TurboFan JIT Design

Ben L. Titzer
Google Munich

V8 Background

- JavaScript has some difficult to optimize features
  - No explicit types
  - Prototype-based property lookup
  - Dynamic evaluation of code
- V8 was the first really *fast* JavaScript VM
  - Sophisticated and efficient object layout
  - Compile-only: no interpreter
    - Quick, non-optimizing JIT (fullcode)
    - Inline caching, type feedback
    - Generational GC with low pause times
- V8 launched with Chrome in 2008
  - Optimizing JIT (CrankShaft) launched in 2010

TurboFan Design Goals

- Achieve best peak performance
  - Highest quality machine code
  - Within normal constraints of JIT compilation

- Make best use of static type information
  - asm.js, latent JavaScript types, TypeScript, SoundScript proposal

- Reduce platform-specific implementation effort
  - Better separation between front, middle, and backend of compiler

- Improve testability
  - Prevent correctness bugs and verify optimizations activate

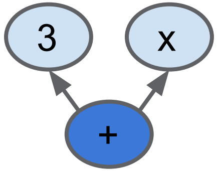
TurboFan Program Representation (IR)

- NOT: Control Flow Graph (CFG)
  - Fully-specified evaluation order; e.g. pure operations like integer addition

- INSPIRATION: Sea of Nodes
  - Relax evaluation order for most operations
  - Effect edges order stateful operations
  - Skeleton of a CFG remains
  - Why? Better redundant code elimination, more code motion

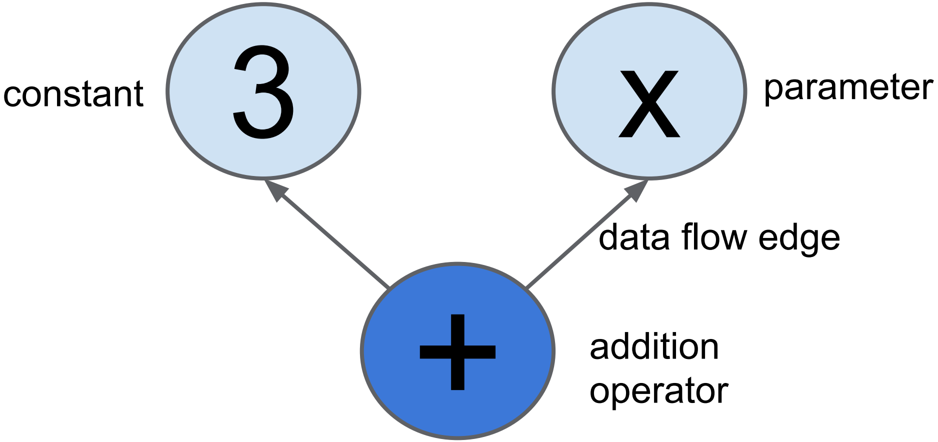
- REALLY: “Soup” of Nodes
  - Relax Sea of Nodes control flow subgraph even further
  - Disconnected “floating control” islands offer more scheduling freedom
  - Why? Lowering of language levels, even more code motion

Do not get seasick!

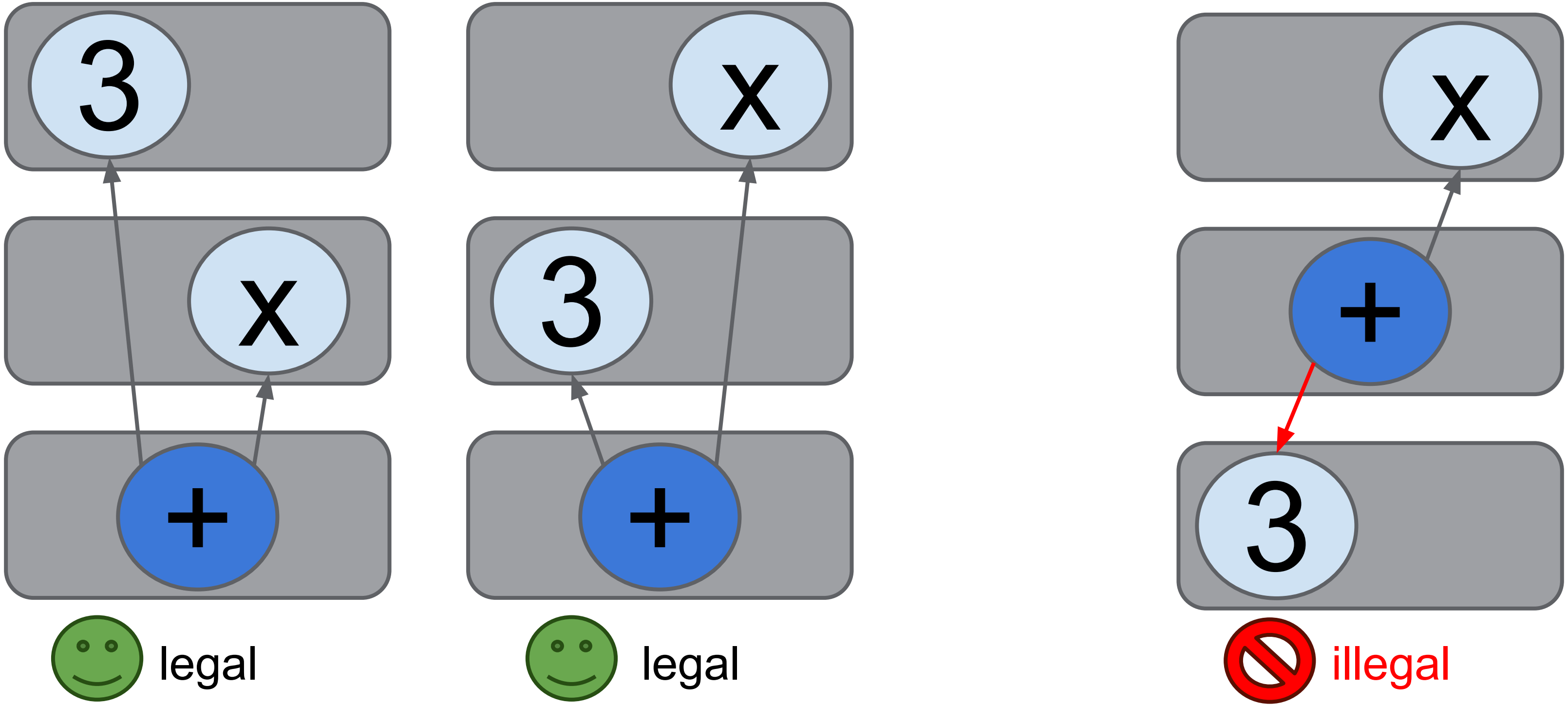


All computations are expressed as nodes in the sea of nodes

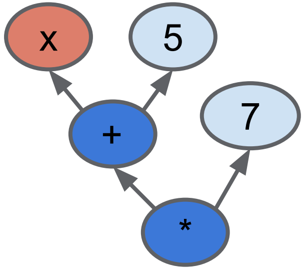
Edges represent dependencies between computations



Dependencies constrain Ordering



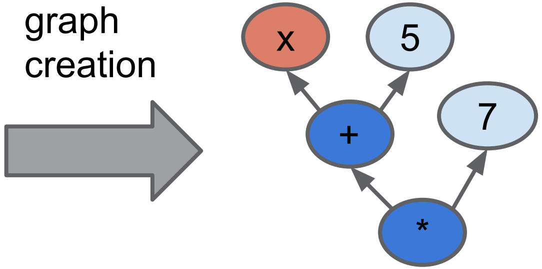
Lack of Ordering means Compiler Freedom!



x	x	x	x	5	5	5	5	7	7
5	5	5	7	x	x	x	7	x	5
+	7	7	5	+	7	7	x	5	x
7	+	+	+	7	+	+	+	+	+
*	*	*	*	*	*	*	*	*	*

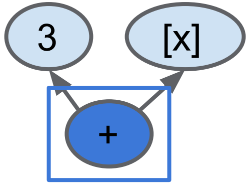
Larger class of equivalences

x	x	x	x	5	5	5	5	7	7
5	5	5	7	x	x	x	7	x	5
+	7	7	5	+	7	7	x	5	x
7	+	+	+	7	+	+	+	+	+
*	*	*	*	*	*	*	*	*	*



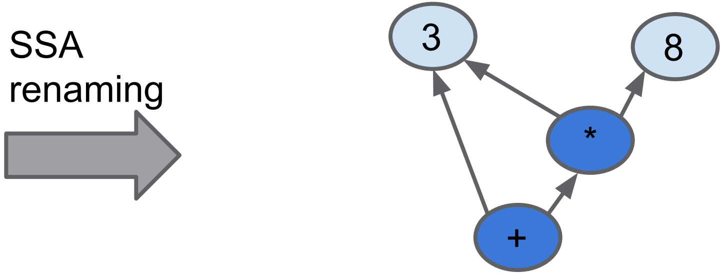
The Sea is SSA

```
x = 3 * 8
x + 3
```



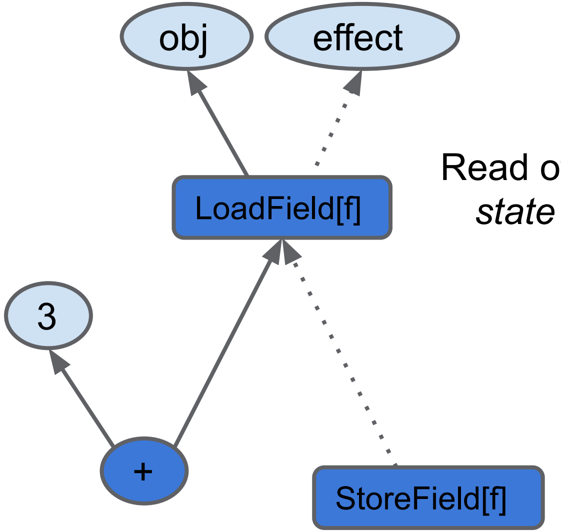
No such thing as local variables!

Graph building from source renames locals

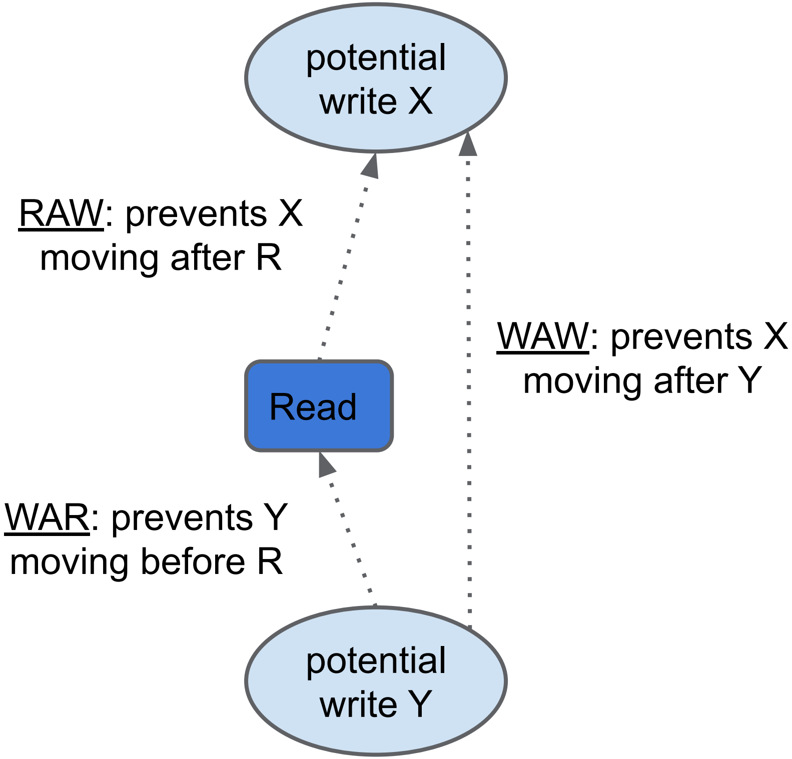

SSA renaming

Multiple incoming edges possible

Effect Edges



Read of *mutable state* `obj.f`



<u>RAW</u>: prevents X moving after R

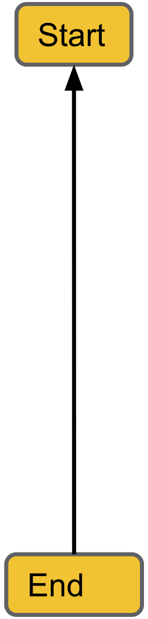
<u>WAW</u>: prevents X moving after Y

<u>WAR</u>: prevents Y moving before R

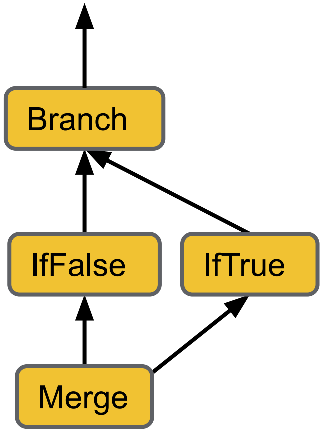
Expressing Control

- Nodes: express computation
  - Constants, parameters, arithmetic, load, store, calls
  - Source program is SSA renamed so locals are substituted with nodes
- Edges: express dependencies (constrain order)
  - dataflow edges express using the value output of a computation
  - effect edges order operations reading and writing state
- **NEXT**: Control with start, branches, loops, merge, and end
  - How do we express non-straight line code?

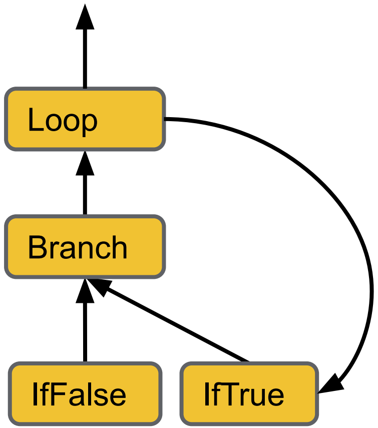
Control nodes and Control edges


Start
End

straightline program


Branch
IfFalse
IfTrue
Merge

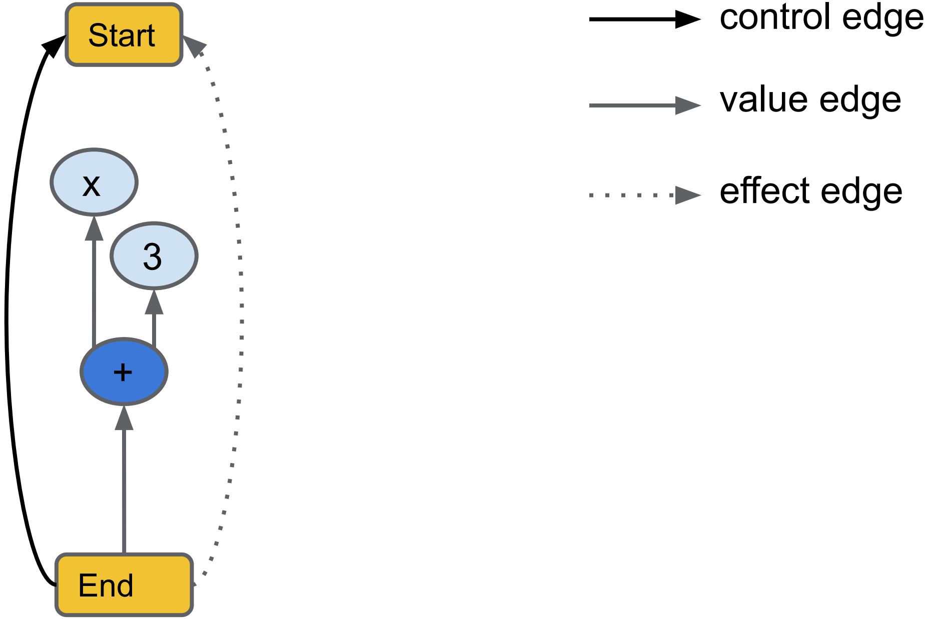
branch


Loop
Branch
IfFalse
IfTrue

while loop

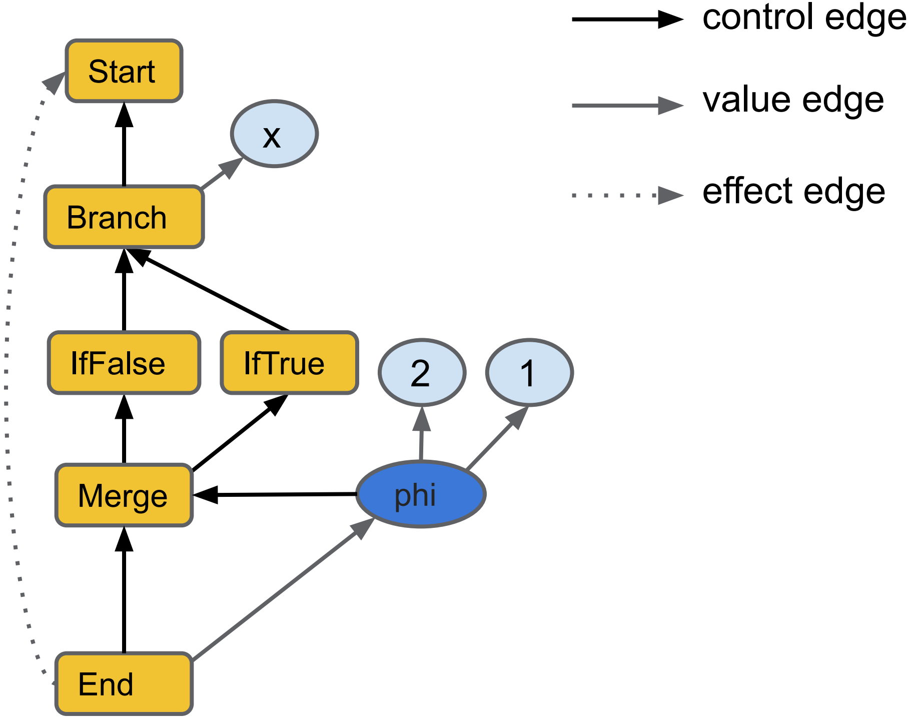
Our first complete graph

function (x) { return x + 3; }



Branch example

function (x) { return x ? 1 : 2; }



Language Levels

- JavaScript: ("JS") operators
  - Express semantics of JavaScript's overloaded operators
  - Produce and consume effects in the graph
- Intermediate: ("Simplified") operators
  - Express VM-level operations, such as allocation, bounds checks
  - Arithmetic independent of number representation
- Machine: ("Machine") operators
  - Correspond closely to single machine instructions
  - Most have no side effects
  - Must be supported by backend for each platform

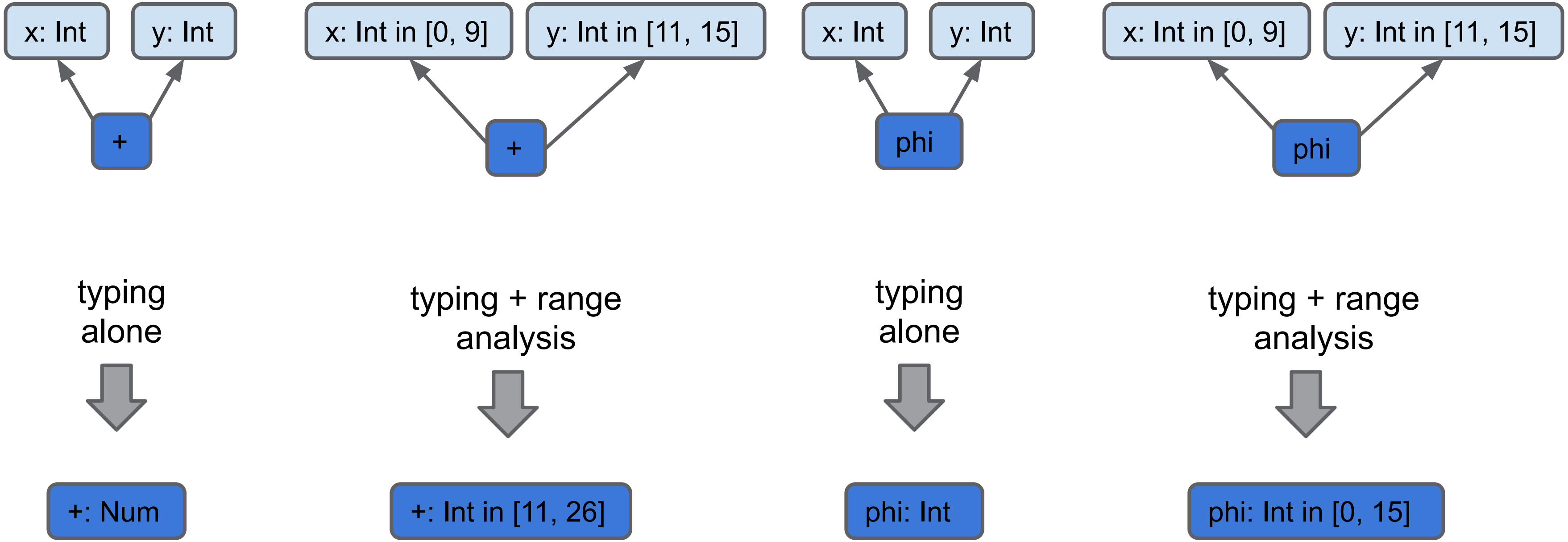
Language Levels

- JavaScript: JSAdd JSSubtract JSMultiply JSDivide JSModulus JSDivide JSBitwiseOr JSBitwiseAnd JSBitwiseXor JSShiftLeft JSShiftRight JSEqual JSStrictEqual JSToBoolean JSToNumber JSCall
- Intermediate: NumberAdd NumberSub NumberMul NumberDiv NumberMod NumberEqual NumberLessThan LoadField StoreField StringEqual StringAdd ChangeTaggedToInt32
- Machine: Int32Add Int32Sub Int32Mul Float64Add Float64Sub Float64Mul Load Store Call ConvertFloat64ToInt32

Type and Range Analysis

- JavaScript is not statically typed
  - Values have types, not variables
  - `8` is a Number, `“x”` is a String
  - All basic operators (`+ - * / % == !=`) overloaded for objects
- All arithmetic is done in 64-bit floating point
  - Empirically, most programs use only small integers (<= 31bits)
  - Overflow to double usually causes code to bailout to slow path
  - Troublesome cases: `NaN, Infinity, -Infinity, -0.0`
- asm.js language subset
  - Annotations such as `(x + y) | 0`
  - Truncation maps `NaN, Infinity, -Infinity, -0.0` to integer `0`

Type and Range Analysis

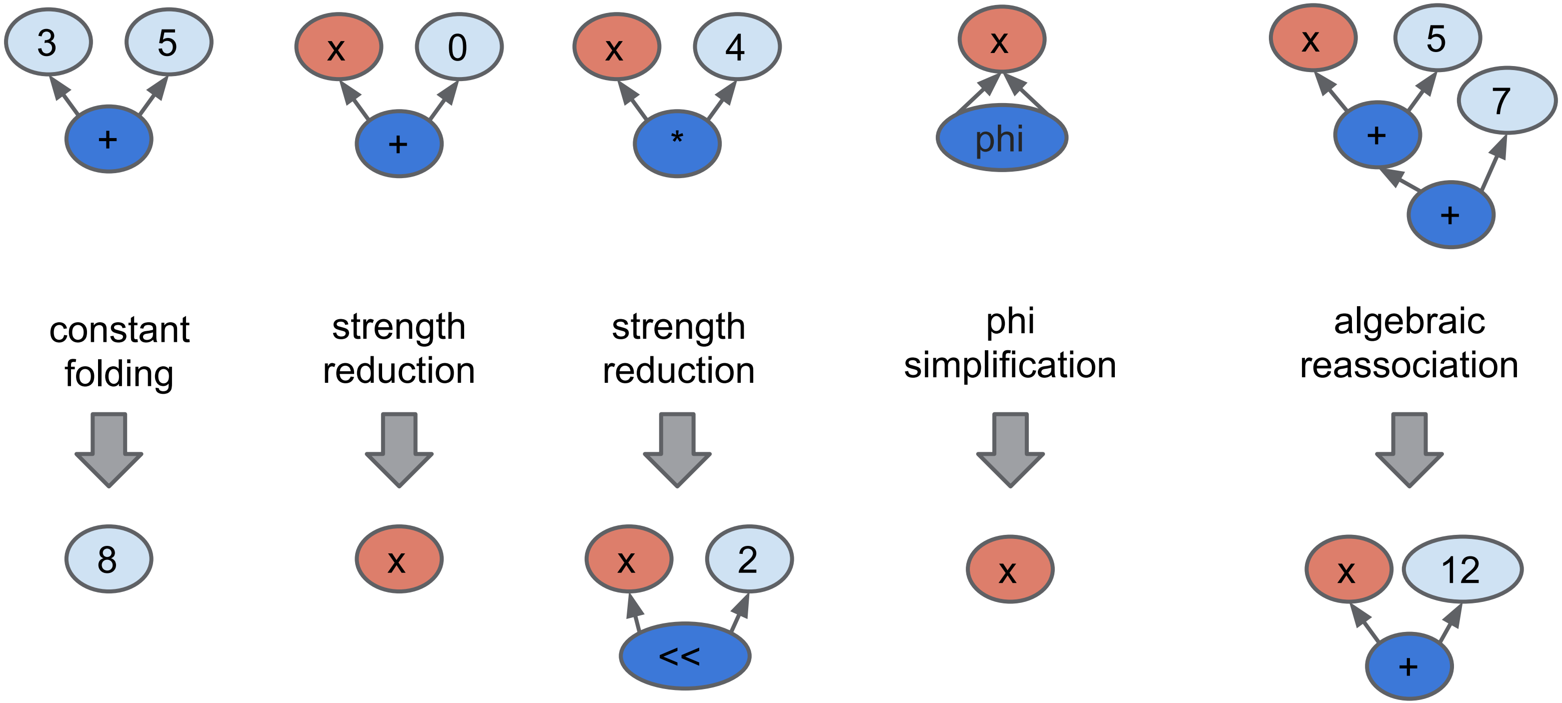


Optimization

- Nearly all optimization happens on the sea of nodes
  - Top-down or bottom-up graph transformations
  - Isolates transformations from error-prone ordering of computations
  - Local reasoning leads to incremental transformations

- Reachability => Liveness
  - Nodes not reachable from end are *dead*
    - Including dead control, dead effects, dead computation
  - Most phases never see dead code
  - Dead code never placed in final schedule

Reduction



constant folding

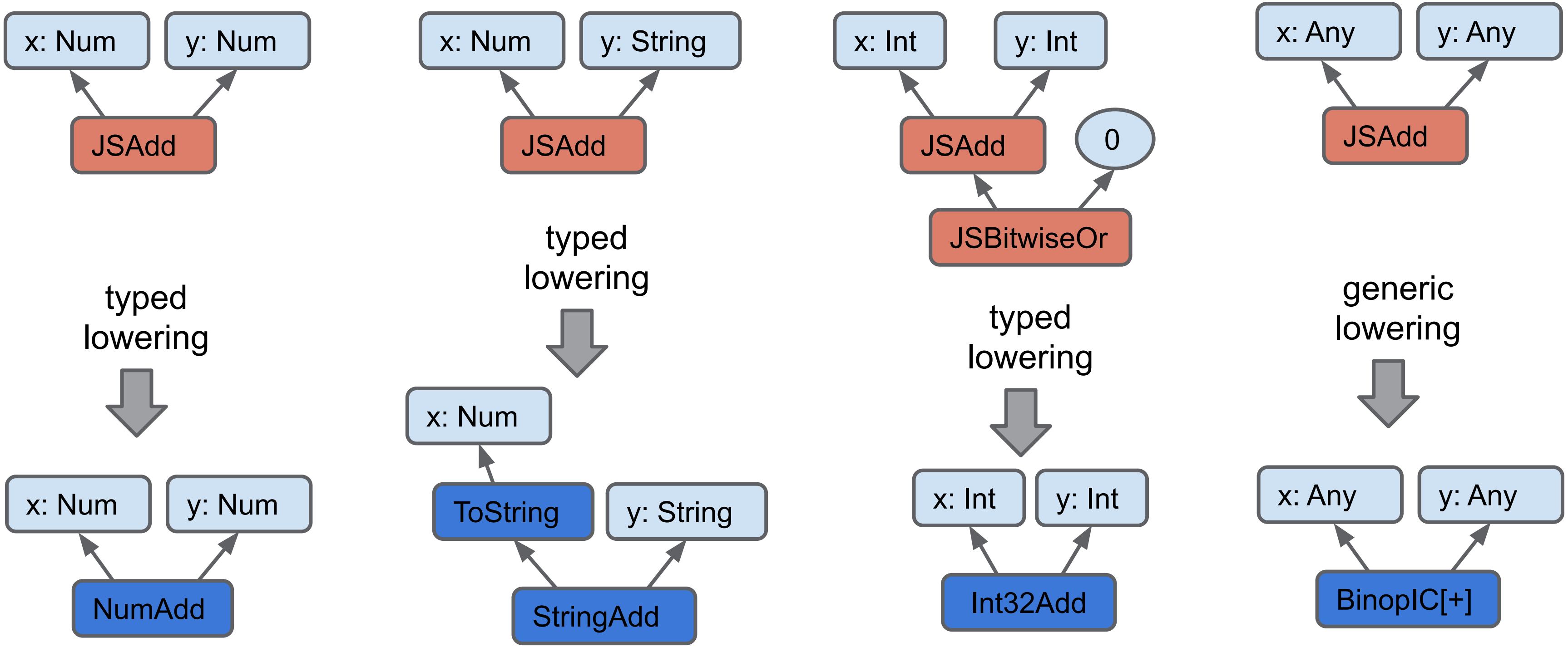
strength reduction

strength reduction

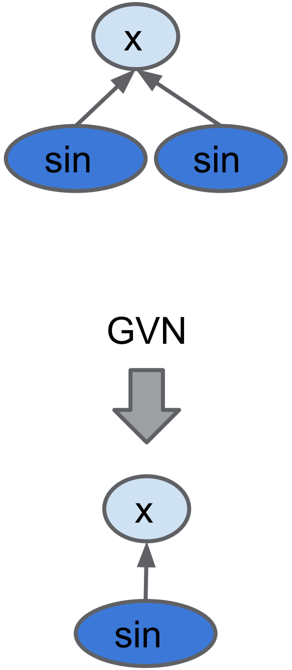
phi simplification

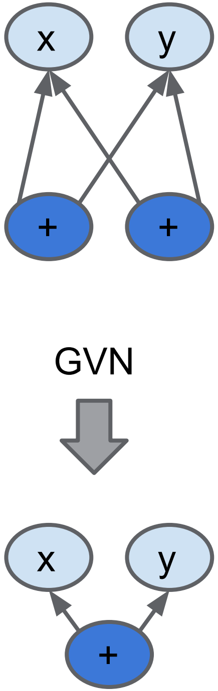
algebraic reassociation

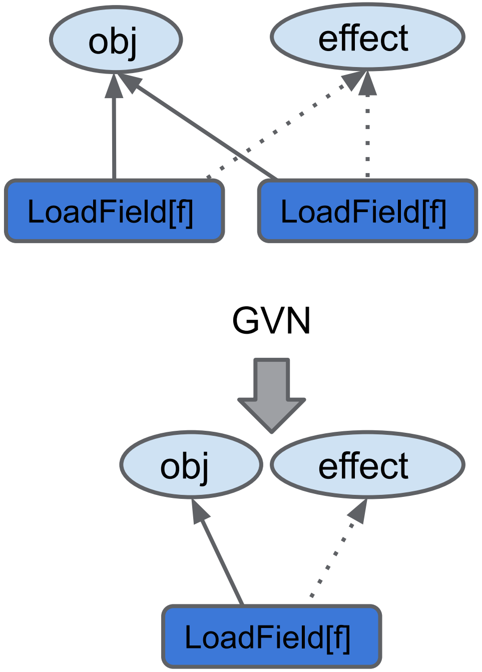
Typed Lowering as Reduction



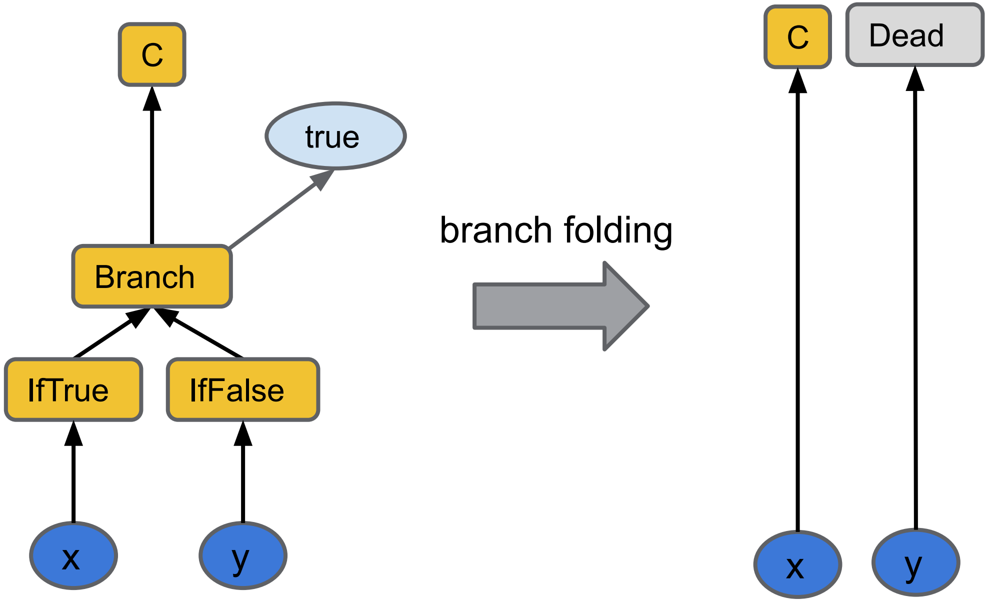
Global Value Numbering as Reduction



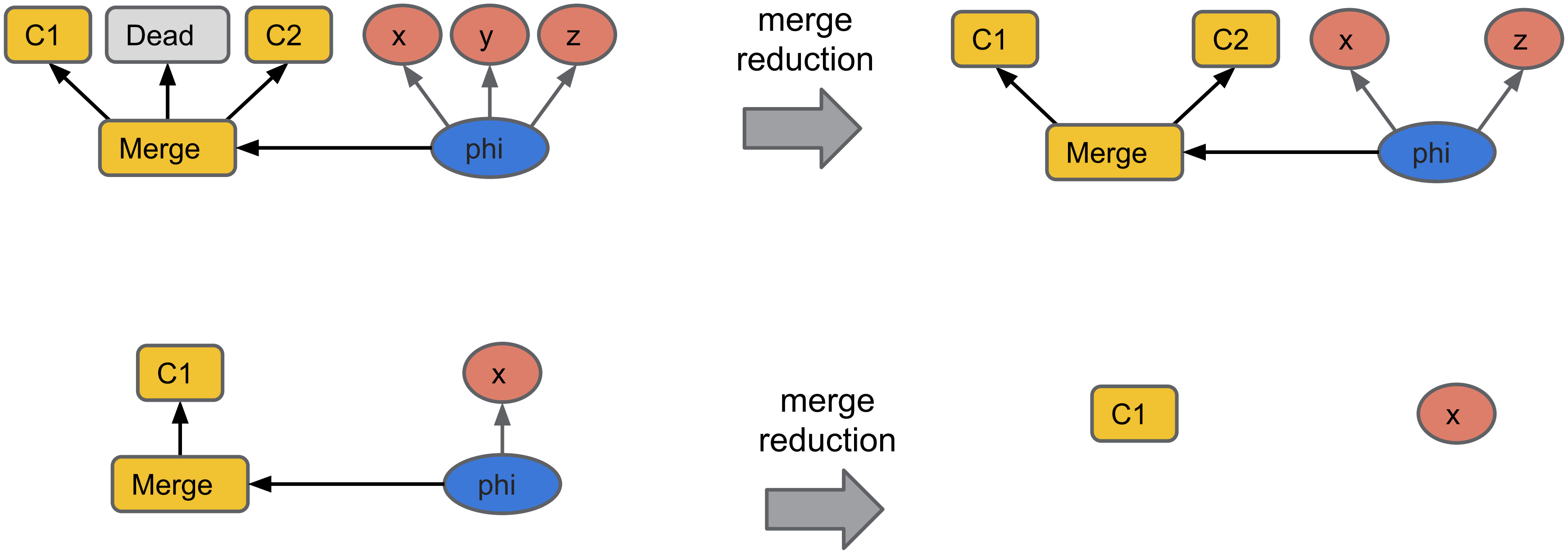




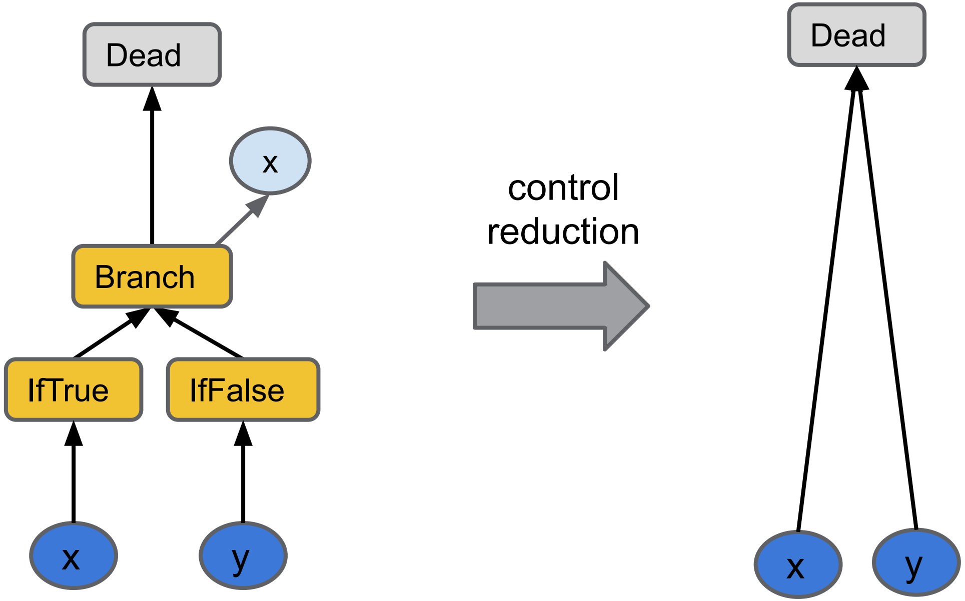
Control Optimization as Reduction



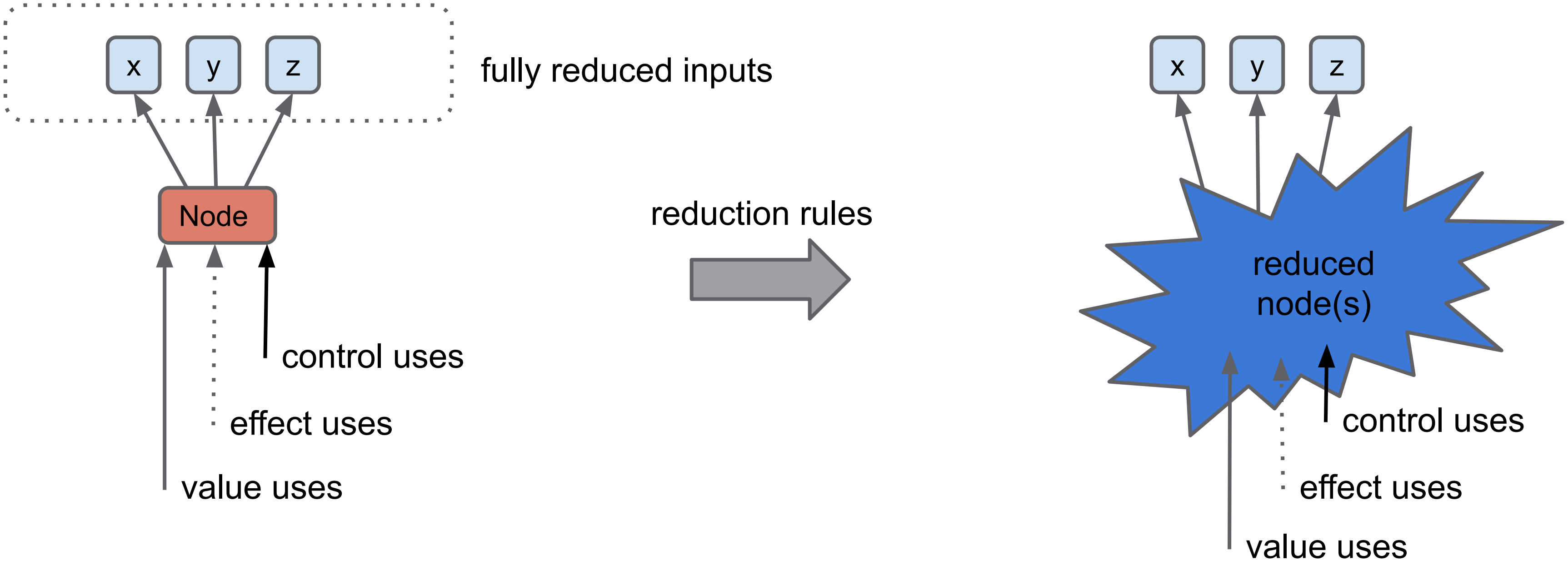
Control Optimization as Reduction



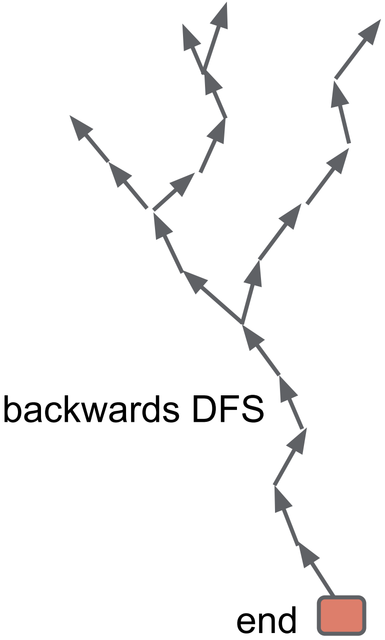
Control Optimization as Reduction

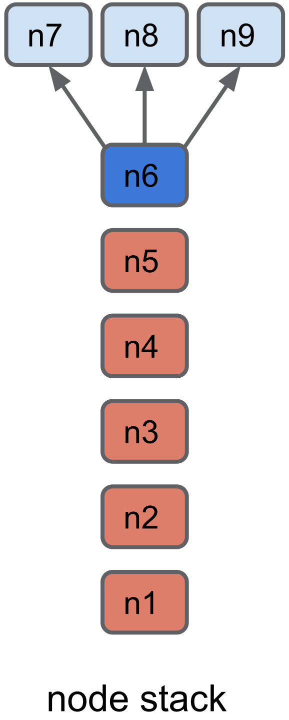


Reduction as Top-down Graph Rewriting



Iterative Reduction (recursion with explicit stack)



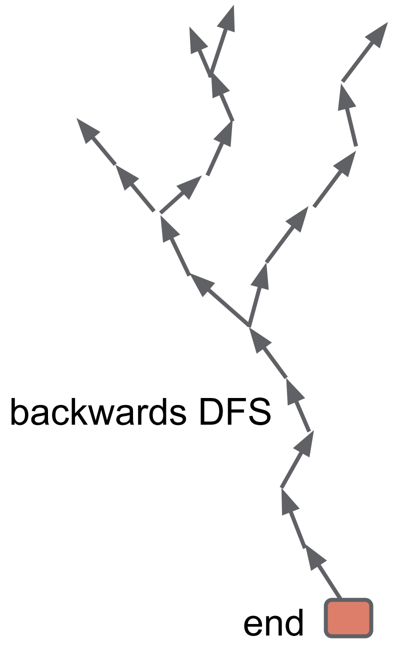


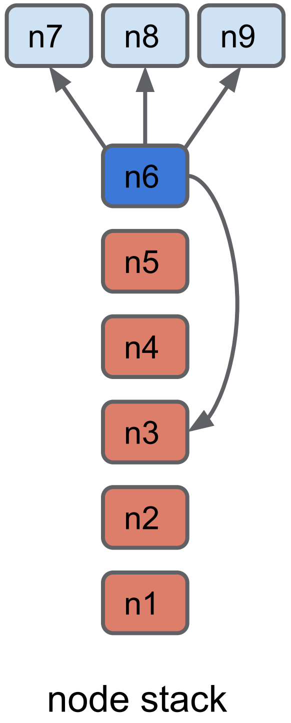
Reduce top of stack when all its inputs are reduced.

Pop the stack after applying reduction rules.

Applies reduction to each node once in the optimal order.

Iterative Reduction (in the presence of cycles)



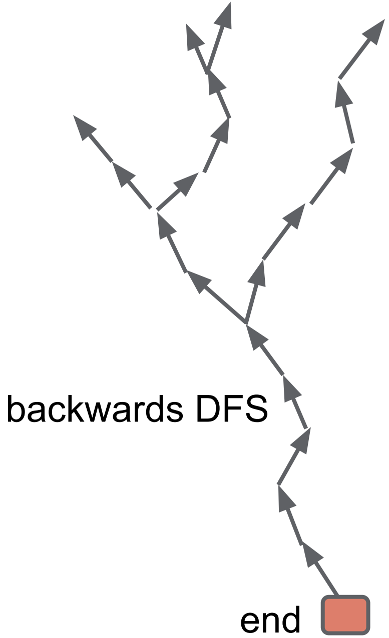


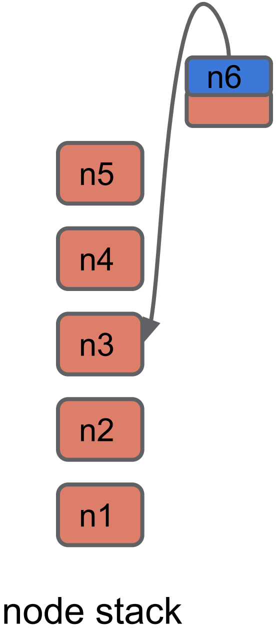
Reduce top of stack when all its inputs are either reduced ***or are themselves on the stack*.**

When a node is successfully reduced, revisit any uses that were partially reduced due to cycles.

Computes a fixpoint over reduction rules.

Iterative Reduction (in the presence of cycles)



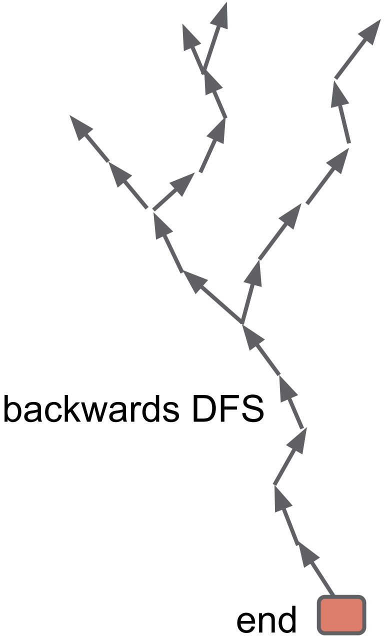


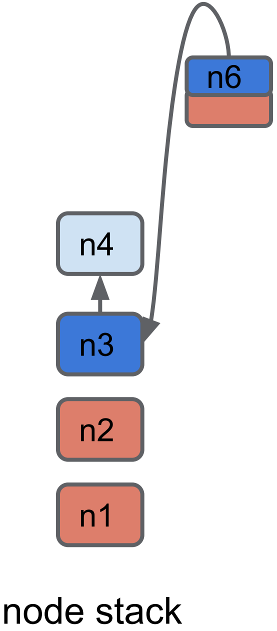
Reduce top of stack when all its inputs are either reduced ***or are themselves on the stack*.**

When a node is successfully reduced, revisit any uses that were partially reduced due to cycles.

Computes a fixpoint over reduction rules.

Iterative Reduction (in the presence of cycles)



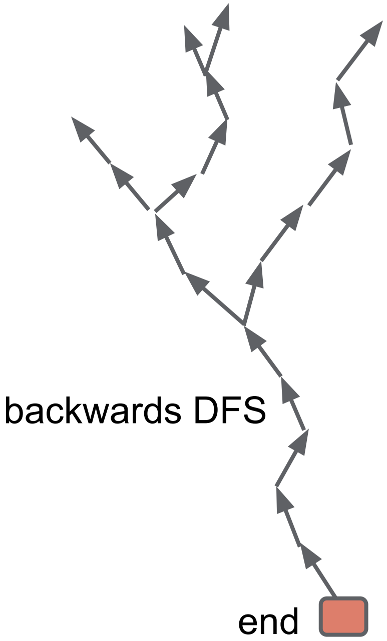


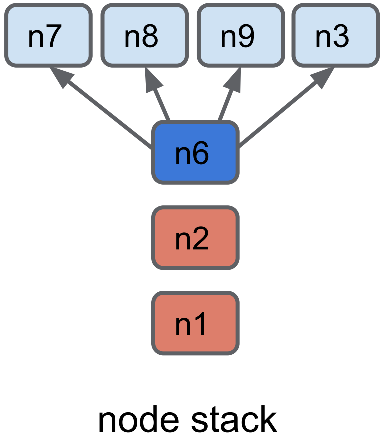
Reduce top of stack when all its inputs are either reduced or are themselves on the stack.

When a node is successfully reduced, ***revisit any uses that were partially reduced due to cycles*****.**

Computes a fixpoint over reduction rules.

Iterative Reduction (in the presence of cycles)





Reduce top of stack when all its inputs are either reduced or are themselves on the stack.

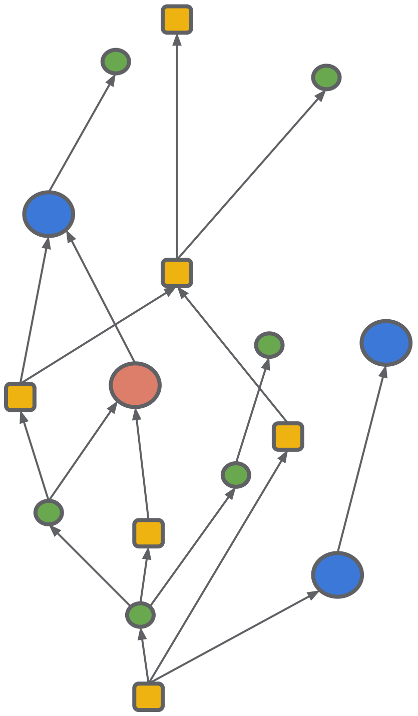
When a node is successfully reduced, revisit any uses that were partially reduced due to cycles.

Computes a fixpoint over reduction rules.

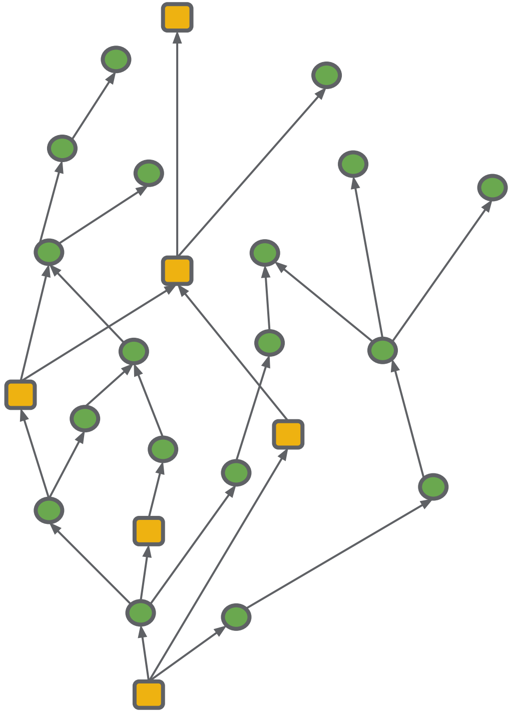
Lowering to Machine

- JavaScript: JSAdd JSSubtract JSMultiply JSDivide JSModulus JSDivide JSBitwiseOr JSBitwiseAnd JSBitwiseXor JSShiftLeft JSShiftRight JSEqual JSStrictEqual JSToBoolean JSToNumber JSCall
- Intermediate: NumberAdd NumberSub NumberMul NumberDiv NumberMod NumberEqual NumberLessThan LoadField StoreField StringEqual StringAdd ChangeTaggedToInt32
- Machine: Int32Add Int32Sub Int32Mul Float64Add Float64Sub Float64Mul Load Store Call ConvertFloat64ToInt32

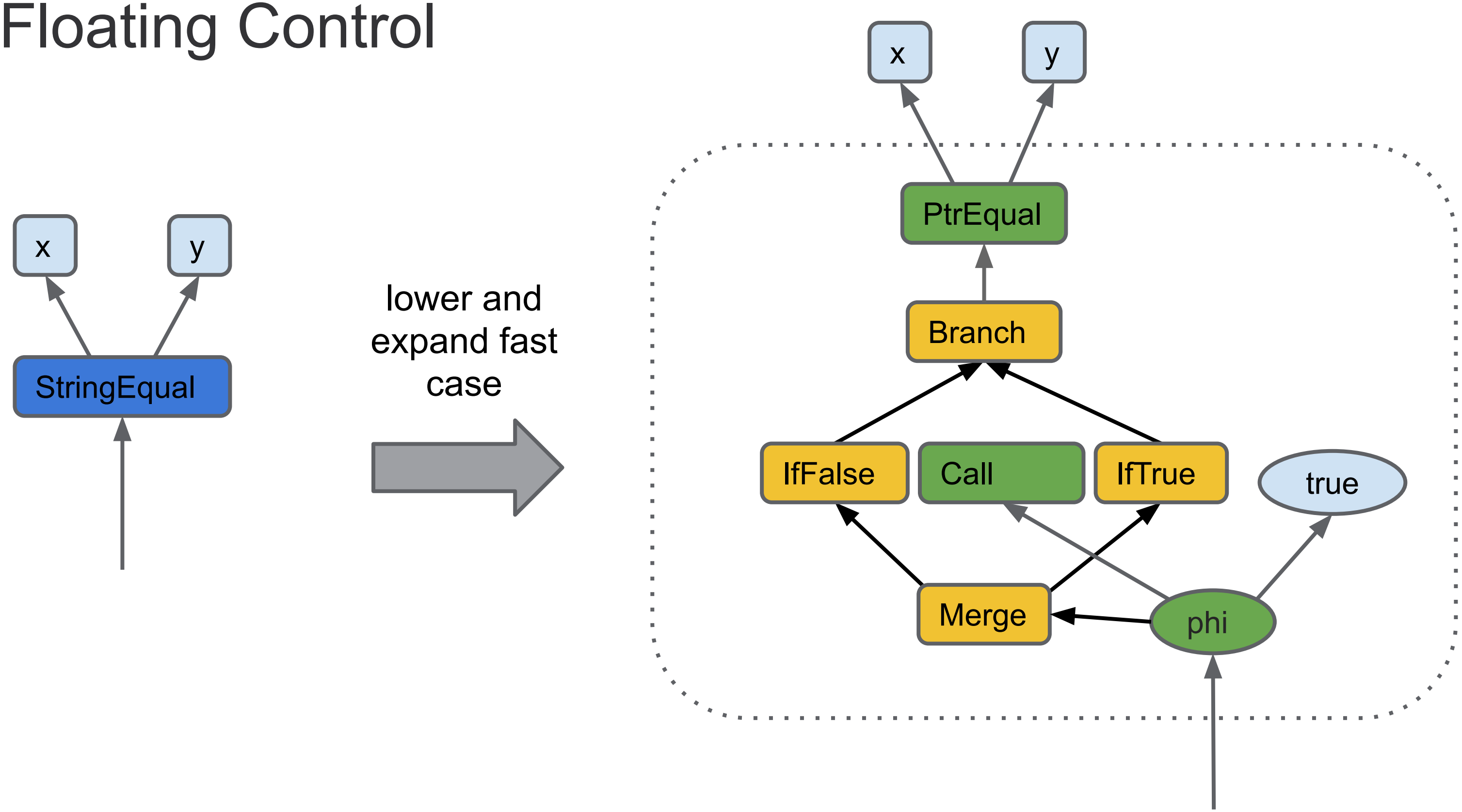
Lowering to Machine



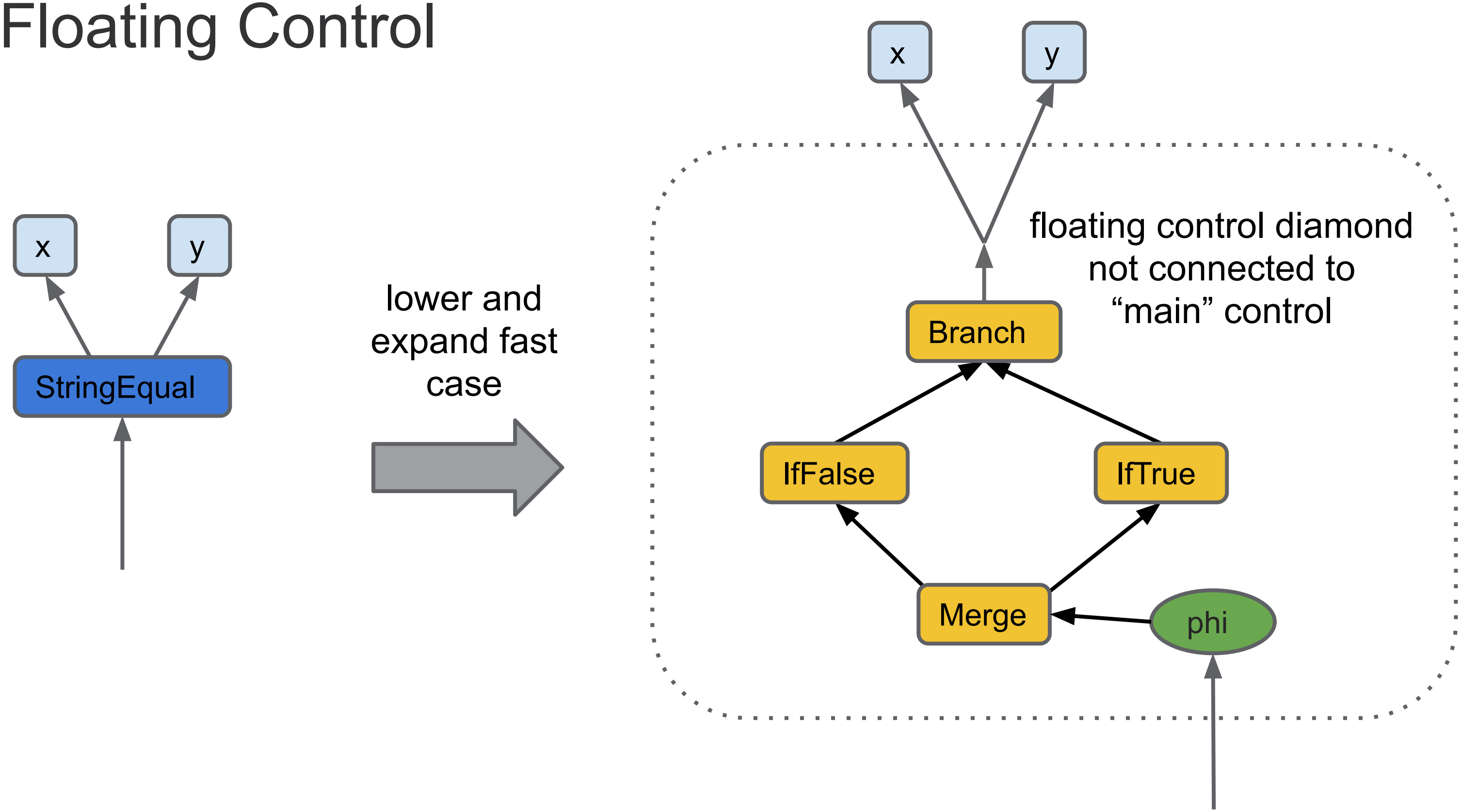
expand and optimize
JS* and Simplified*
nodes



Floating Control



Floating Control


x
y
StringEqual
lower and expand fast case
x
y
floating control diamond not connected to “main” control
Branch
IfFalse
IfTrue
Merge
phi

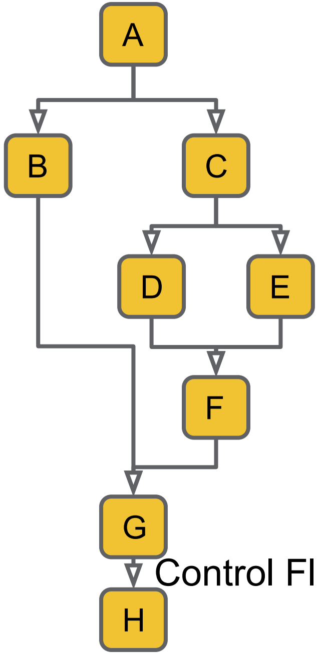
Scheduling the Sea of Nodes

- Sea of nodes expresses many possible legal orderings of code
  - Many possible CFGs
  - Many possible assignment of nodes to CFG blocks
  - Many possible orderings within basic blocks

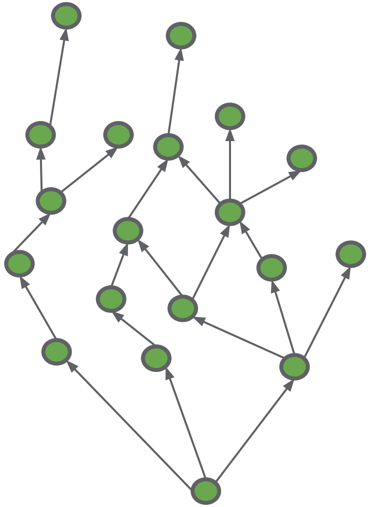
- What is the most efficient order and placement?
  - Depends on control dominance, loop nesting, register pressure

- Outcome: traditional CFG
  - Traditional code generation and register allocation can take over

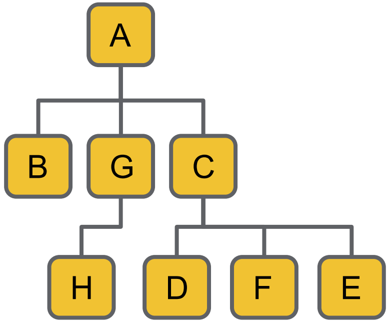
Scheduling the Sea of Nodes (sketch)



Control Flow Graph

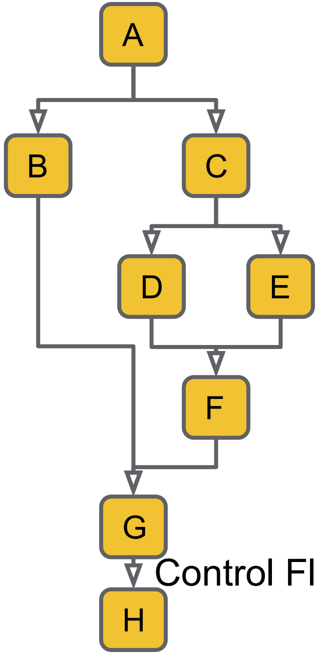


Unscheduled nodes



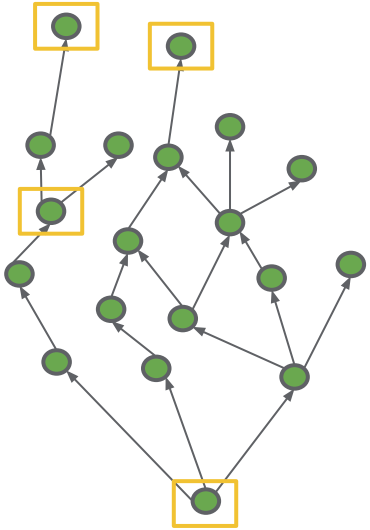
Dominator Tree

Scheduling the Sea of Nodes (sketch)

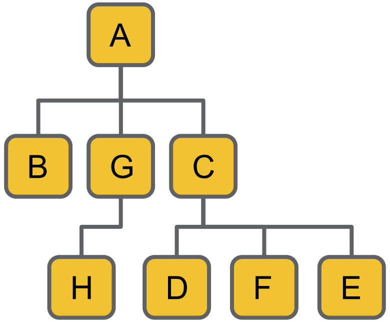


Control Flow Graph

Place fixed nodes (phis, params)

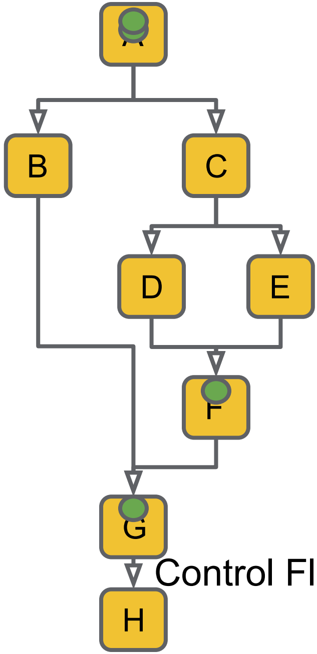


Unscheduled nodes

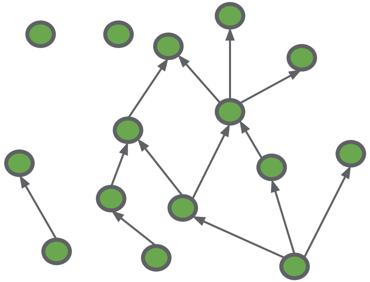


Dominator Tree

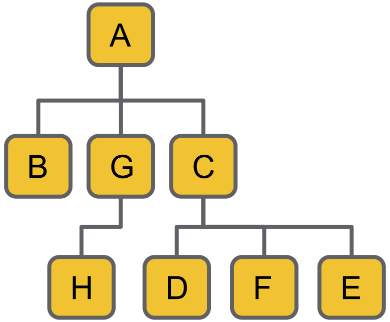
Scheduling the Sea of Nodes (sketch)



Control Flow Graph

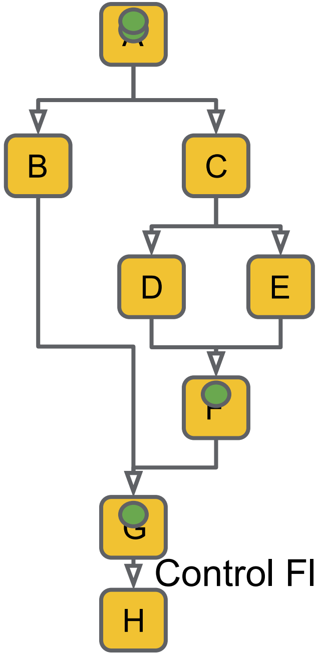


Unscheduled nodes

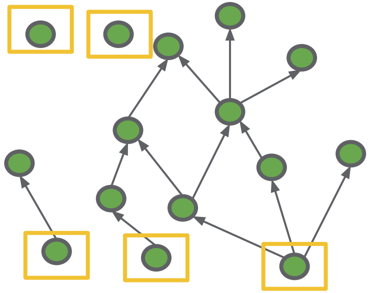


Dominator Tree

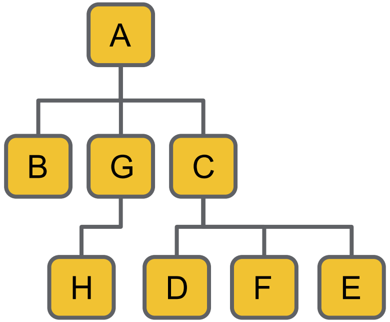
Scheduling the Sea of Nodes (sketch)



Control Flow Graph

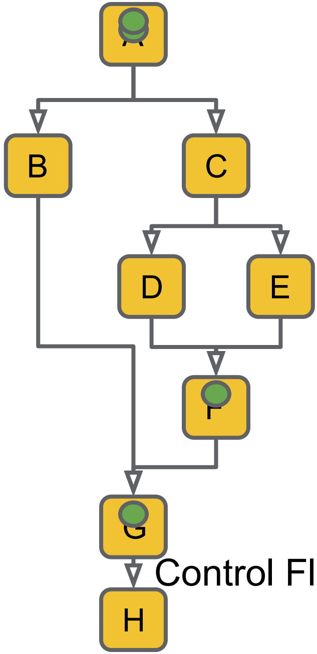


Schedulable nodes



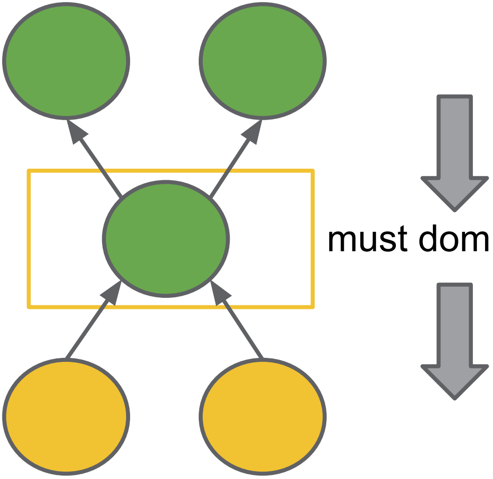
Dominator Tree

Scheduling the Sea of Nodes (sketch)



Control Flow Graph

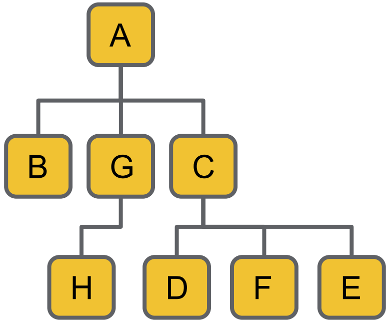
inputs



eligible node

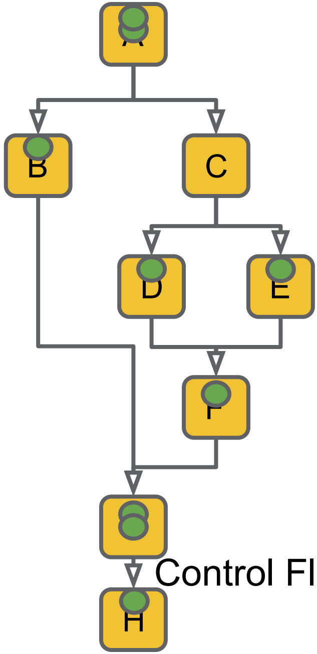
must dominate

scheduled uses

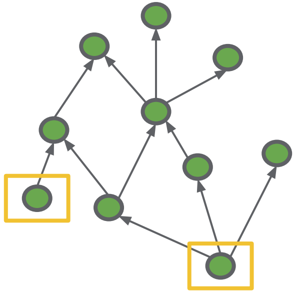


Dominator Tree

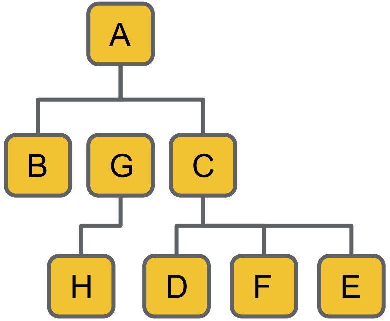
Scheduling the Sea of Nodes (sketch)



Control Flow Graph

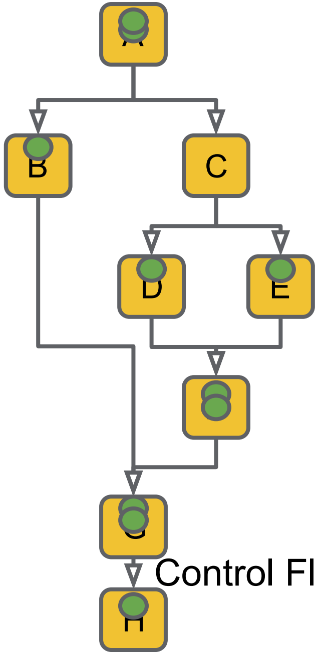


Schedulable nodes

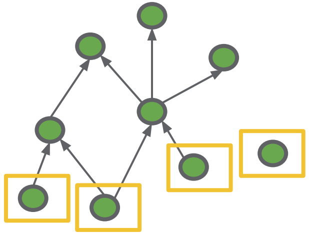


Dominator Tree

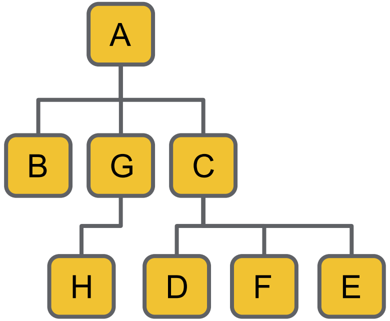
Scheduling the Sea of Nodes (sketch)



Control Flow Graph

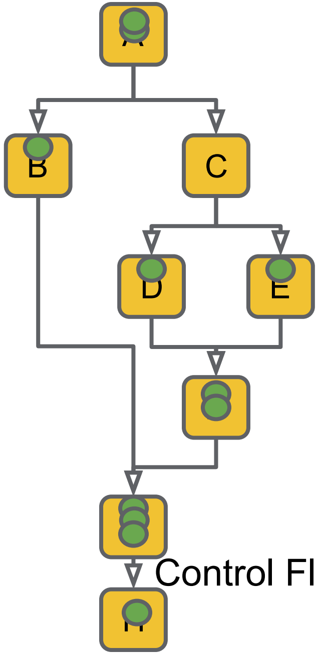


Schedulable nodes

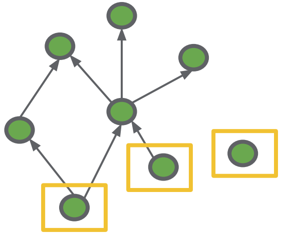


Dominator Tree

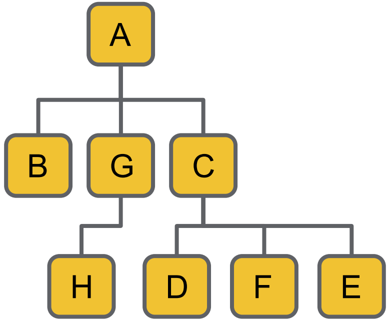
Scheduling the Sea of Nodes (sketch)



Control Flow Graph

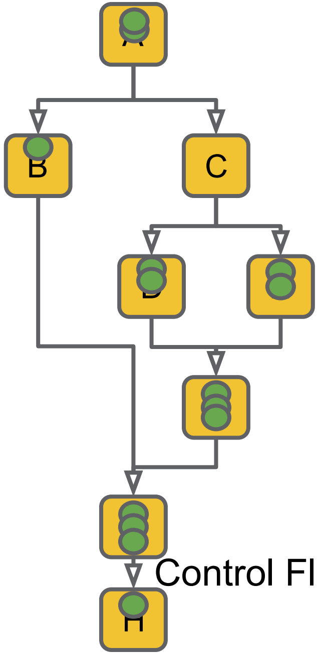


Schedulable nodes

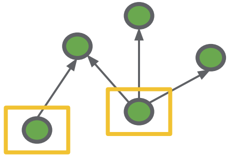


Dominator Tree

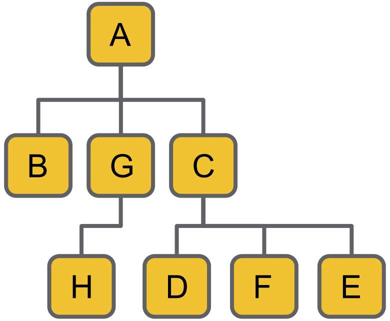
Scheduling the Sea of Nodes (sketch)



Control Flow Graph

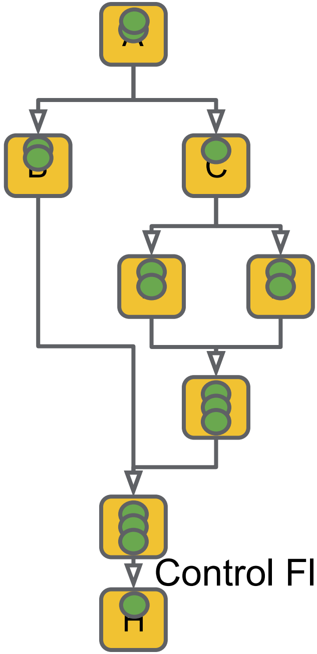


Schedulable nodes

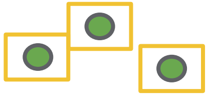


Dominator Tree

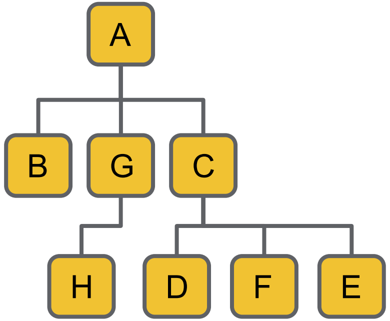
Scheduling the Sea of Nodes (sketch)



Control Flow Graph

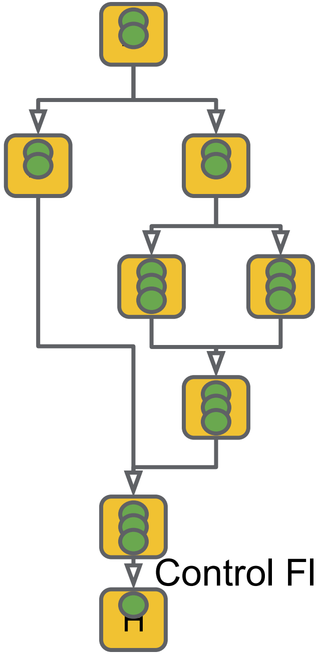


Schedulable nodes



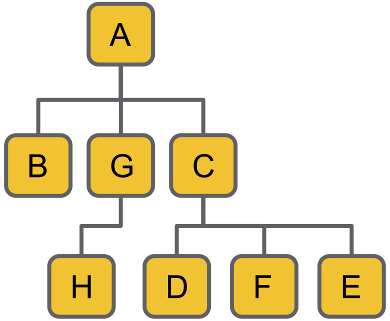
Dominator Tree

Scheduling the Sea of Nodes (sketch)



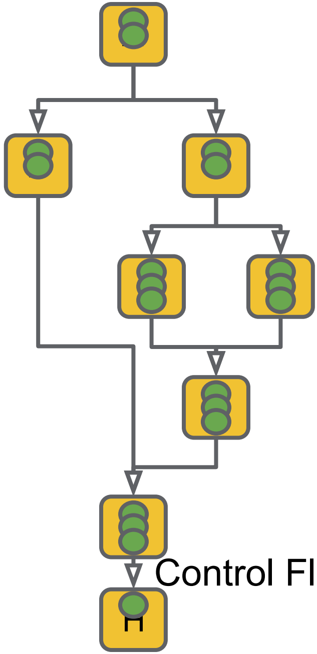
Control Flow Graph

Schedulable nodes



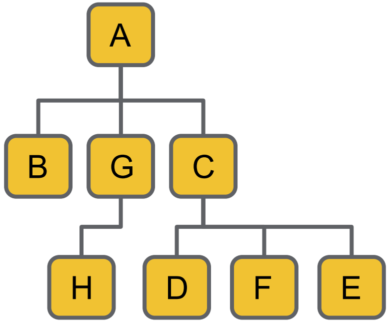
Dominator Tree

Scheduling the Sea of Nodes (sketch)



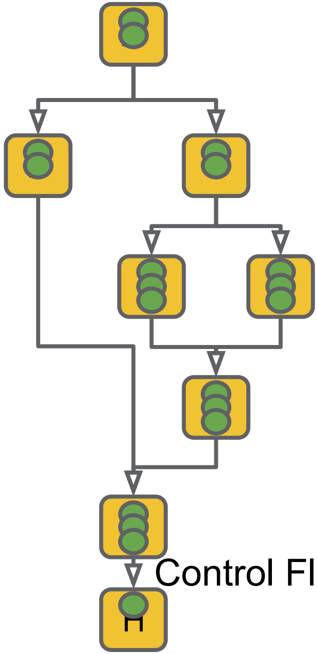
Control Flow Graph

A fully scheduled graph is exactly the same as a CFG.



Dominator Tree

Scheduling the Sea of Nodes (sketch)



Control Flow Graph

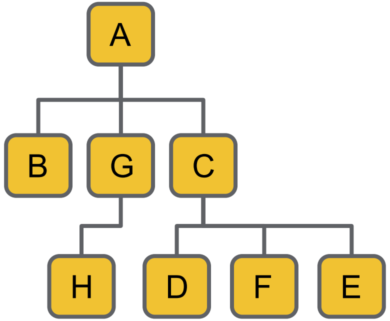
<u>Placement is important!</u>

Hoist code out of loops if possible

Place code as late as possible

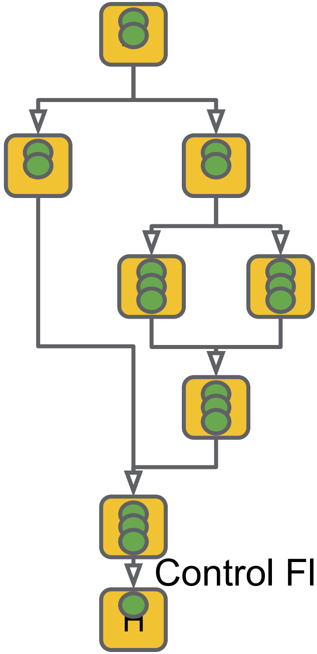
Minimize register pressure

Eliminate redundancy



Dominator Tree

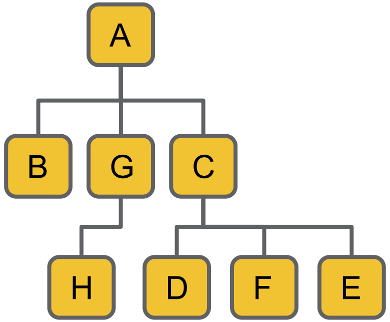
Where is Loop Invariant Code Motion?



Control Flow Graph

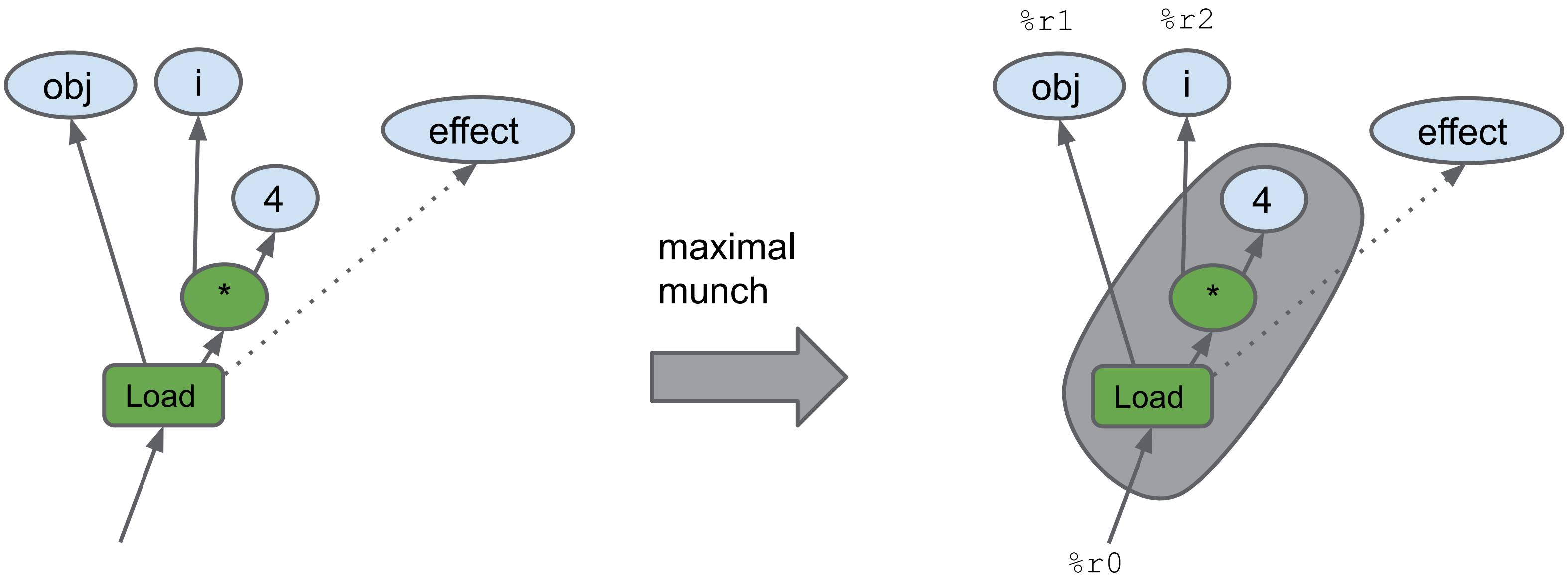
<u>Nowhere!</u>

Scheduling subsumes loop invariant code motion.



Dominator Tree

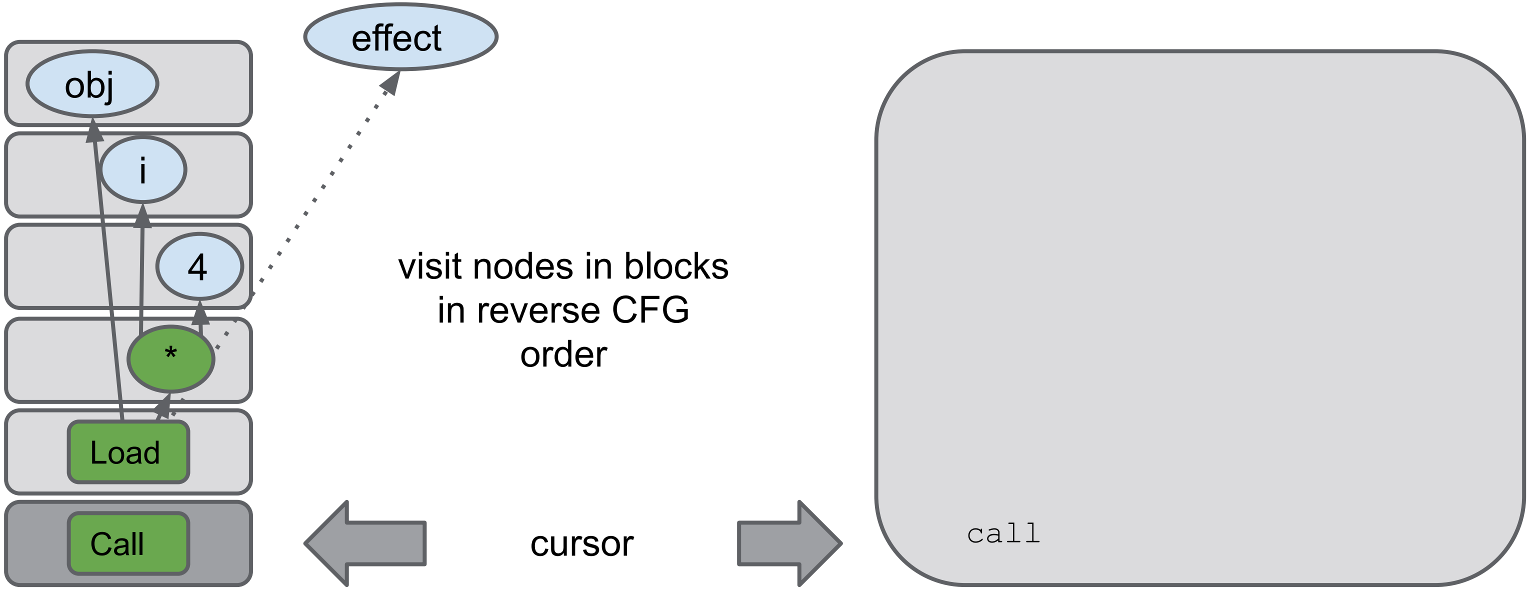
Instruction Selection (theory)



maximal munch

```
mov %r0, [%r1 + %r2 * 4]
```

Instruction Selection (practice)

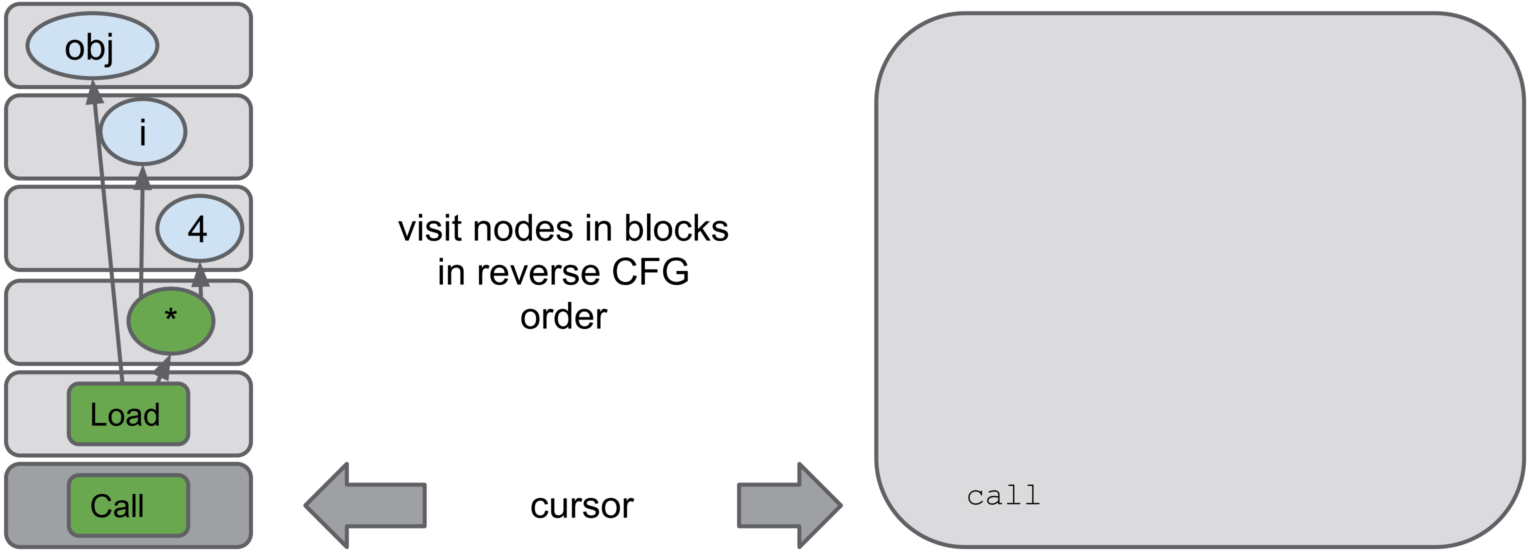


visit nodes in blocks
in reverse CFG
order

basic block

instruction sequence

Instruction Selection (practice)



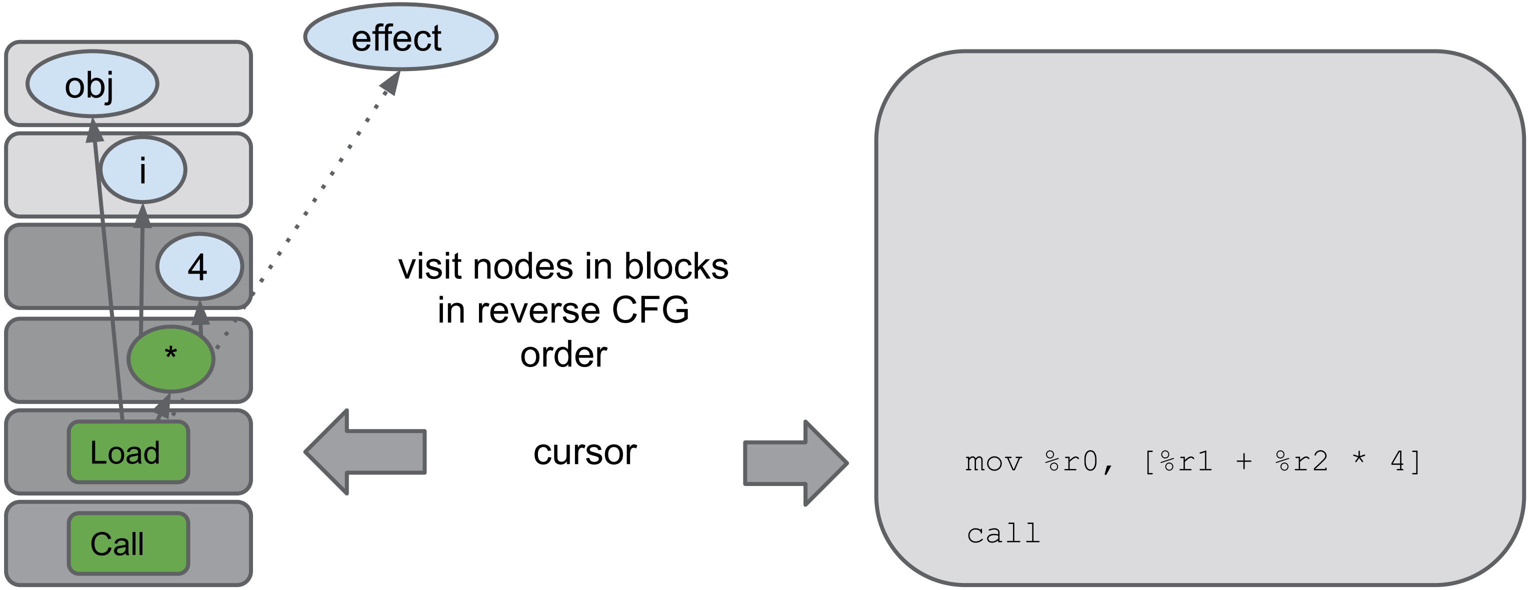
visit nodes in blocks in reverse CFG order

cursor

basic block

instruction sequence

Instruction Selection (practice)



visit nodes in blocks in reverse CFG order

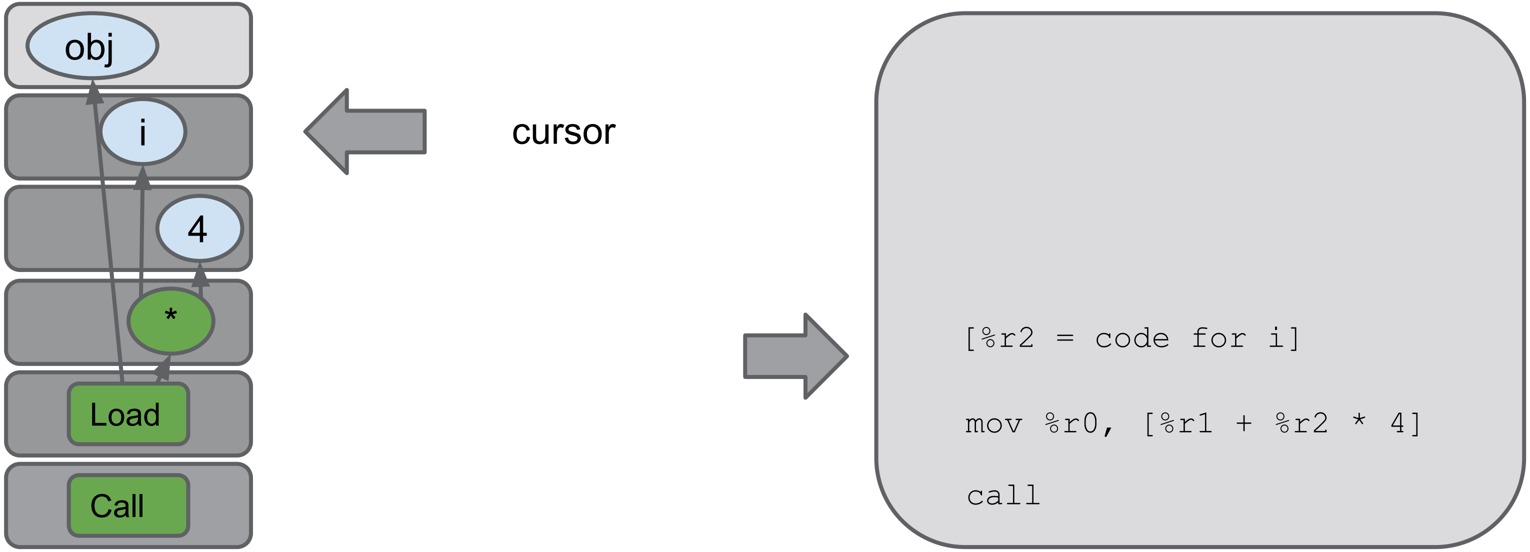
cursor

```
mov %r0, [%r1 + %r2 * 4]

call
```

basic block

instruction sequence

Instruction Selection (practice)

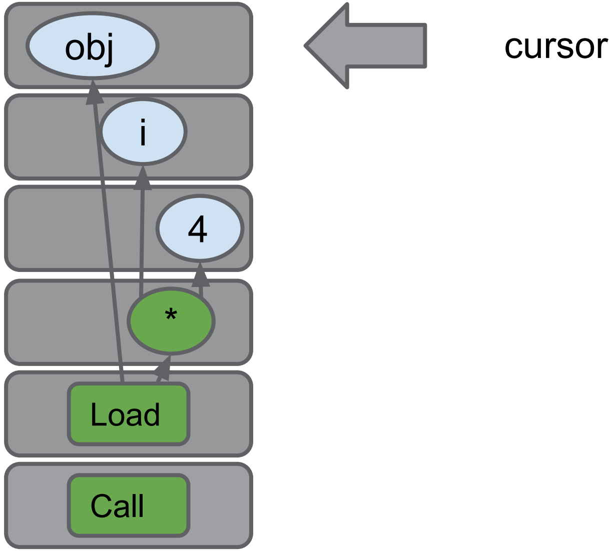


```
[%r2 = code for i]

mov %r0, [%r1 + %r2 * 4]

call
```

basic block

instruction sequence

Instruction Selection (practice)



basic block

```
[%r1 = code for obj]

[%r2 = code for i]

mov %r0, [%r1 + %r2 * 4]

call
```

instruction sequence

Register Allocation

TurboFan uses a linear scan allocator with live range splitting to assign registers and insert spill code.

SSA form can be deconstructed before or after register allocation with explicit moves.

```
[%r1 = code for obj]

[%r2 = code for i]

mov %r0, [%r1 + %r2 * 4]

call
```

instruction sequence

Register Allocation

TurboFan uses a linear scan allocator with live range splitting to assign registers and insert spill code.

SSA form can be deconstructed before or after register allocation with explicit moves.

The result of register allocation is to replace uses of virtual registers with real registers and to insert spill code between instructions.

```
mov [sp + 12], %eax

[%eax = code for obj]

[%ebx = code for i]

mov %ecx, [%eax + %ebx * 4]

call
```

instruction sequence

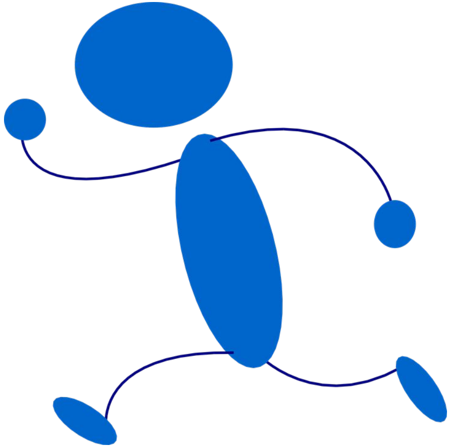
Testability

- Unit testing
  - Basic data structures, traversal algorithms, lowering, graph building, graph transformations, type relations, instruction selection, spill code insertion, register allocation, code generation, assemblers
- Integration testing
  - Run multiple optimization passes together, create specific graphs explicitly, try all combinations of arithmetic + generate and run code
- Performance tracking
  - Microbenchmarks, benchmark suites
- Fuzzing
  - Randomly mutate JavaScript source and feed it compiler

Status

- TurboFan is now in beta testing in Chrome 41
  - Initially enabled for asm.js
- 6 Fully supported platforms
  - ia32, x86-64, arm, arm64, mips, mips64
  - 2000-3000 lines per platform (vs 13000-16000 for CrankShaft)
- Improves Octane zlib benchmark 30-45%
- Most Emscripten benchmarks 10-25% faster
- Working on “general” JavaScript
  - Goal: all of ES6 within a couple of months

Thank You!



Ben L. Titzer	Michael Starzinger	Daniel Clifford	Benedikt Meurer	Dan Carney
Andreas Rossberg	Jaroslav Sevcik	Sven Panne	Sigurd Schneider	Georg Neis