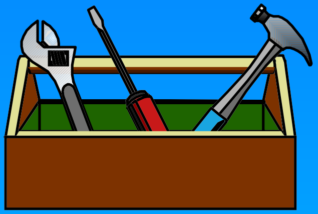
HotSpot Compiler - The Core IR

Cliff Click



Agenda

- HotSpot VM - Optimizer Context
- The Core IR - Small is Beautiful
- Data Nodes
- Control Nodes
- More Engineering
- Building SSA Fast
- Some Optimization

HotSpot VM

- Interpreter
- Profiling
  - Targeted optimization
- Runtime support
  - GC, exceptions, threads
  - Type analysis
  - Deoptimization
  - Code patching

Optimizer Context

- Fast, fast, ~~fast~~
- C++ quality code
- Inputs:
  - Bytecodes, type analysis, profile
  - Inline decision tree
- Outputs:
  - Machine code, GC annotations
  - exception walkback tables, safepoints

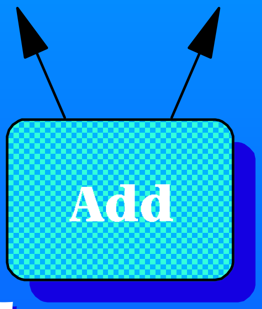
The Core IR:
Small is Beautiful

- Small is Fast
- Small is Simple
- Small can get the job done
  - ► *...but little room for design errors*
- Small requires Engineering
  - ► some Small from Recycle & Reuse
  - ► *...but to get really Small requires*

Reengineering!

Graph & SSA IR

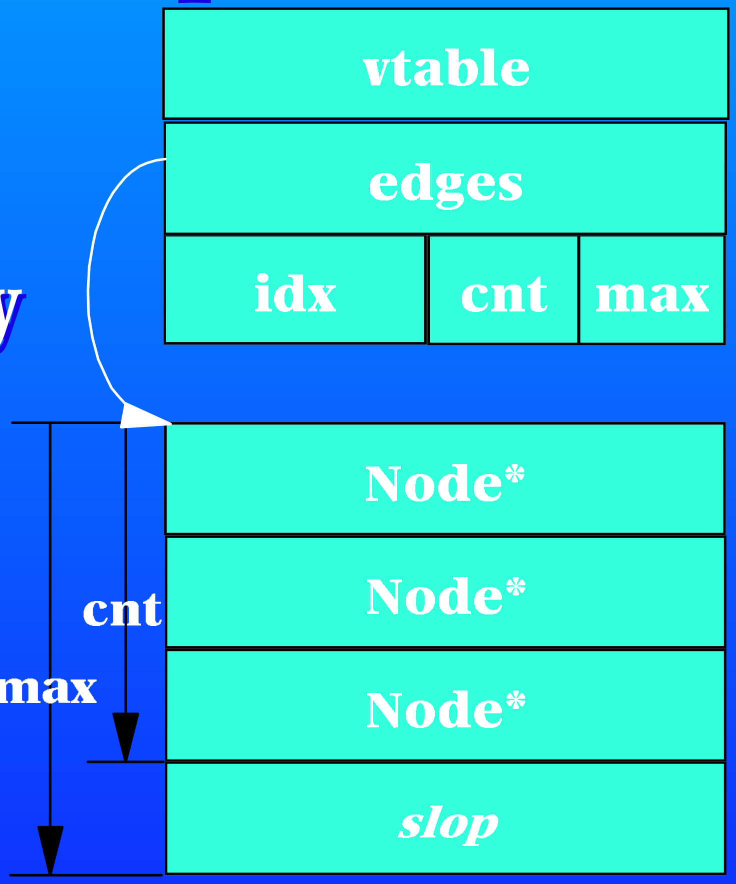
- Nodes
  - Primitive operations
  - Control ops (aka Basic Blocks)
  - Phi, conditionals, memory, calls, ...



- Edges
  - Data flow/data dependencies
  - Control flow/control dependencies
- Explicit Use-Def chains!

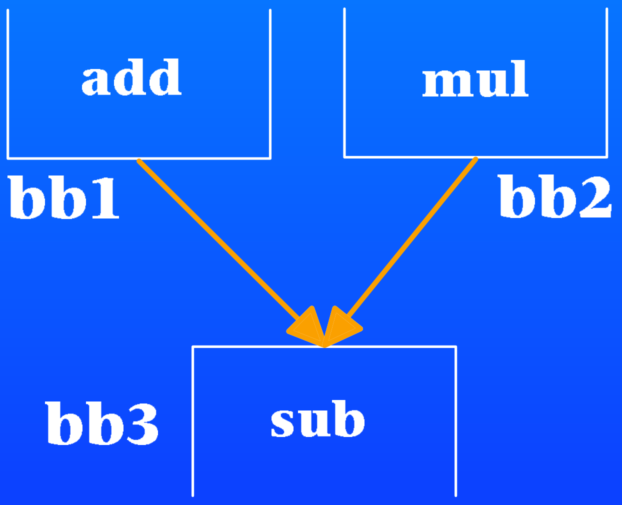
class Node

- 12 bytes +4 per edge +slop
- vtable is opcode
  - (e.g., add, phi, if)
- extensible edge array
- dense integer index
  - into side arrays
- use-def edges
  - *...but not def-use!*

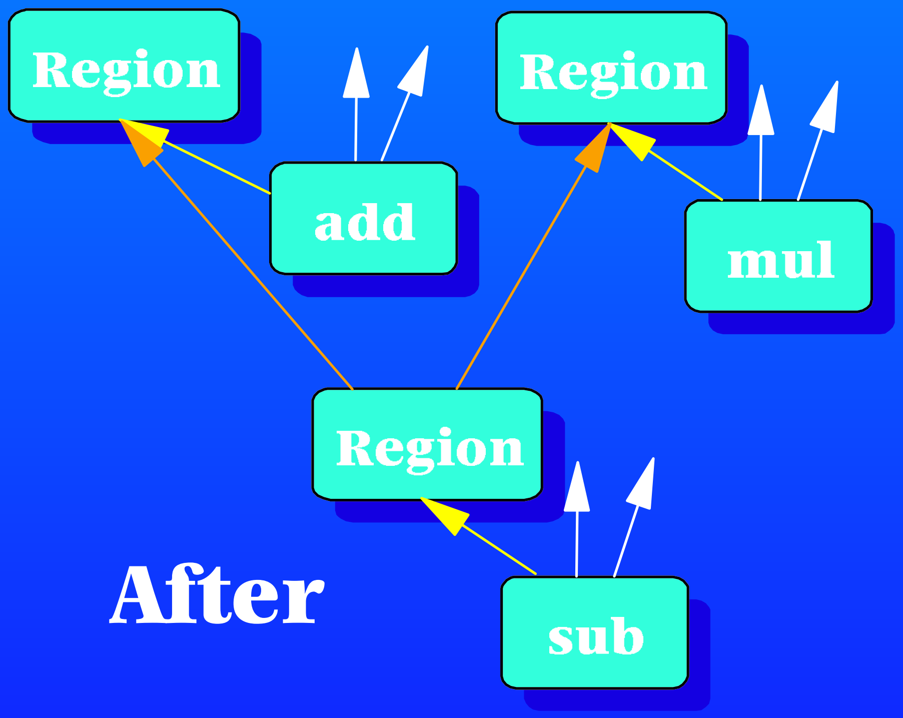


class RegionNode - Basic Blocks

- No different than other Nodes!
- Data Nodes point to Region

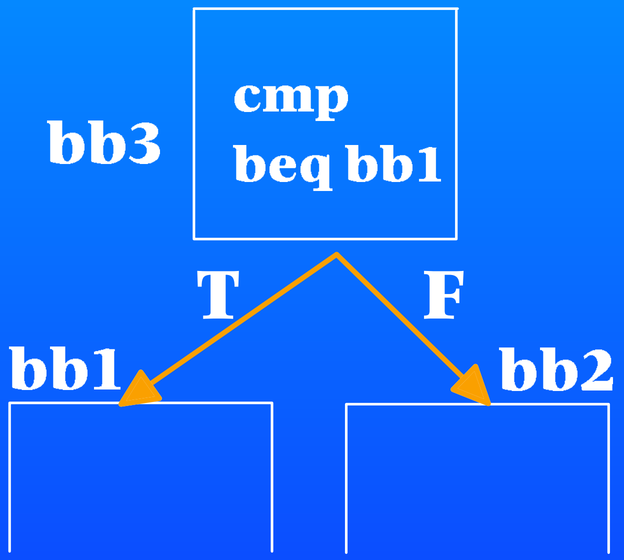


Before

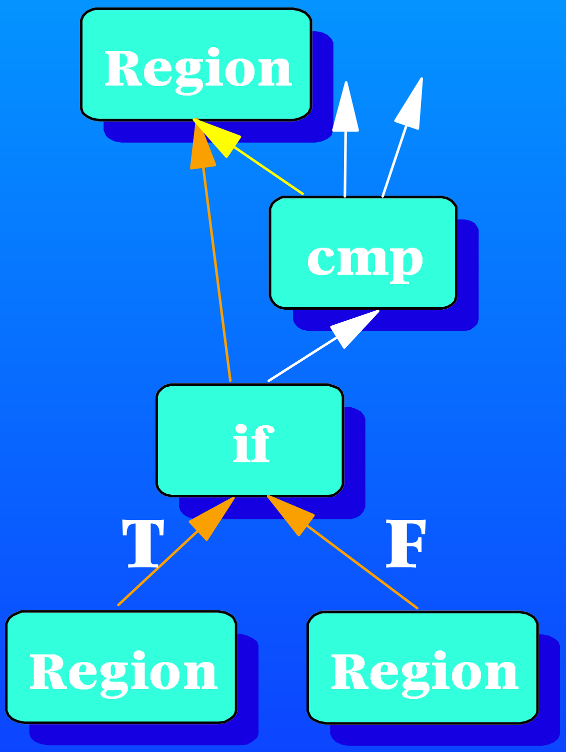


After

class IfNode



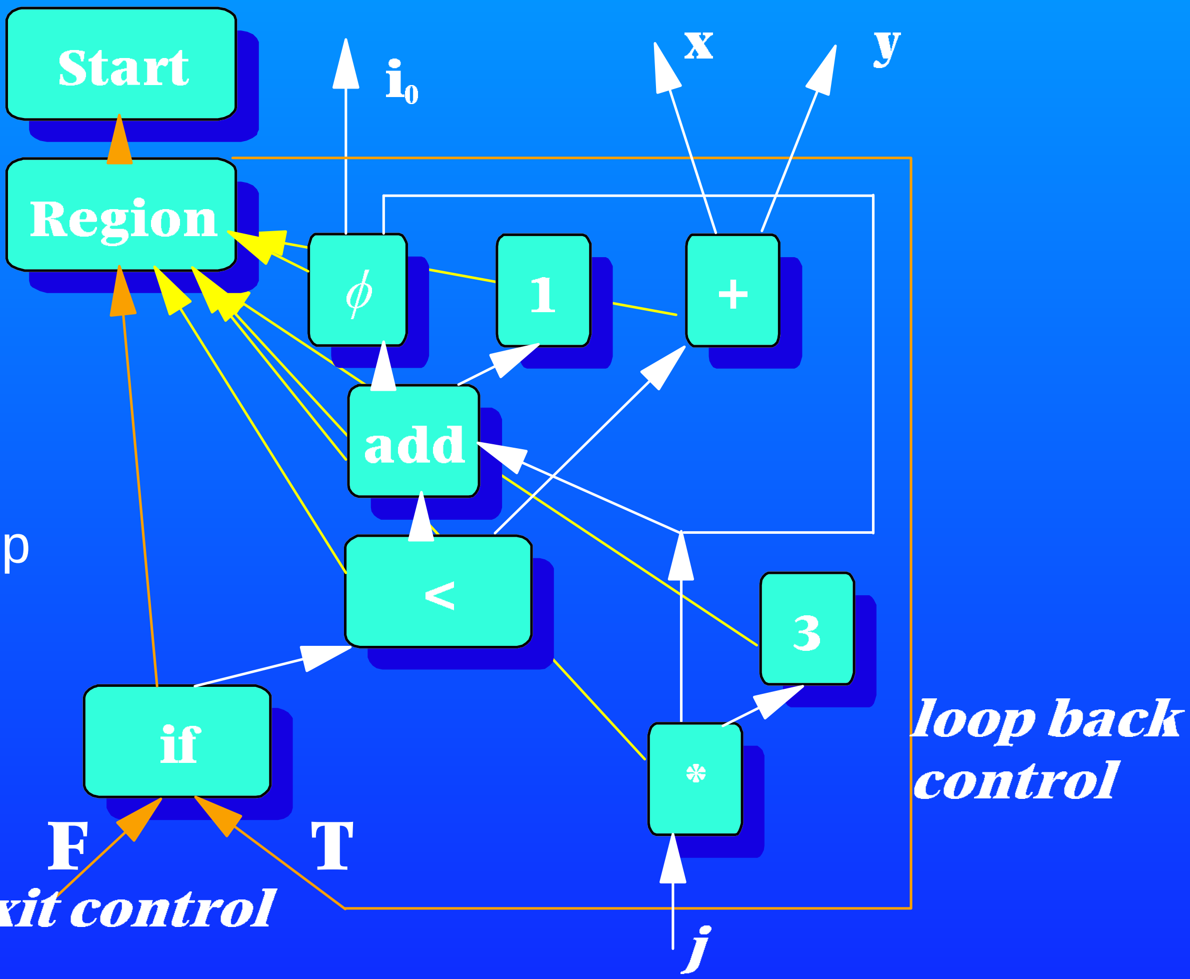
Before



After

A simple Loop

```
i = i0;
loop:
  i++;
  j = i*3;
  if i < x+y
    goto loop
...j...
```

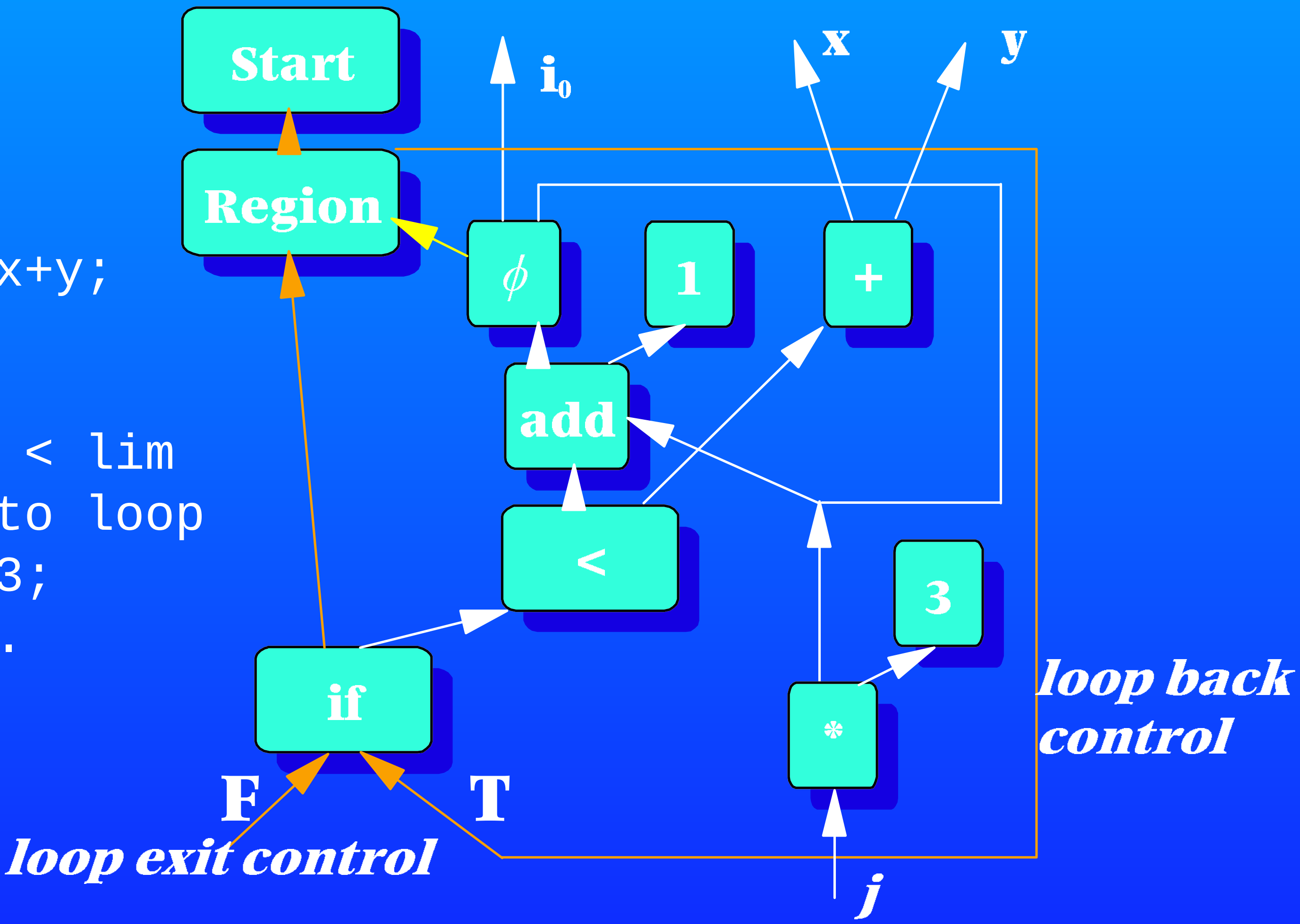


Sea of Nodes

- Zap the control input
- Data Node does not belong to any basic block
- Enables Global Value Numbering
- Filled in by Global Code Motion

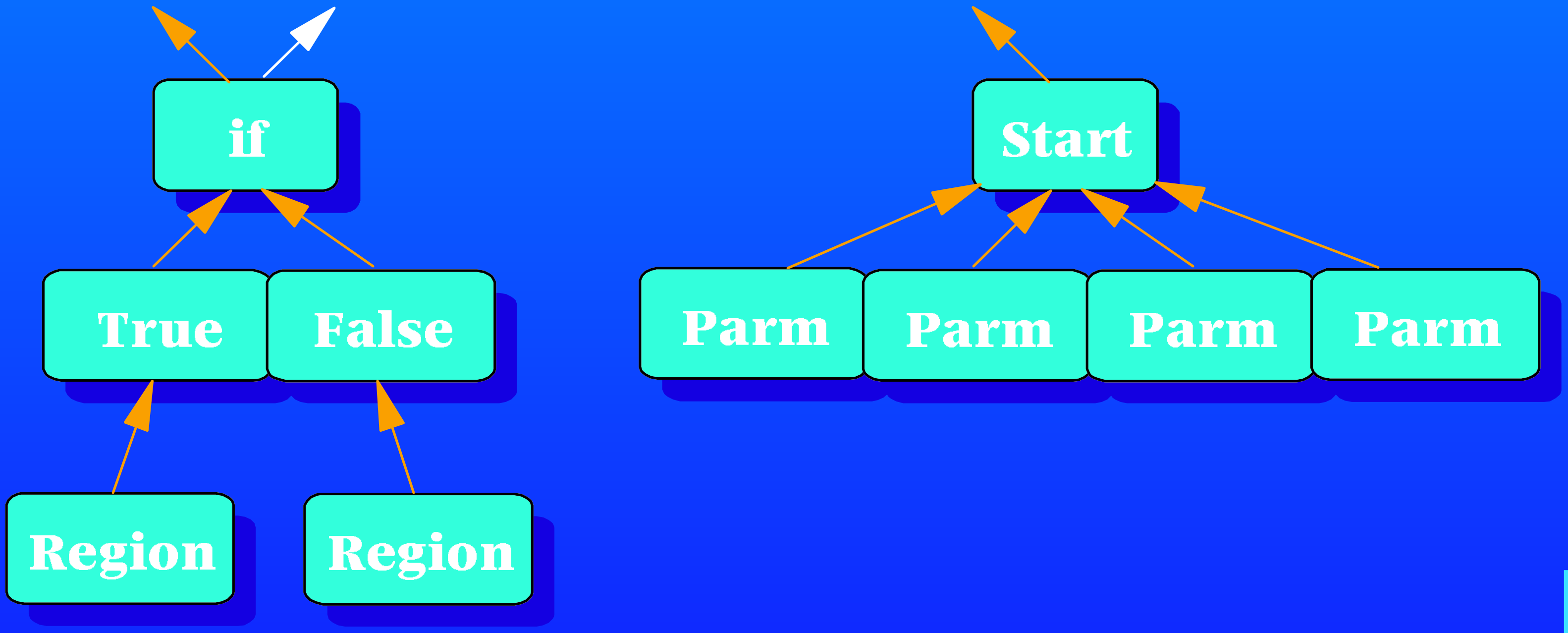
A simple Loop

```
i = i₀;
lim = x+y;
loop:
  i++;
  if i < lim
    goto loop
j = i+3;
...j...
```

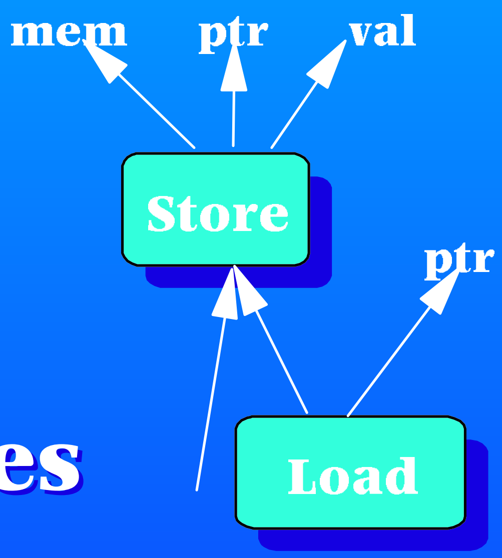


MultiNode & ProjNode

- Labels an Edge!
- MultiNode produces a tuple
- ProjNode slices out one field



Memory

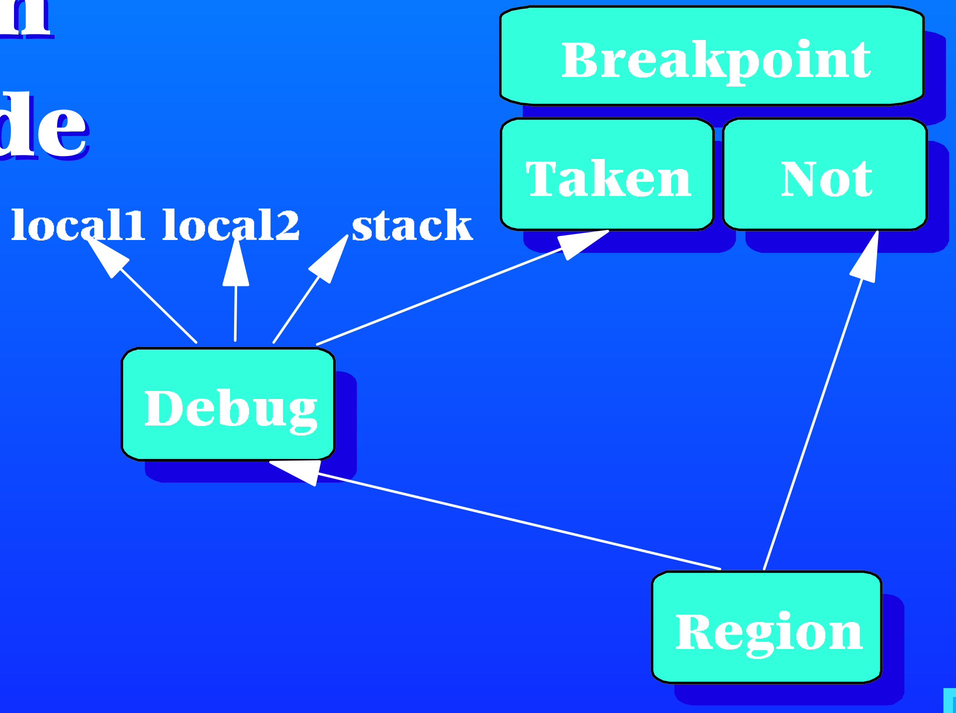


- Just another Value
- Input to LoadNode
- Result of StoreNode
- Break into disjoint pieces

- I/O is treated same as memory
  - Read also outputs new I/O Value

Deoptimize, Debug

- DebugNode captures JVM state
- Optimizer honors dependencies
- Low freq branch
- No machine code
- Safepoint

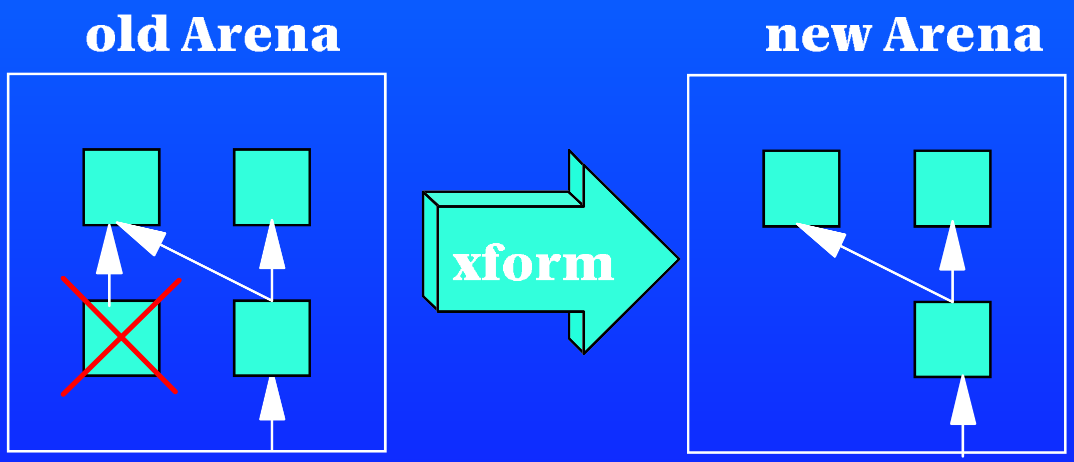


Phases

- Use a side-array
- Index by Node::idx
- Analysis lifetime is controlled
- Faster to re-analyze than to keep analysis correct after transform
  - conservative approximation
  - subtle bugs
- E.g., build def-use in 230 cycles/Node

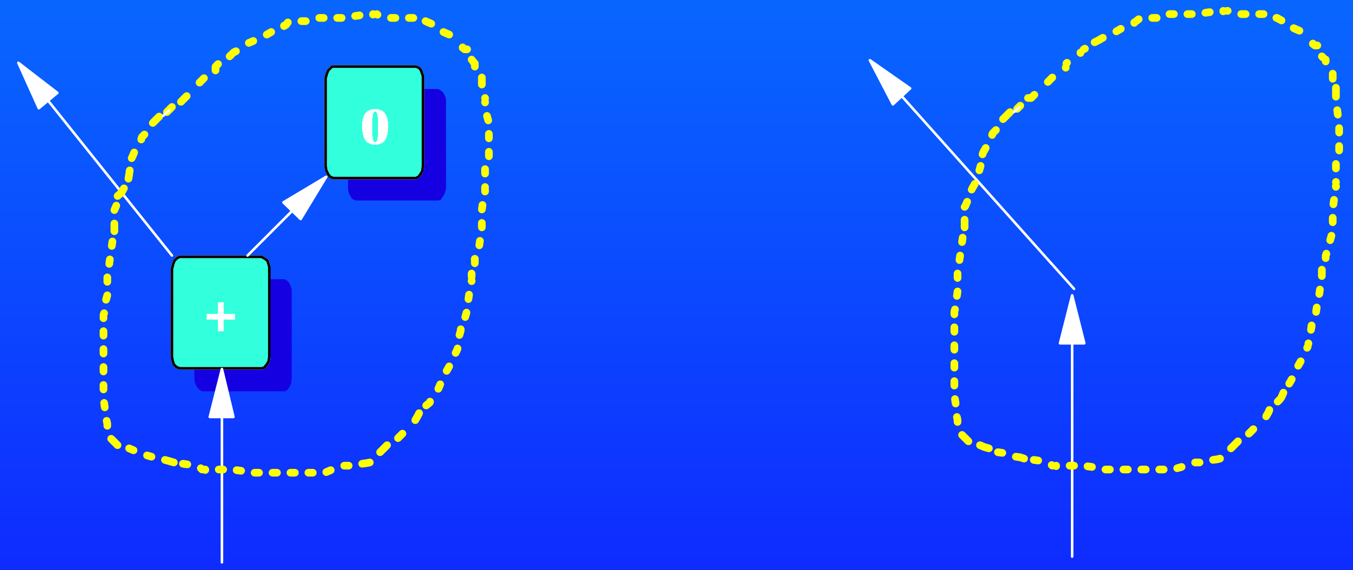
Allocation

- Arena-based
  - Overload new, delete is a no-op
- Copy live Nodes to new arena
  - Programmer specified GC
- Not DCE - already happens "for free"



Peephole Optimization

- Graph rewrite rules
- Virtual functions
  - ▸class AddINode::Identity () {
    return (in[2]==zero) ? in[1] : this; }



Global Value Numbering

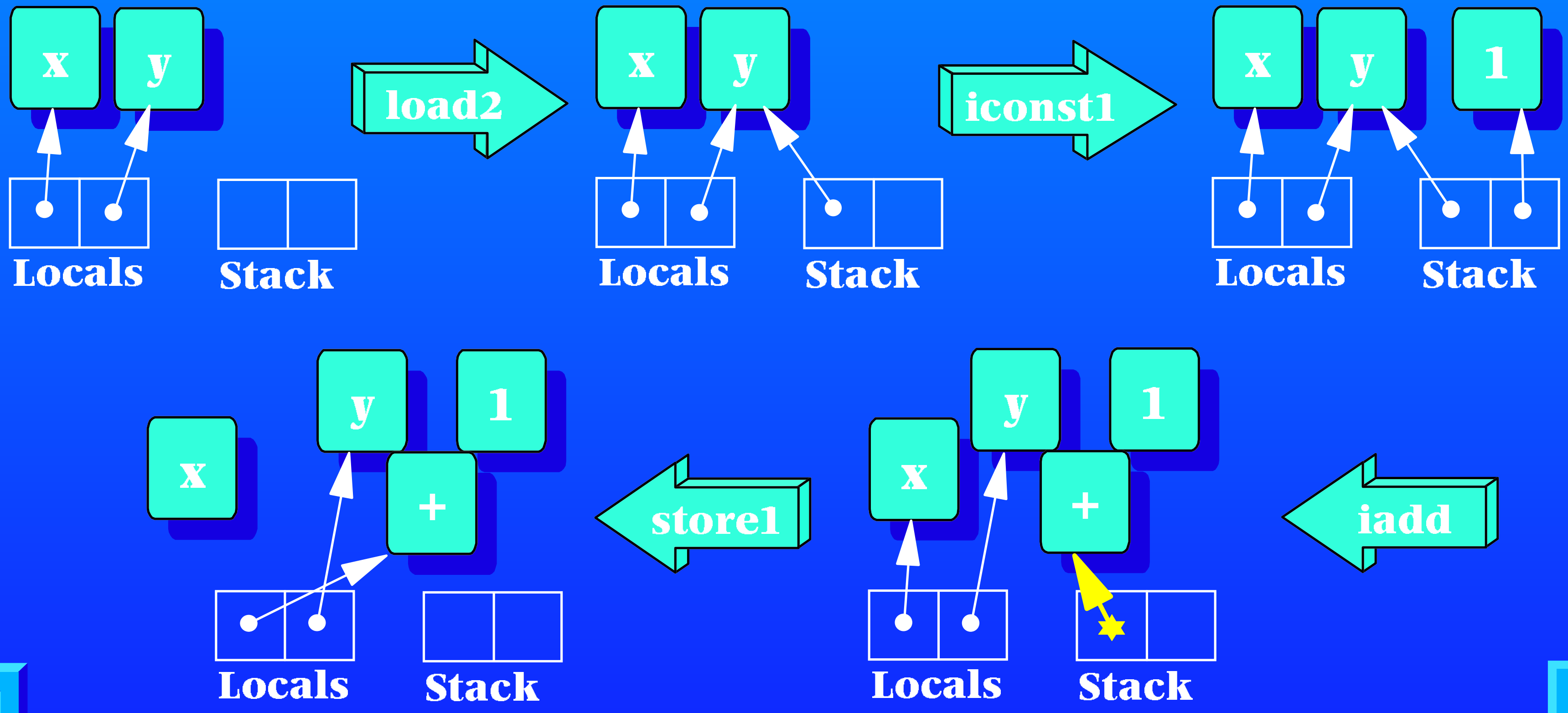
- V-call to constant fold
  - Dead control folds also!
- V-call to find identities
- V-call to "idealize"
- V-call to hash/compare
  - Combine Nodes from different blocks
  - Global Code Motion will fixup later

Parsing bytecodes

- 1 pass to find merge points
  - cache full type info
- 2nd pass builds it all
  - Build CFG (aka Region/If Node)
  - Walk in Reverse Post Order
    - No useless Phis except at loops
  - Build data Nodes, PhiNodes
  - Peephole optimize as you go

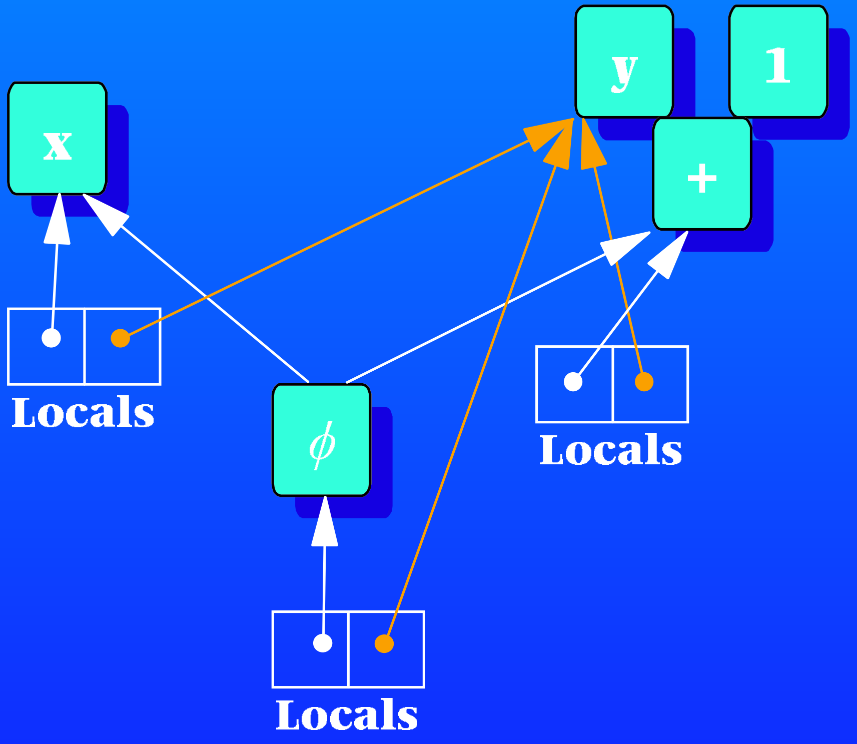
Straightline Code

- Parser maps JVM state to Nodes
- Example: x = y+1



Control Flow & SSA

- Asymptotically slower
- Really fast in practice
- At merge points:
  - Compare maps, insert Phis



Summary

- Small!
  - E.g., One big method has 4890 BCs, 5000 Nodes, around 120K bytes
  - Small allows Fast
- High Quality Code
  - GVN, GCM
  - SSA form
  - plus BURS instruction selection, Briggs-Chaitin allocator, etc...