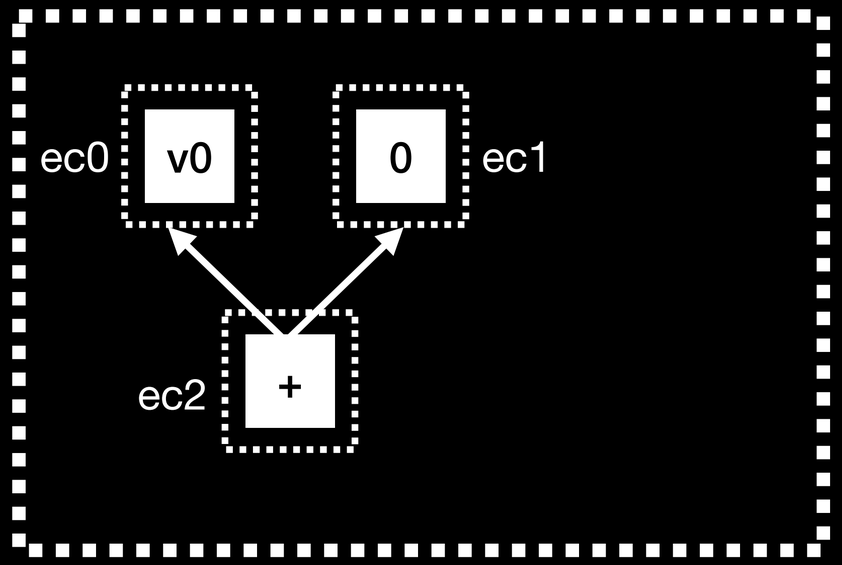
How do we handle this cycle?

- Cycles preclude single pass (imply fixpoint algorithm)
- We're rewriting the arg after its use (no longer eager)
  —> need parent lists again

Cycles in E-graphs

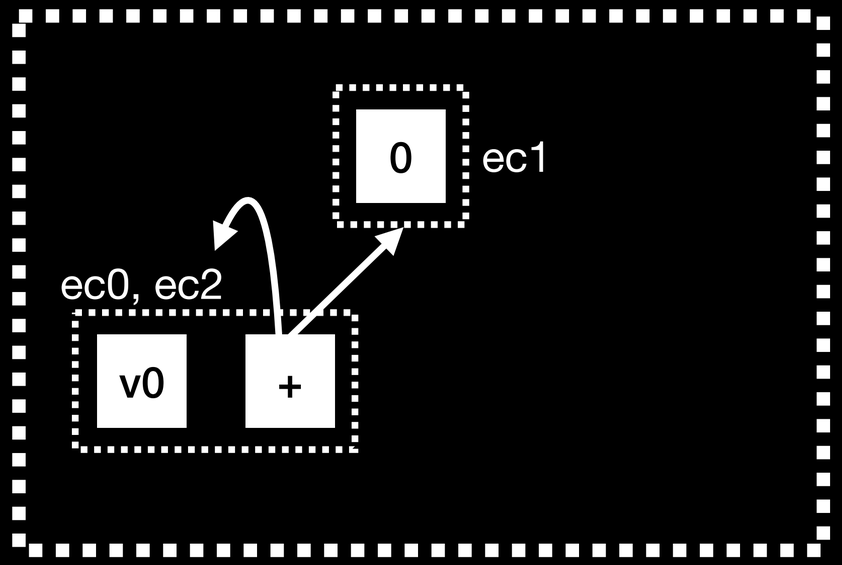


Cycles in E-graphs



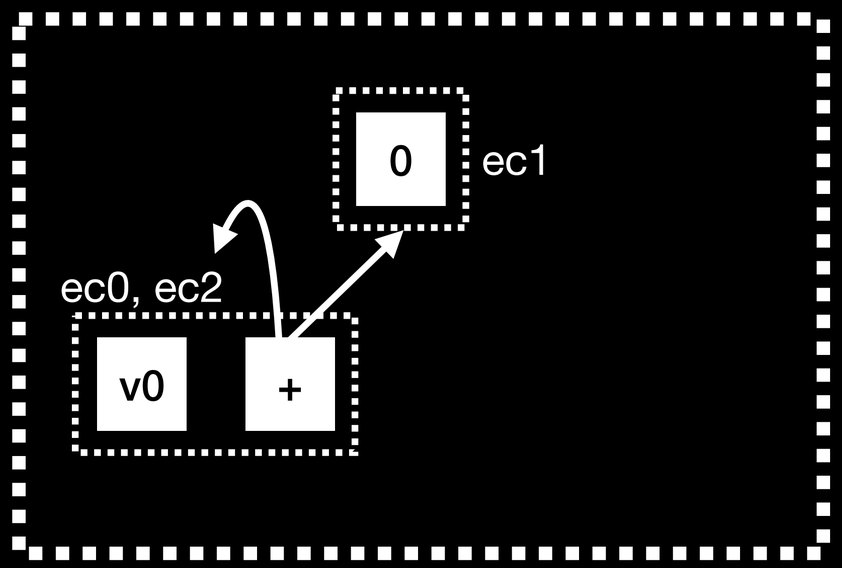
x + 0 => x

Cycles in E-graphs



x + 0 => x

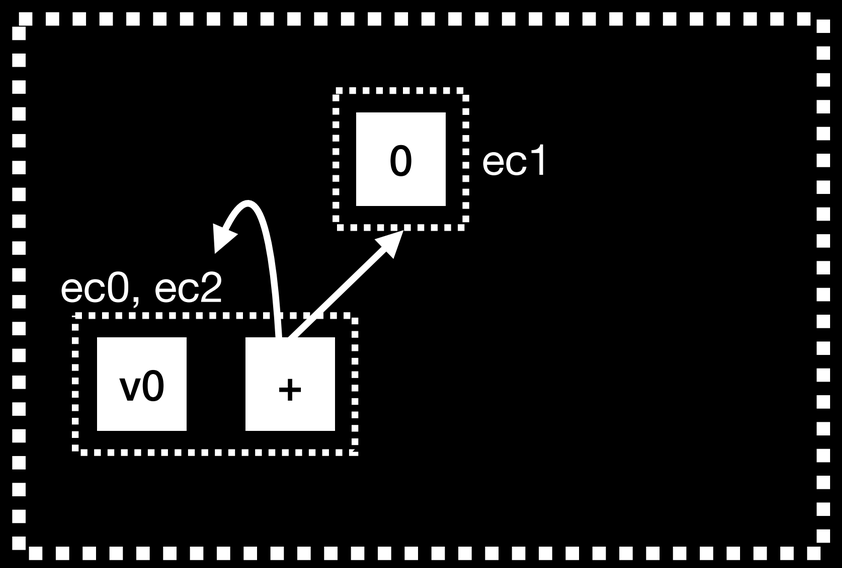
Cycles in E-graphs



x + 0 => x

Observation:
- egraph does not record rewrite “direction”

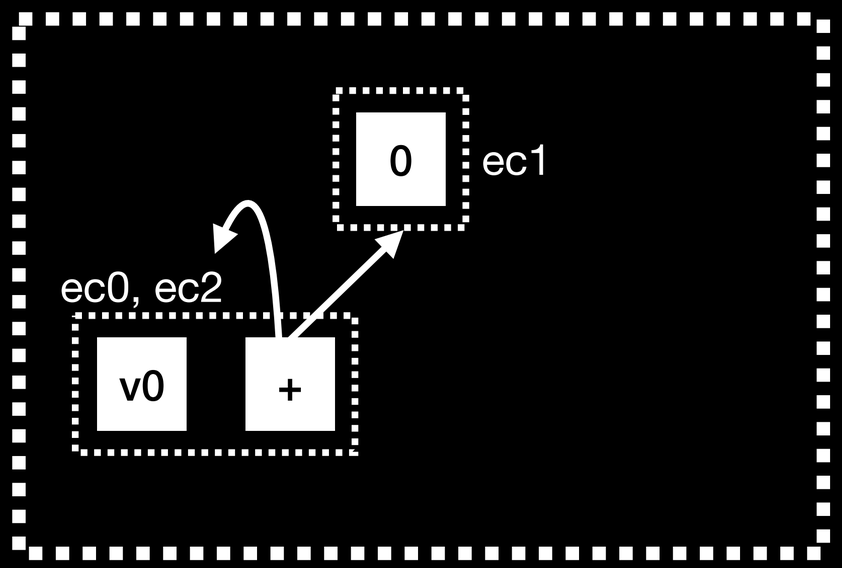
Cycles in E-graphs



x + 0 => x

Observation:
- egraph does not record rewrite “direction”
- this egraph equivalent to
  - start with x
  - rewrite with x => x + 0

Cycles in E-graphs

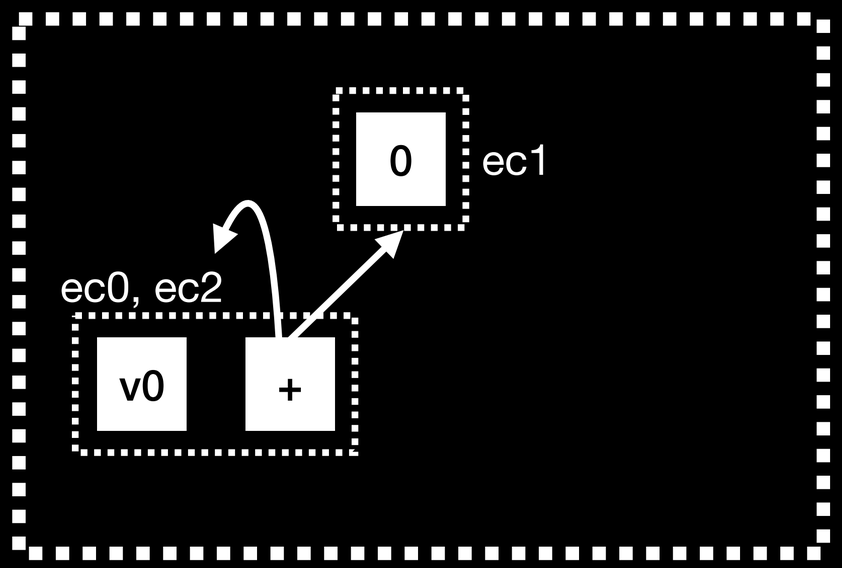


x + 0 => x

Observation:

- egraph does not record rewrite “direction”
- this egraph equivalent to
  - start with x
  - rewrite with x => x + 0
- rewrite rules that equate *part* to *whole* are (reverse)-generative

Cycles in E-graphs



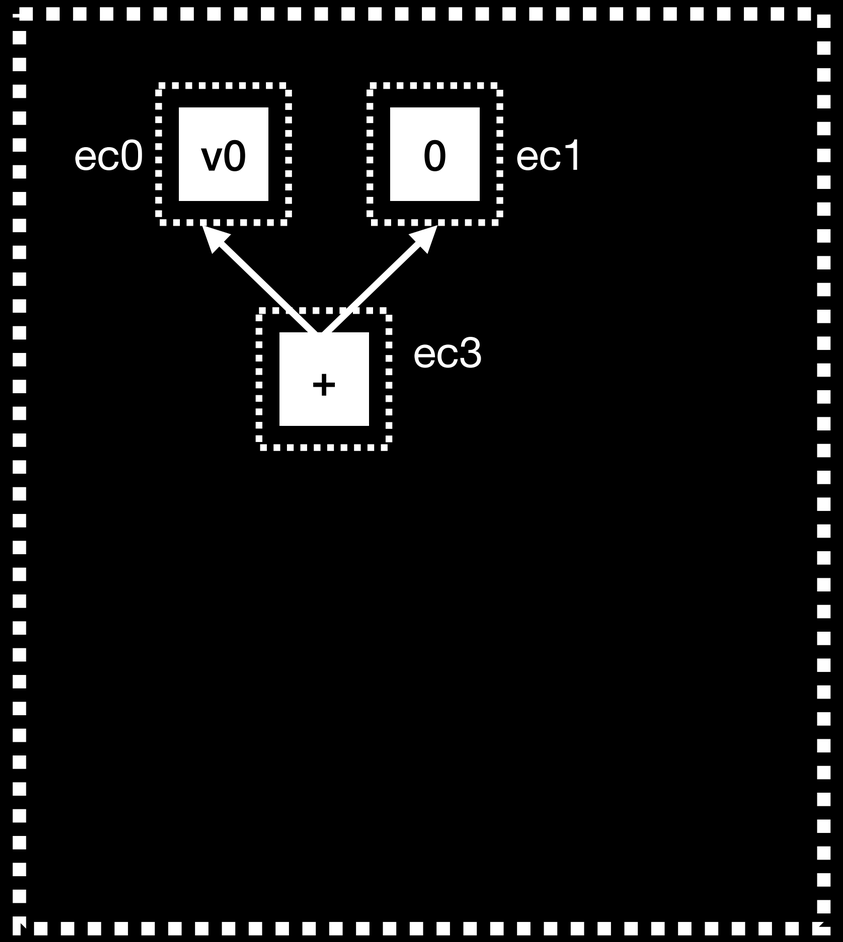
x + 0 => x

Observation:

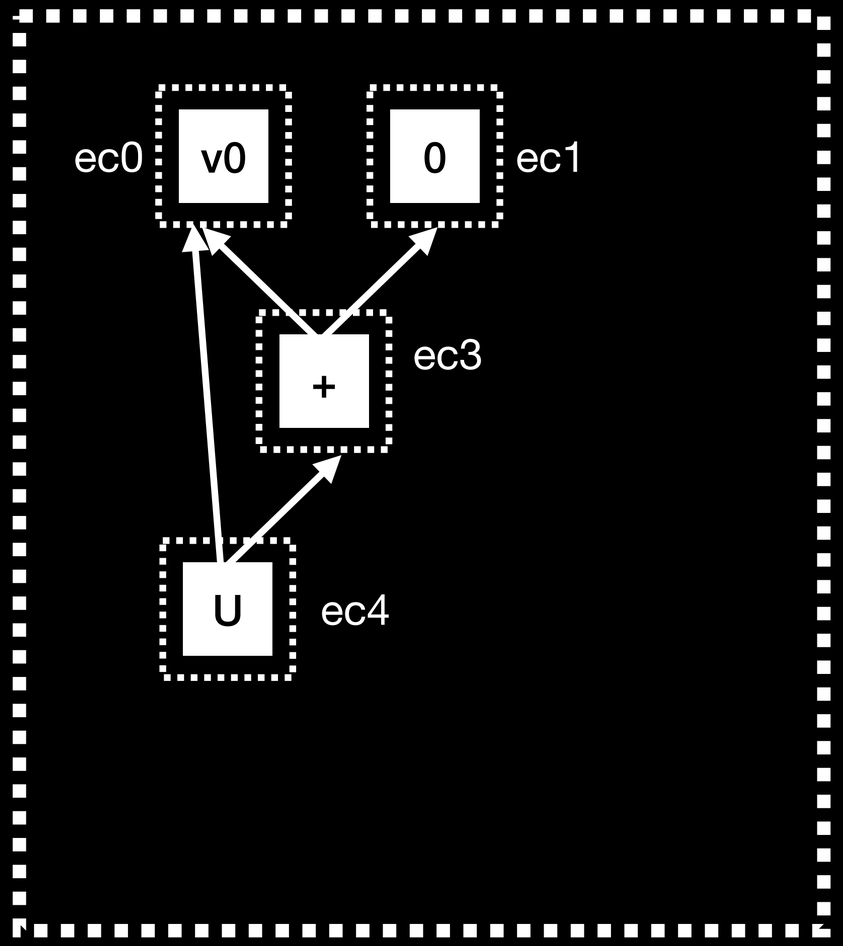
- egraph does not record rewrite “direction”
- this egraph equivalent to
  - start with x
  - rewrite with x => x + 0
- rewrite rules that equate *part* to *whole* are (reverse)-generative

Cycles occur even if original egraph is acyclic (e.g., from SSA)

Persistent immutable e-classes

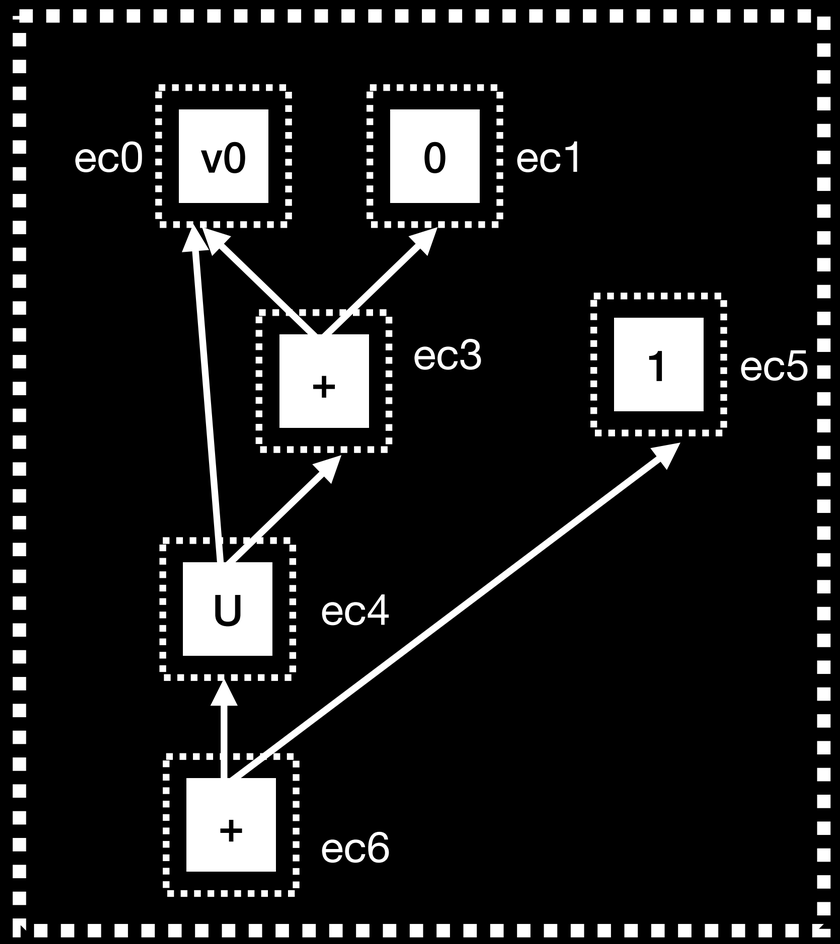


Persistent immutable e-classes



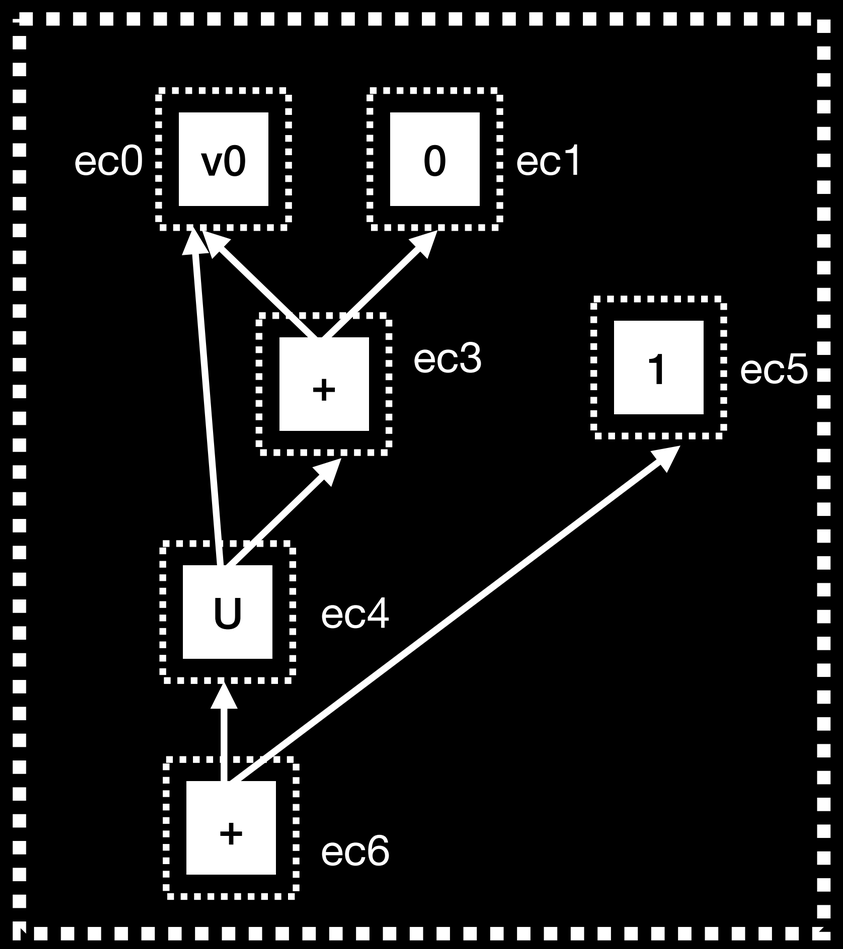
- Never rewrite a node
- Represent eclasses as *trees* of *union nodes*

Persistent immutable e-classes



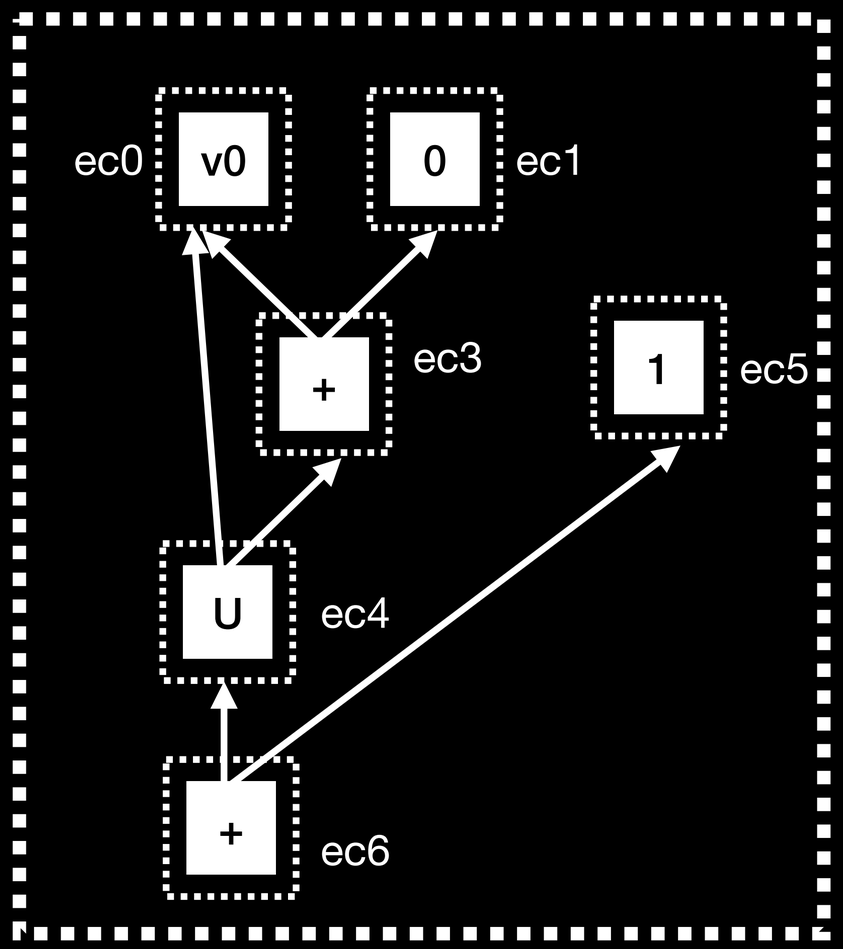
- Never rewrite a node
- Represent eclasses as *trees* of *union nodes*

Persistent immutable e-classes



- Never rewrite a node
- Represent eclasses as *trees* of *union nodes*
- As we build the egraph, track *latest* id for a given value
  - there is no union-find
- Invoke rewrite rules *when a node is created*

Persistent immutable e-classes



- Never rewrite a node
- Represent eclasses as *trees* of *union nodes*
- As we build the egraph, track *latest* id for a given value
  - there is no union-find
- Invoke rewrite rules *when a node is created*

- Key insight: just *don't do congruence closure*
- This might preclude some rule interactions
- But we observe "most" of the interesting interactions are direct chaining

egraphs without congruence closure

(or a union-find!)

We needed compile speed …

egraphs without congruence closure

(or a union-find!)

We needed compile speed …

… so we eliminated the rebuild fixpoint loop and parent pointers

egraphs without congruence closure

(or a union-find!)

We needed compile speed …

… so we eliminated the rebuild fixpoint loop and parent pointers

… by building only once, eagerly, doing all rewrites as soon as possible

egraphs without congruence closure

(or a union-find!)

We needed compile speed …

… so we eliminated the rebuild fixpoint loop and parent pointers

… by building only once, eagerly, doing all rewrites as soon as possible

… then every user references the last (root) union node

egraphs without congruence closure

(or a union-find!)

We needed compile speed …

… so we eliminated the rebuild fixpoint loop and parent pointers

… by building only once, eagerly, doing all rewrites as soon as possible

… then every user references the last (root) union node

… and hopefully no *new* enodes are union'd in later

egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

"hope is not a strategy"

egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

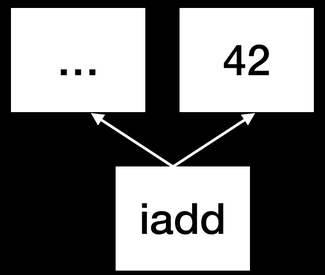
```
v1 = …
v2 = iconst.i32 42
v3 = iadd v1, v2
```

egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

```
v1 = …
v2 = iconst.i32 42
v3 = iadd v1, v2
```

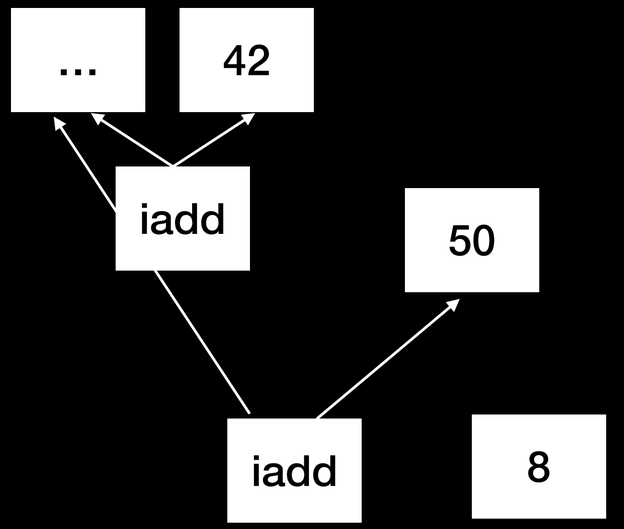


egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

```
v1 = …
v2 = iconst.i32 42
v3 = iadd v1, v2
…
v100 = iconst.i32 50
v101 = iconst.i32 8
v102 = iadd v1, v100
```

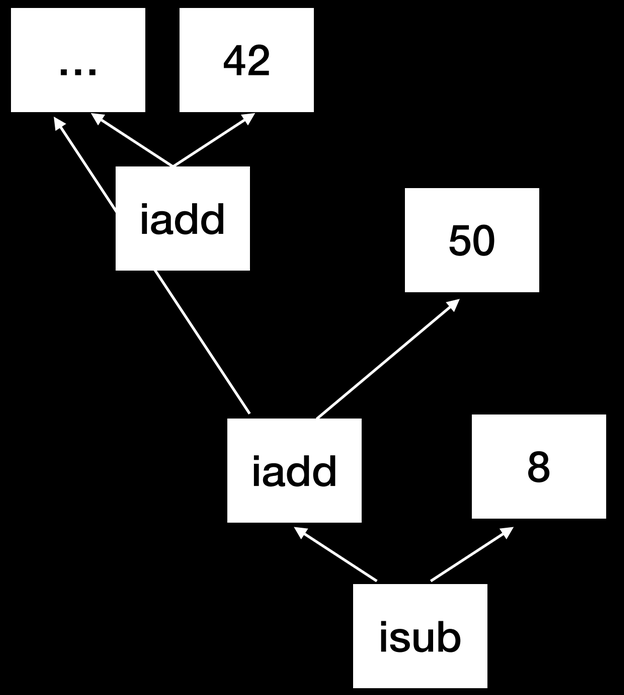


egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

```
v1 = …
v2 = iconst.i32 42
v3 = iadd v1, v2
…
v100 = iconst.i32 50
v101 = iconst.i32 8
v102 = iadd v1, v100
v103 = isub v102, v101
```

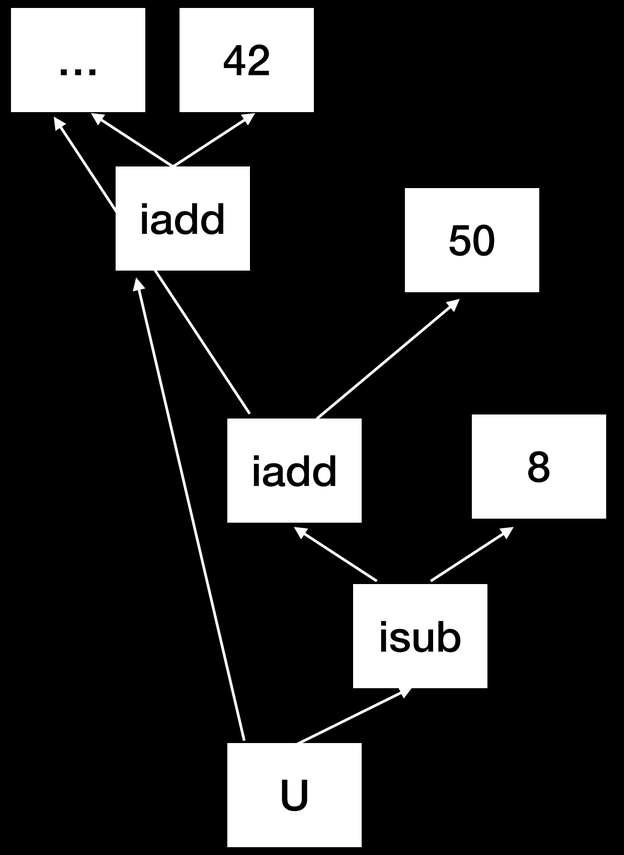


egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

```
v1 = …
v2 = iconst.i32 42
v3 = iadd v1, v2
…
v100 = iconst.i32 50
v101 = iconst.i32 8
v102 = iadd v1, v100
v103 = isub v102, v101
```

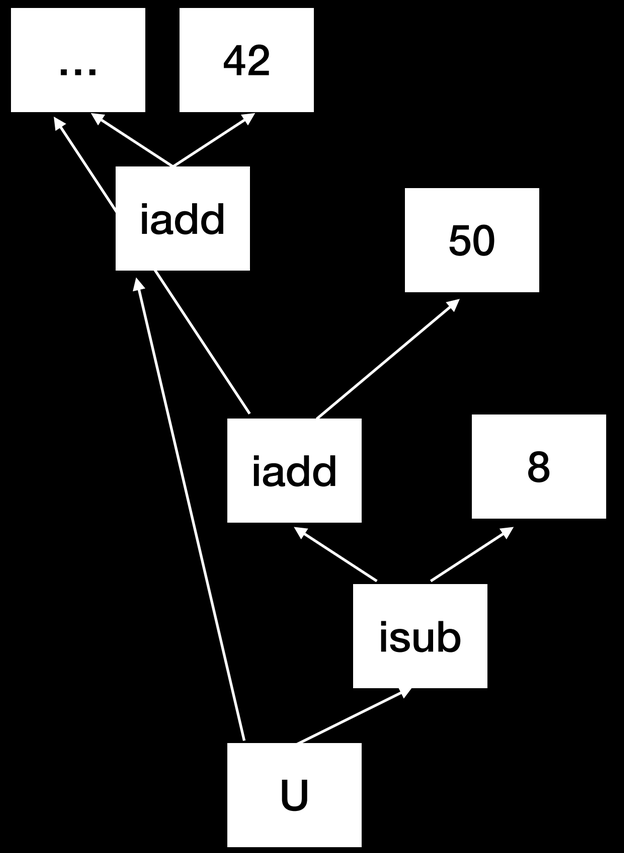


egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

```
v1 = …
v2 = iconst.i32 42
v3 = iadd v1, v2
…
v100 = iconst.i32 50
v101 = iconst.i32 8
v102 = iadd v1, v100
v103 = isub v102, v101
```



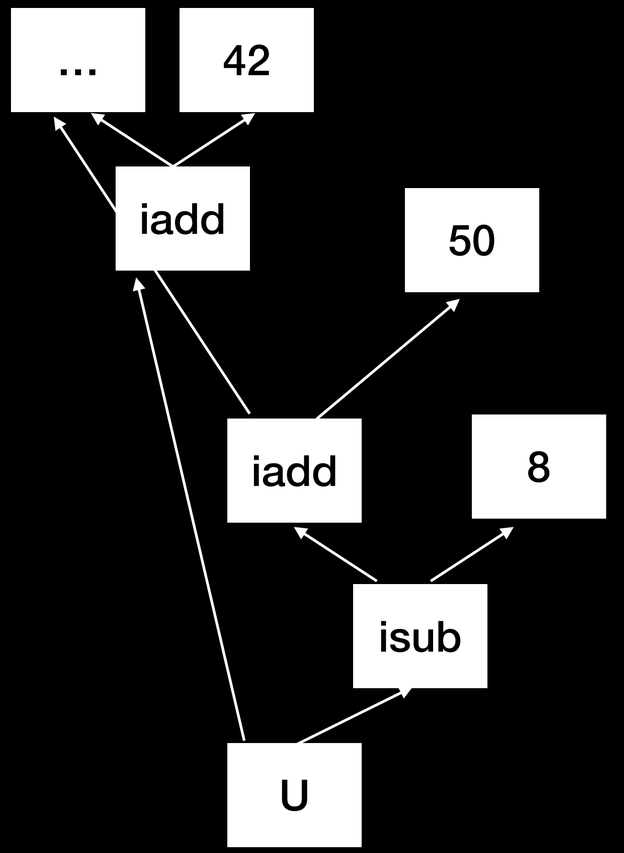
When does this produce suboptimal results?

egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

```
v1 = …
v2 = iconst.i32 42
v3 = iadd v1, v2
…
v100 = iconst.i32 50
v101 = iconst.i32 8
v102 = iadd v1, v100
v103 = isub v102, v101
```



When does this produce suboptimal results?

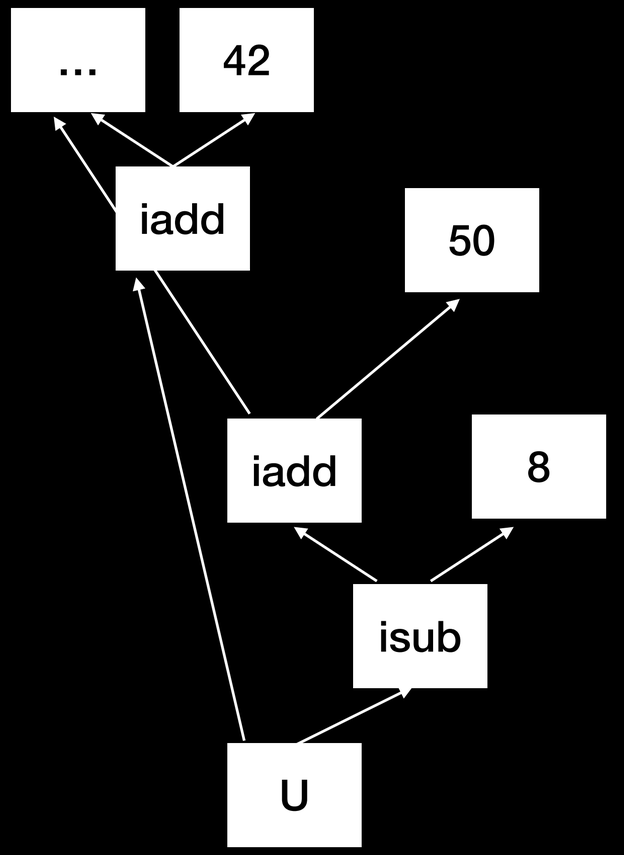
A: when we have a rule f -> g where cost(g) > cost(f)

egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

```
v1 = …
v2 = iconst.i32 42
v3 = iadd v1, v2
…
v100 = iconst.i32 50
v101 = iconst.i32 8
v102 = iadd v1, v100
v103 = isub v102, v101
```



When does this produce suboptimal results?

A: when we have a rule f -> g where cost(g) > cost(f)

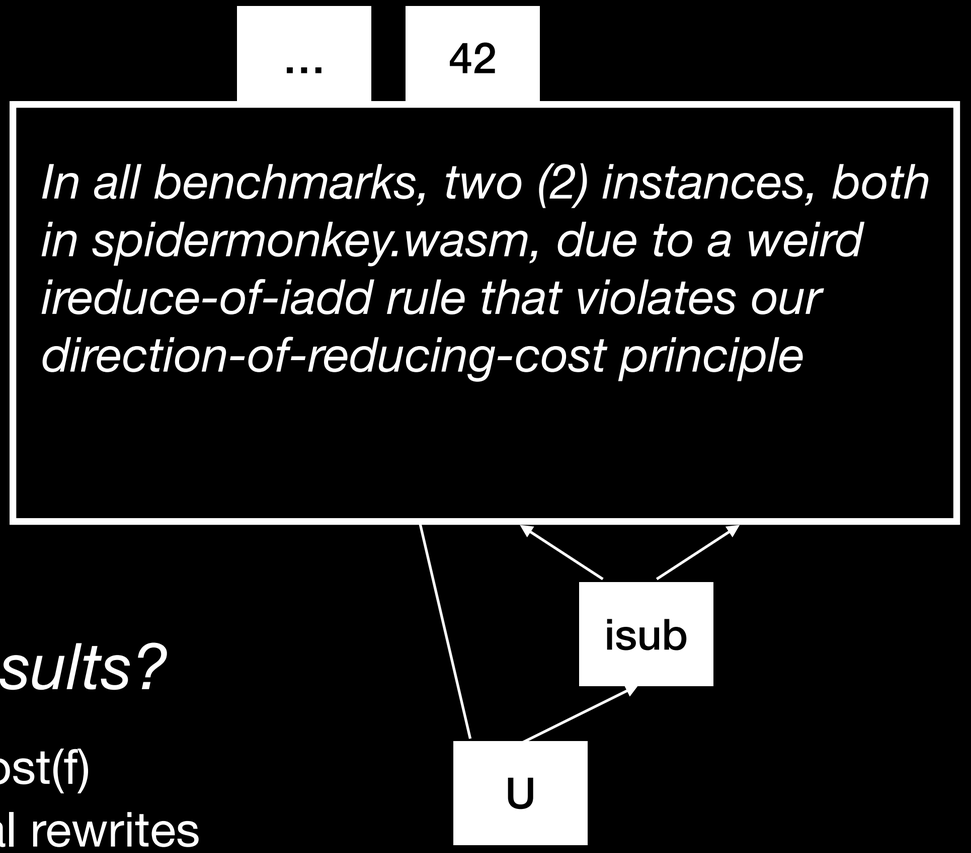
Essentially: bidirectional equality (eqsat) vs directional rewrites

egraphs without congruence closure

(or a union-find!)

… and hopefully no *new* enodes are union'd in later

```
v1 = …
v2 = iconst.i32 42
v3 = iadd v1, v2
…
v100 = iconst.i32 50
v101 = iconst.i32 8
v102 = iadd v1, v100
v103 = isub v102, v101
```



When does this produce suboptimal results?

A: when we have a rule f -> g where cost(g) > cost(f)

Essentially: bidirectional equality (eqsat) vs directional rewrites

How we write rules (our cost calculus)

Rules are cheap: efficient rule compiler (one combined LHS), apply once

How we write rules (our cost calculus)

Rules are cheap: efficient rule compiler (one combined LHS), apply once

Rewrites are expensive: try not to do them unless we’re confident we are improving something

How we write rules (our cost calculus)

Rules are cheap: efficient rule compiler (one combined LHS), apply once

Rewrites are expensive: try not to do them unless we're confident we are improving something

Hence: lean away from composition and intermediate terms, and toward "manual inlining"

How we write rules (our cost calculus)

Rules are cheap: efficient rule compiler (one combined LHS), apply once

Rewrites are expensive: try not to do them unless we're confident we are improving something

Hence: lean away from composition and intermediate terms, and toward "manual inlining"

- N.B.: intermediate ISLE terms are ephemeral (compiled away); only external effects (creating IR nodes) remain

How we write rules (our cost calculus)

Rules are cheap: efficient rule compiler (one combined LHS), apply once

Rewrites are expensive: try not to do them unless we're confident we are improving something

Hence: lean away from composition and intermediate terms, and toward "manual inlining"

- N.B.: intermediate ISLE terms are ephemeral (compiled away); only external effects (creating IR nodes) remain

Also: *subsume* wherever appropriate (e.g. aggressively for cprop)

Is this still an egraph? (Or: what are we getting?)

- **Multi-value representation** —> more downstream rewrites
- **Global fixpoint loop** (implicitly GVN; can incorporate other analyses)
- **Congruence closure** —> rules can interact indirectly

Is this still an egraph? (Or: what are we getting?)

- **Multi-value representation** —> more downstream rewrites
- **Global fixpoint loop** (implicitly GVN; can incorporate other analyses)
- **Congruence closure** —> rules can interact indirectly

Recall: these were the original motivations!

Is this still an egraph? (Or: what are we getting?)

- **Multi-value representation** —> more downstream rewrites
- **Global fixpoint loop** (implicitly GVN; can incorporate other analyses)
- ~~**Congruence closure** —> rules can interact indirectly~~

Recall: these were the original motivations!

Is this still an egraph? (Or: what are we getting?)

- **Multi-value representation** —> more downstream rewrites
  - We can evaluate this! Eagerly cull to one choice
- **Global fixpoint loop** (implicitly GVN; can incorporate other analyses)

- ~~**Congruence closure** —> rules can interact indirectly~~

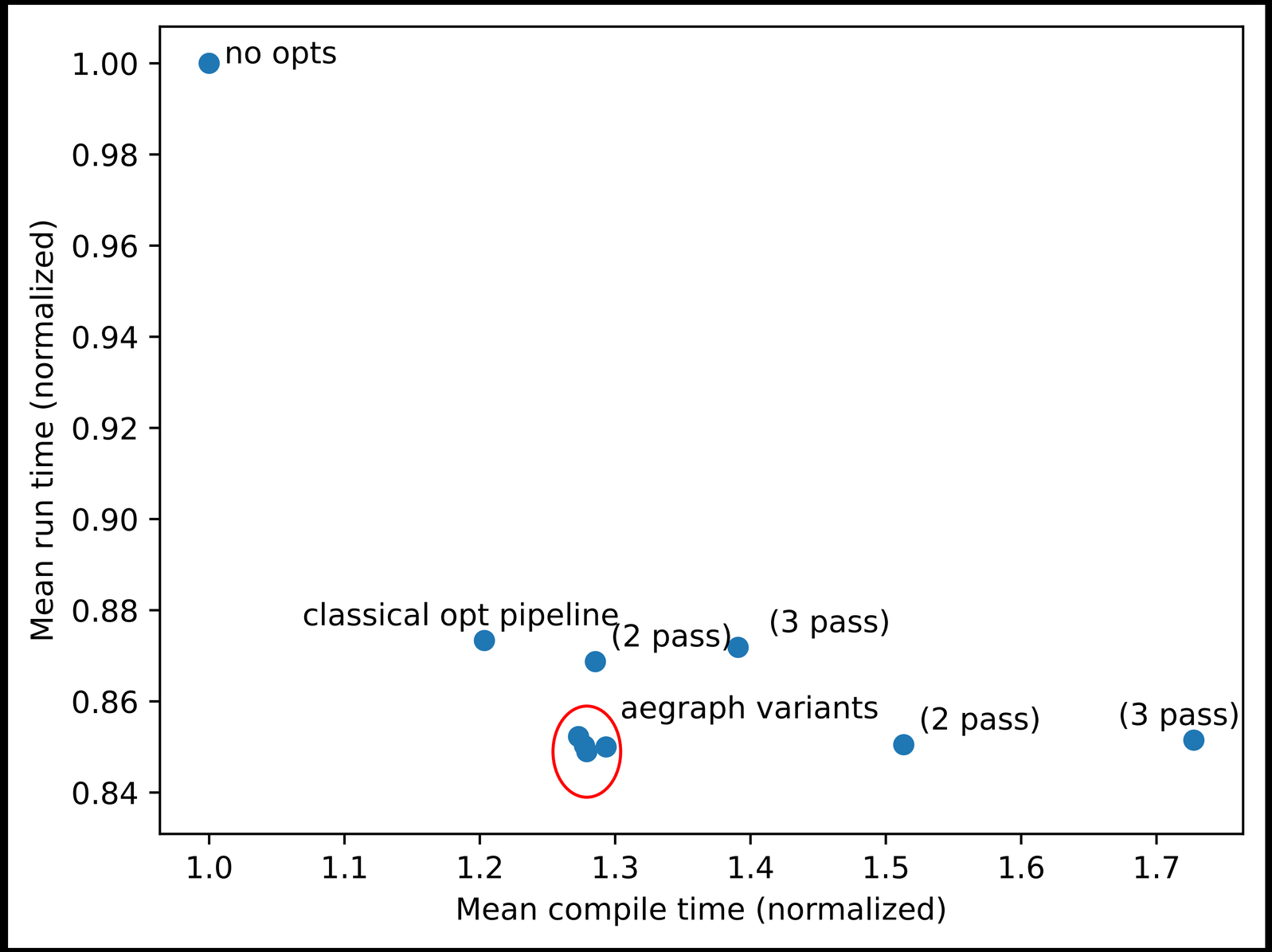
Is this still an egraph? (Or: what are we getting?)

- **Multi-value representation** —> more downstream rewrites
  - We can evaluate this! Eagerly cull to one choice
- **Global fixpoint loop** (implicitly GVN; can incorporate other analyses)
  - We can evaluate this! Compare to a traditional opt pipeline with same rules
- ~~**Congruence closure** —> rules can interact indirectly~~

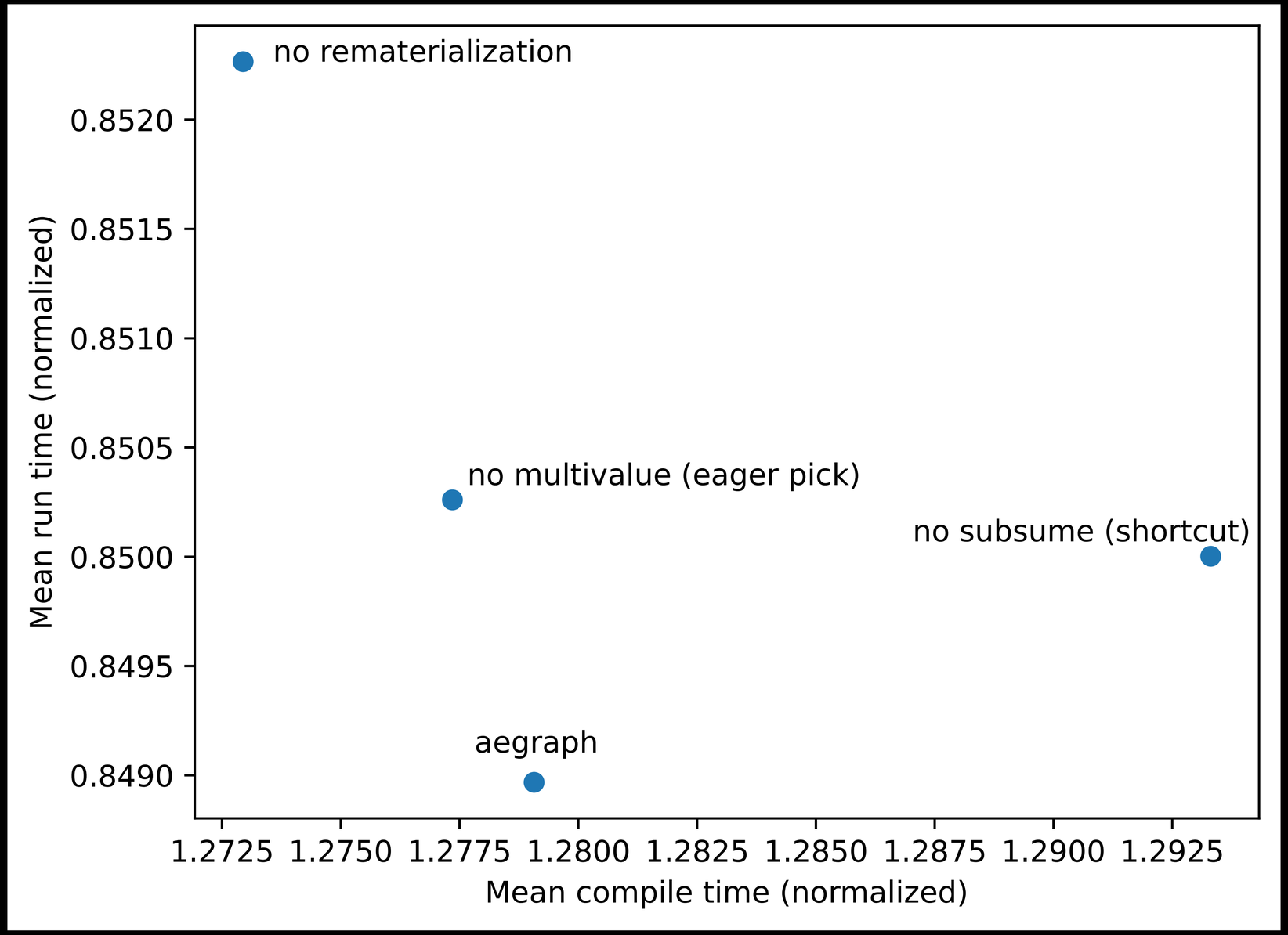
Results

- Instrumented/modified Cranelift+Wasmtime over Sightglass benchmarks
  - “Large”: spidermonkey, gcc-loops, hashset, regex, meshoptimizer, blind-signatures, Markdown renderer, bzip2
  - Microbenchmarks: qsort, intgemm-simd, blake3 cipher, richards, hex
- 23,389 functions with 4,260,474 pure insts
- Compile time, runtime (cycles, 50 runs), instrumented stats

Results


no opts
classical opt pipeline
(2 pass)
(3 pass)
aegraph variants
(2 pass)
(3 pass)
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0.84
Mean run time (normalized)
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
Mean compile time (normalized)

Results


no rematerialization
no multivalue (eager pick)
no subsume (shortcut)
aegraph
0.8520
0.8515
0.8510
0.8505
0.8500
0.8495
0.8490
Mean run time (normalized)
1.2725
1.2750
1.2775
1.2800
1.2825
1.2850
1.2875
1.2900
1.2925
Mean compile time (normalized)

Discussion

- Multi-value representation (i.e., eclass) effect is within the noise?!

- Hypotheses
  - Our rule discipline has minimized actual multiple representation?
    - Maybe: average eclass size is 1.13 nodes
  - Cost function needs improvement?
  - We are not exposing opportunity to where it could help (the backend)?
  - Code from compiled Wasm modules is already fairly optimized?

Cost Function

- Approximations to the combinatorial optimization problem?

  - account for overlap / structure sharing (transitive-closure set)
  - dynamically update costs somehow (as we elaborate, “lock in” sunk costs)
  - use instruction selection matcher rules as a natural cost function

Cost Function

- Approximations to the combinatorial optimization problem?

  - account for overlap / structure sharing (transitive-closure set)

  - dynamically update costs somehow (as we elaborate, “lock in” sunk costs)

  - use instruction selection matcher rules as a natural cost function

- Insight: in “conventional compiler”, cost func + extraction is everything
  - Overlaps with classically hard + important problems like instruction selection, register allocation (reg pressure, rematerialization)
  - Multi-rooted extraction fundamentally changes the problem

Conclusion

- aegraphs in Cranelift has been a successful project
  - We replaced the mid-end, saw no regressions, and solved the motivating issues
  - Over time, the use of term-rewriting has allowed significant growth in rules

- Today we mainly see benefit in:
  - Unified fixpoint loop
  - Code motion (via elaboration — “sea of nodes” effect)
  - Maybe a *tiny* bit in multi-value representation

- We’re heavily constrained by compile time, but maybe there is still opportunity:
  - Unified mid-end + backend (elaborate-while-we-lower; pick best inst seq)
  - Smaller cost-function tweaks?

Thanks!