C2
The JIT In HotSpot

Cliff Click

Who Am I?

Cliff Click

Leader, Founder
Cratus, Rocket School, Neurensic,
H2O.ai, Azul, Sun
cliffc@acm.org

PhD Computer Science

1995 Rice University

HotSpot JVM Server Compiler

"showed the world JITing is possible"

45 yrs coding

40 yrs building compilers

35 yrs distributed computation

30 yrs OS, device drivers, HPC, HotSpot

15 yrs Low-latency GC, custom java hardware,
        NonBlockingHashMap

10 yrs ML tool building, ML applications

20+ patents, dozens of papers

100s of public talks

It was 1997...

- Plenty of static (AoT) compilers
  - Very slow code gen, good code quality
- Plenty of interpreted languages
- Some blended languages (e.g. Forth)
  - But only with template-style code gen
- I'm doing a fast & good compiler
- Heavy emphasis on speed… which is tied to memory footprint, which means a small IR
- But not a simple IR… these things can be subtle

… and now it's 2019

- 22 years later
  - I left Sun when it was still Sun, in 2002
  - My knowledge is dated
  - And yet: the Bones of C2 remain
  - And a clear lineage from my Rice U compiler days
- So expect some dated stuff, and maybe some just plain wrong stuff
- But I think mostly: It Hasn't Changed (much)
  - Still in dir "opto" as it was when I brought it from Rice

This Is A Compiler Talk

- IR – Intermediate Representation

- SSA – Static Single Assignment

  – Graph Rewrite Rules, RPO – Reverse Post Order

- Optimization passes

  – DCE – Dead Code Elimination, GCP – Global Constant Propagation, GVN – Global Value Numbering, CSE, RCE, Inlining, Unrolling, …

- Portable Code Generation (not just X86!)

  – Machine Code(s) for lots of chips

- Graph Coloring Register Allocation

This Is A *Hardcore* Compiler Talk

- IR – Intermediate Representation
- SSA – Static Single Assignment
  - Graph Rewrite Rules, RPO – Reverse Post Order
- Optimization passes
  - DCE – Dead Code Elimination, GCP – Global Constant Propagation, GVN – Global Value Numbering, CSE, RCE, Inlining, Unrolling, …
- Portable Code Generation (not just X86!)
  - Machine Code(s) for lots of chips
- Graph Coloring Register Allocation

This Is A Hardcore Compiler Talk

Engineering: A Compiler:
Keith Cooper, Linda Torczon

- No apologies!   :-)

- I'm going to use a lot of jargon
    - Which is fairly specific to compilers
    - ...some of which is explained in later slides
    - And all of it can be read about elsewhere

- This talk is more about the C2 design philosophy

- And is targeted for those hacking C2
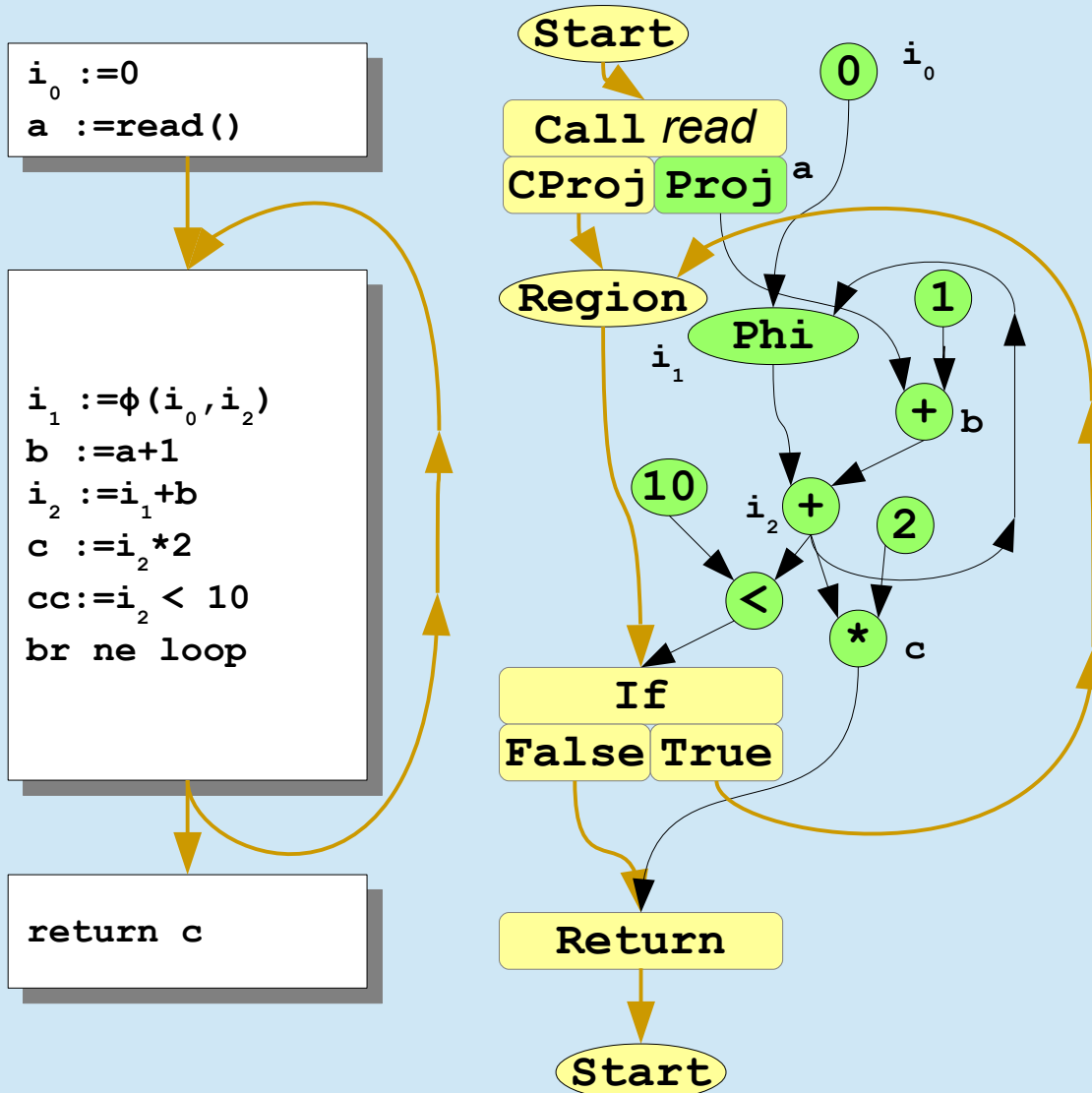   - Or trying to understand C2 in more detail
    - Or in compilers like C2

The Java HotSpot Server Compiler:
Click, Paleczny, Vick

Sea of Nodes

- **SSA Always**

    – Even during final code-schedule & reg-alloc

- Nodes and (implicit) Edges, i.e. a Graph

    – **Small == Fast** – Very limited data in nodes

    – Edges are *direct pointers* for speed, so unlabeled

    – Data and Control use the **same** Graph

    – Data is decoupled from CFG and floats around

- Strongly Typed – and more precise than Java

    – Type System is large, complex, subtle & very fast

- Loosely coupled to the JVM Runtime

Sea of Nodes

```
i_0 :=0
a :=read()
```

```
i_1 :=φ(i_0,i_2)
b :=a+1
i_2 :=i_1+b
c :=i_2*2
cc:=i_2 < 10
br ne loop
```

```
return c
```

Start

Call *read*

CProj Proj ^{a}

Region

0 i_0

Phi i_1

1

$+$ b

10 i_2 $+$ 2

$<$ $*$ c

If

False True

Return

Start

Program Semantics
   as a single unified Graph

As opposed to a two-layer:
   CFG + Basic Blocks

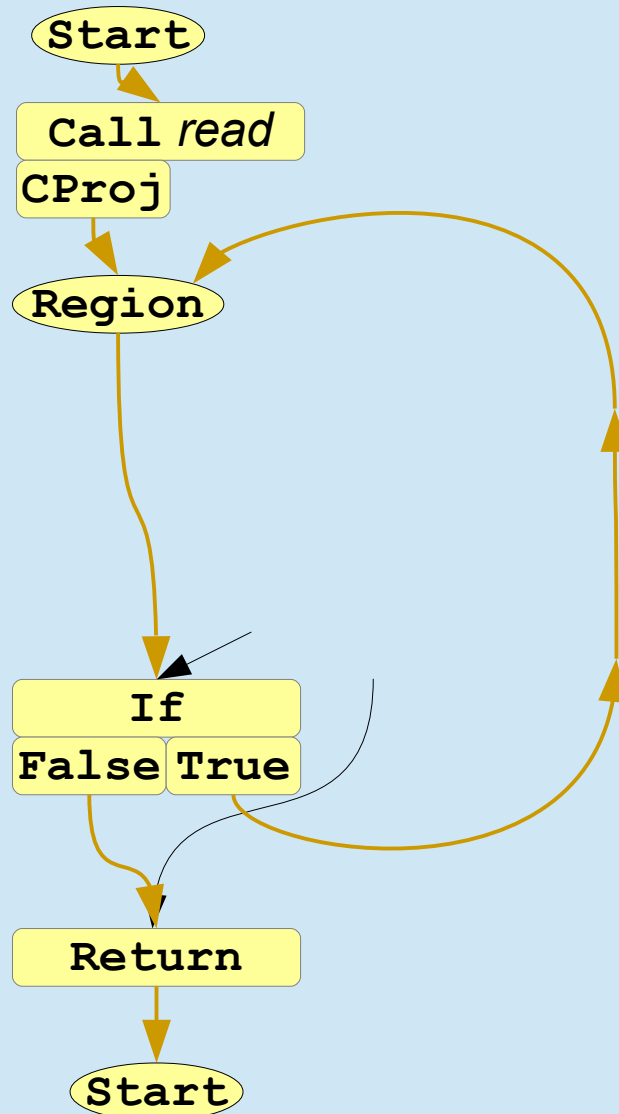*A Simple Graph-Based IR:Click, Paleczny*

From Quads to Graphs: An IR's Journey: Click

Sea of Nodes

Control Flow Graph

Embedded CFG
"Control as a Value"
No explicit basic blocks

Start – (and end)
Region – merge
If   – split
Call   – serialized

Start is also the End merely for convenience.

Graph can be walked in either direction.
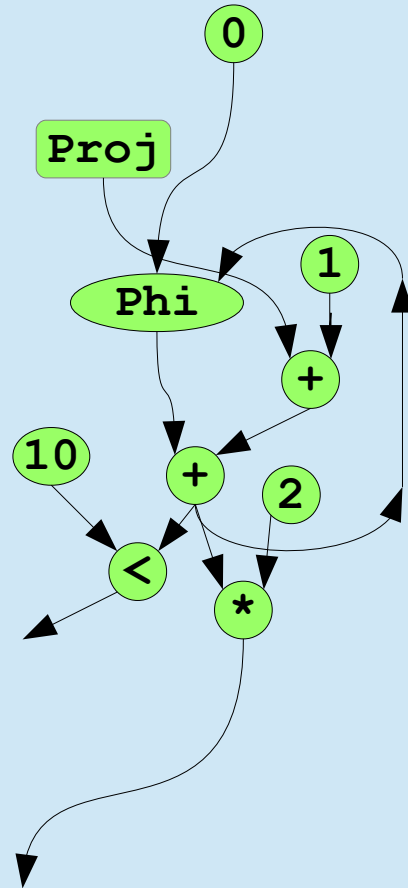
Start

Call *read*
CProj

Region

If
False True

Return

Start

Why no BB's?

Because most data ops just don't care. Means most transforms can just ignore BB boundaries.

Because all program mods (both control and data) become Graph Rewrite Rules for the same Rules engine, in slides coming.

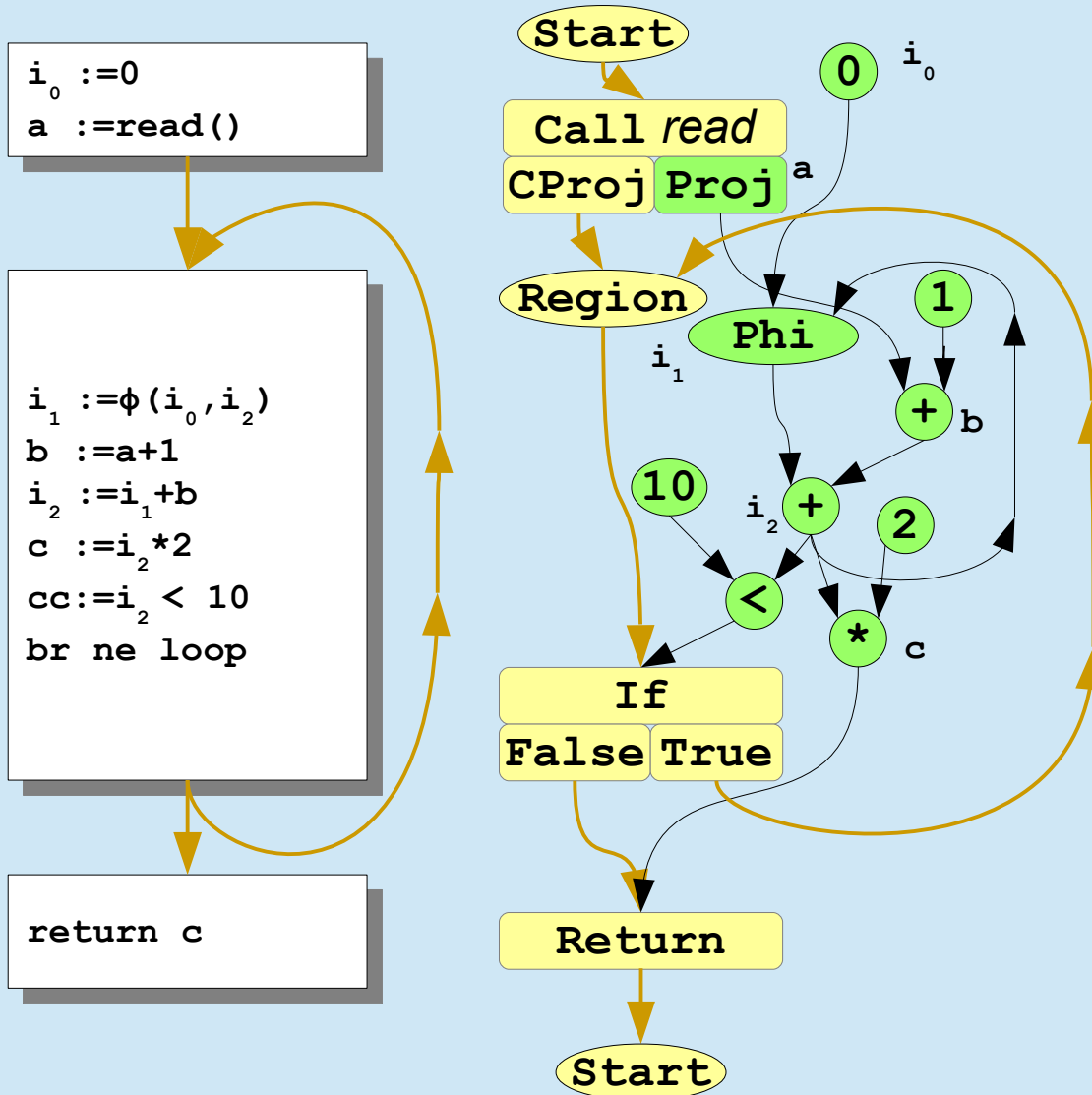Region implies BB start and If implies BB end, but not vice-versa.

Sea of Nodes

0

Proj

Phi

1

10

+

2

+

<

*

Static Single Assignment

Embedded Data Flow
Basic math, constants
Phi nodes
No variable names
No basic blocks

Sea of Nodes


```
i₀ :=0
a :=read()
```

```
i₁ :=φ(i₀,i₂)
b :=a+1
i₂ :=i₁+b
c :=i₂*2
cc:=i₂ < 10
br ne loop
```

```
return c
```

Control and Data Blended

Call –
    Produces control
    Produces data
If -
    Consumes control & data
    Produces 2 controls
Return -
    Consumes control & data
    Produces 1 control

Sea of Nodes

```
i₀ :=0
a :=read()
```

```
i₁ :=φ(i₀,i₂)
b :=a+1
i₂ :=i₁+b
c :=i₂*2
cc:=i₂ < 10
br ne loop
```

```
return c
```


Control and Data Blended

Call –
    Produces control
    Produces data

If –
    Consumes control & data
    Produces 2 controls

Return –
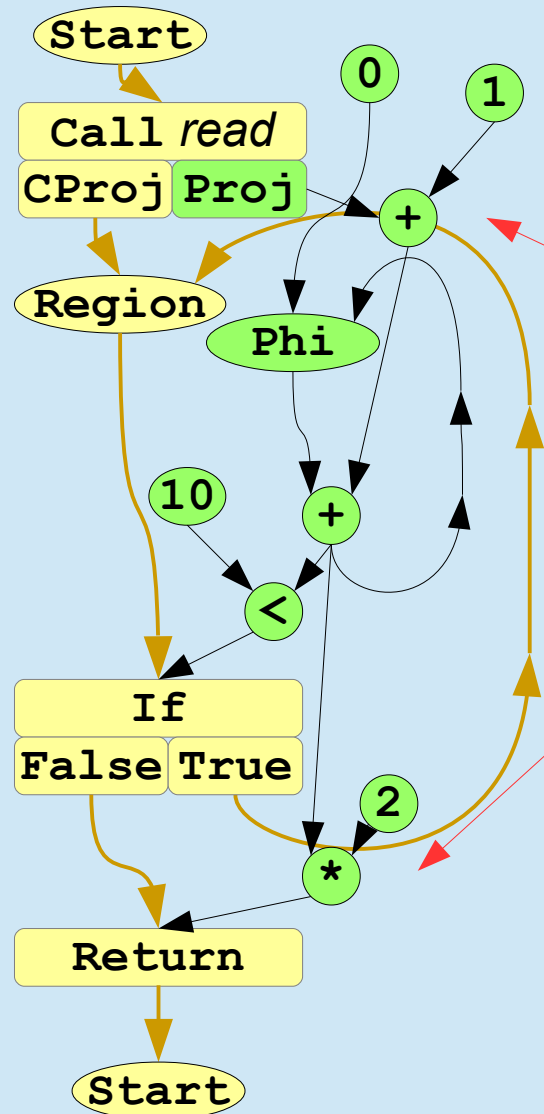   Consumes control & data
    Produces 1 control

Sea of Nodes

```
i₀ :=0
a :=read()
```

```
i₁ :=φ(i₀,i₂)
b :=a+1
i₂ :=i₁+b
c :=i₂*2
cc:=i₂ < 10
br ne loop
```

```
return c
```

Start

Call *read*

CProj Proj

Region

Phi

i_0

0

i_0

i_1

1

$+$ b

10

i_2 $+$ 2

$<$ $*$

If

False True

Return

Start

Control and Data Blended

Call –
    Produces control
    Produces data
If –
    Consumes control & data
    Produces 2 controls
Return –
    Consumes control & data
    Produces 1 control

Sea of Nodes

```
i₀ :=0
a :=read()
```

```
i₁ :=φ(i₀,i₂)
b :=a+1
i₂ :=i₁+b
c :=i₂*2
cc:=i₂ < 10
br ne loop
```

```
return c
```

Start

Call *read*

CProj **Proj** a

Region

i_0

0

Phi i_1

1

+ b

10

i_2 **+**

2

<

* c

If

False **True**

Return

Start

Control and Data Blended

Call –
  Produces control
  Produces data

If -
  Consumes control & data
  Produces 2 controls
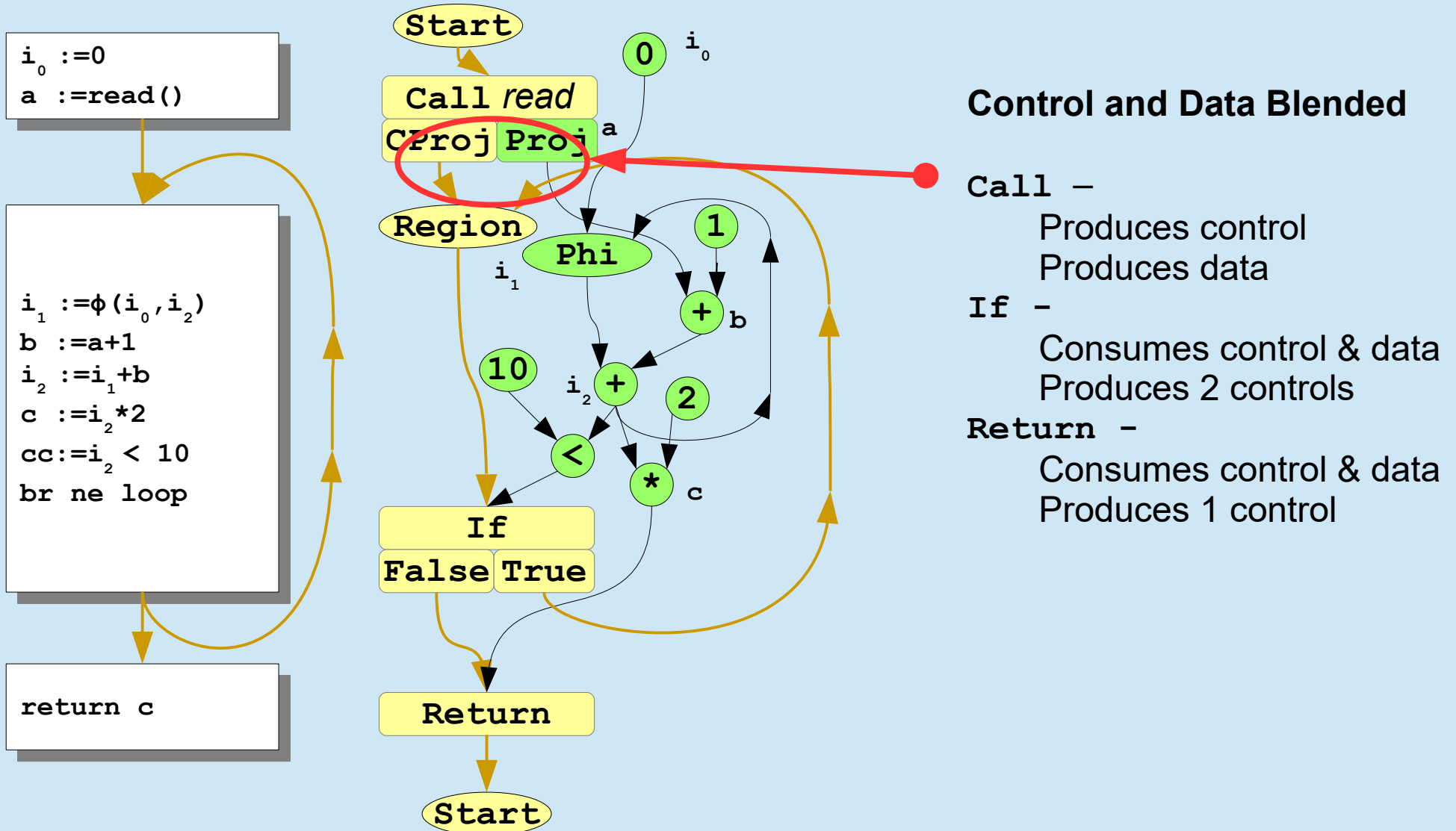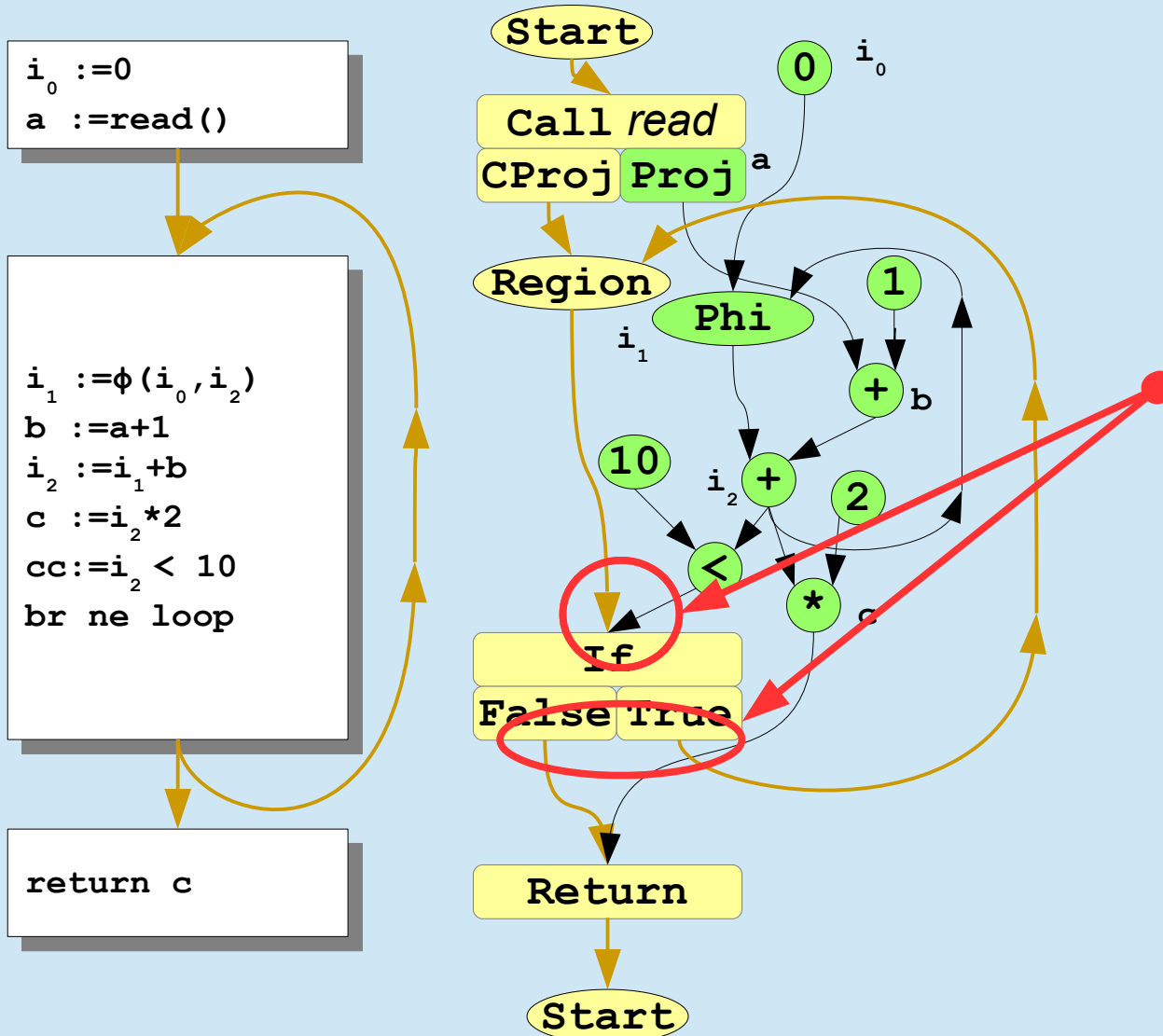
Return -
  Consumes control & data
  Produces 1 control
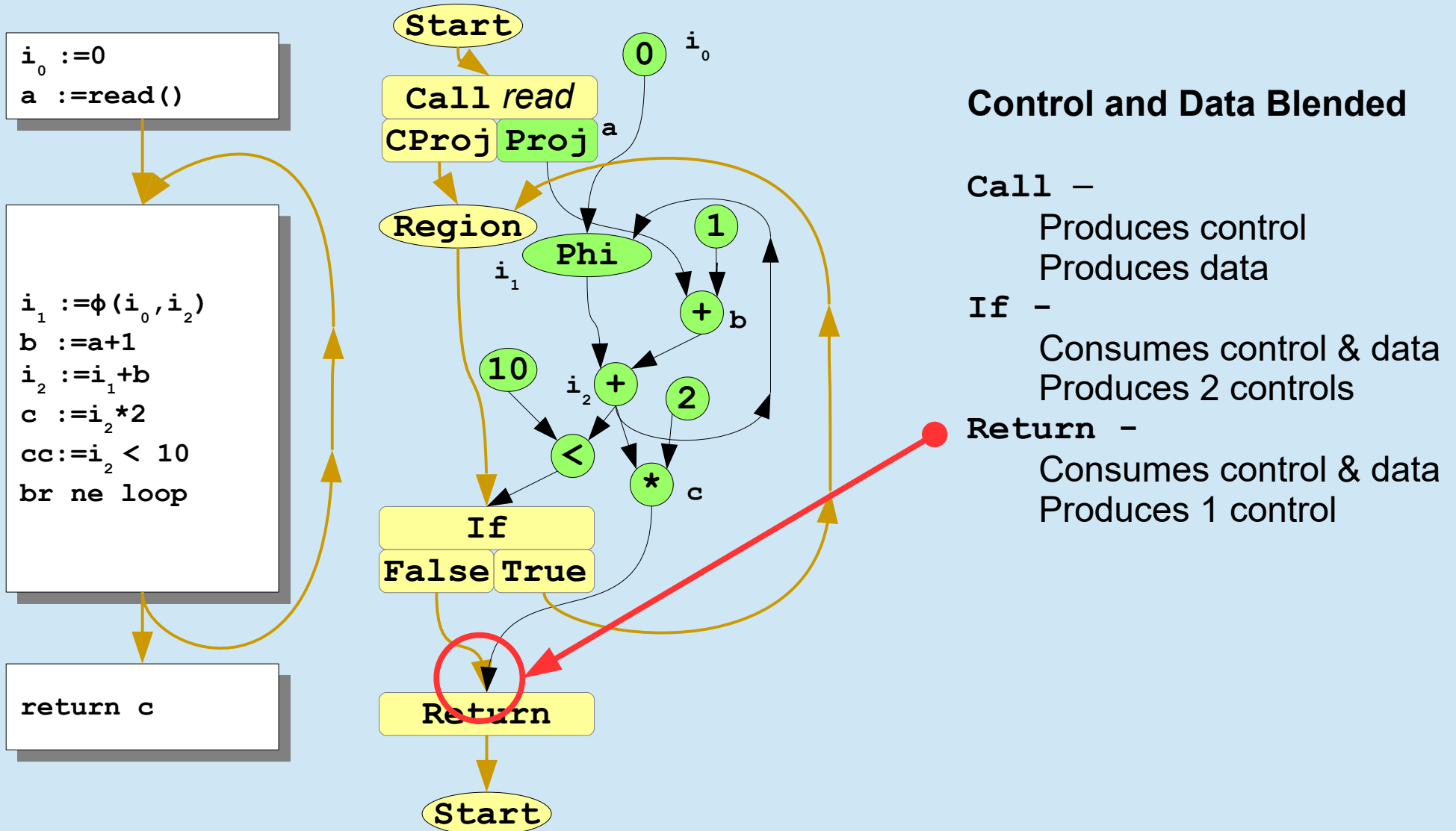
Sea of Nodes

Only Edges Order

Nodes have no real "place".
They float about.

Semantics comes from Edges.
Not names

Sea of Nodes

Only Edges Order

Nodes have no real "place".
They float about.

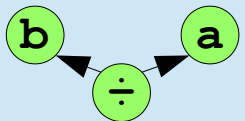Semantics comes from Edges
Not names
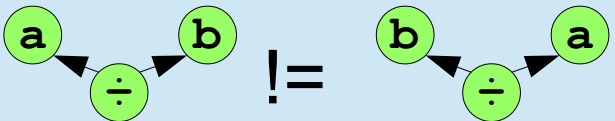
Sea of Nodes

Major Passes (abbreviated)

- Build **Sea of Nodes** from bytecodes
  - Includes inlining and peephole opts
- Iter – Repeat peephole opts until done
  - Peephole: a monotonic Graph Rewrite Rule
  - Includes DCE, c-prop, CSE, ld/st opts
    - … and every kind of small opt you can think of
  - Guaranteed linear (IF all rewrites are "monotonic")
- GCP, Loop Opts (RCE, Unroll)
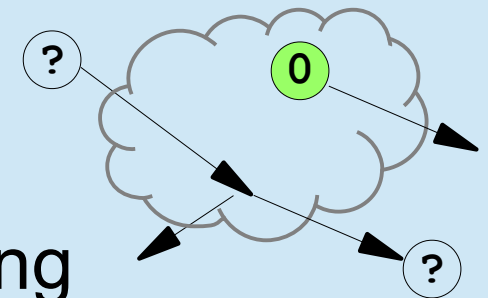- GCM (scheduling, anti-deps)… now "fragile"
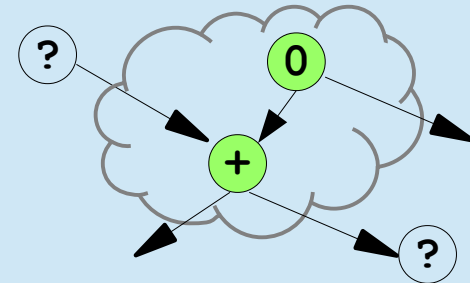- Code-gen, Reg-Alloc, atomic install in JVM

Nodes

- Opcode/semantic is C++ v-table
  - ~35 V-calls: name, peepholes, c-prop, debug prints
  - Speed & size concerns, edge arrays, limited RTTI
    - No Other Data in Nodes! **Small == Fast**
- Explicit Use→Def edges
  - Raw C pointers from Use node to Def node
  - Order matters: !=
  - NULL allowed (and common in `_in[0]`)
- Explicit Def→Use edges added later
  - Unordered: just a list; no nulls

Peepholes

- Graph Rewrite Rules
    - Inspect a *local* graph area
    - Replace with semantically equivalent but "better"
        - No change outside of area
        - Nice neighbors are **unaware of change**
    - *Monotonic* is required to prevent looping
- Most areas are "rooted" at a single Node
    - And that Node's `Ideal` v-call inspects and changes
- Literally hundreds of such rules
    - Most trivially obvious

What is *local*?
Inside some closed area.

Iter: Repeat Peepholes

- Major Optimization "Pass"
  - Pull node from worklist
  - Peephole if possible
  - Also check constant propagation, GVN, DCE
    - These apply uniformly to all nodes
  - If changed, Push neighbors on worklist
  - Lather, Rinse, Repeat…
- Gets all the "junk" or "easy" stuff!
- Called all over the compiler
- **Fast**: Sum of **all** Iter passes *linear in program*

Edges

- Edges are direct pointers: **Speed!!!**
  - Since plain C ptrs… cannot carry labels
- Never a need to label Use→Def edges
  - Already ordered and the order carries the meaning
- Some Nodes produce multiple outputs
  - Need to label which result is carried on Edge
- "Projections" - a slice of a Tuple
  - **ProjNode** follows a **MultiNode**
  - And so, effectively, labels an Edge
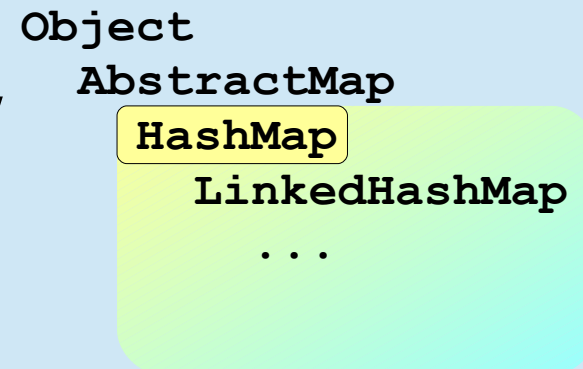 - But only 5% where needed, and not the 95%

`If`
`False` `True`

`Call` *read*
`CProj` `Proj`

Types

- Types define what a compiler can reason about
- A Type is a Set of values
    - All integers, all floats, 3.1415, null, class String, etc
- All Nodes have a Type!
    - Including "Control" for Region Nodes
    - Value() v-call makes Type, using Use→Def Types
- Types obviously used for c-prop
    - But also CHA & Inlining, dead if test, switches, RCE, upcast, and many more places
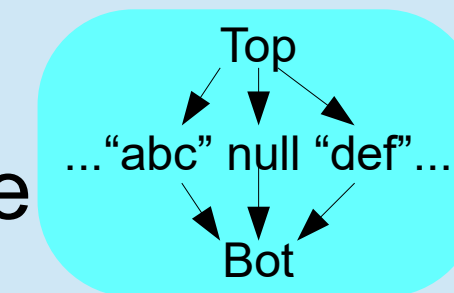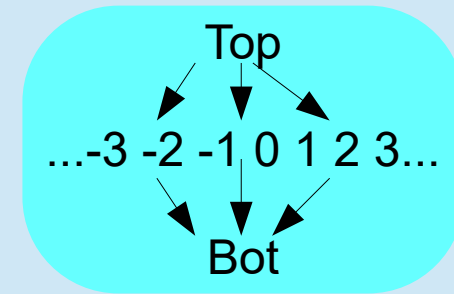- Precision especially useful in CHA & Inlining

Types

- All int sizes, arbitrary ranges (e.g. 0-10)

- Float 32, 64; float constants

- Tuples, Arrays, Instances – Memory state
  - Equivalence-class aliasing, sets of aliases

- Pointers: raw, oop {instance, array, klass}

- Klasses both exact and inexact

  ```
  Object
    AbstractMap
      HashMap
        LinkedHashMap
          ...
  ```

  - e.g. HashMap vs HashMap-or-below

Combining Analysis, Combining Optimizations: Click, Cooper

Types

- Types define a *Lattice*

  - Defined by Meet and Dual (not join)

    - Boolean Algebra: Complete, Complemented, Distributive, Bounded (Ranked)

- "Bot" – Not a constant; value may be unknown; compiler honors how they get made

- "Top" – All constants, all at once. Compiler **_chooses_** which one to use, as convenient.

- Now, extend this notion to instances, classes, and just about everything else

https://en.wikipedia.org/wiki/Lattice_(order)

Top
...-3 -2 -1 0 1 2 3...
Bot

Top
..."abc" null "def"...
Bot

Types

- Type descriptions can get big
- Types are immutable, hash-cons (interned)
  - Compare with "==" and not "equals()"
    - Asymptotically slower to use "equals()"
    - Except for cyclic types, then just a crash
  - Most created types hit in hash-cons table
    - So recycle type memory for speed
- Defined by meet & dual
  - join(t) { return dual().meet(t.dual()).dual(); }
  - isa(t) { return meet(t)==t; }

https://en.wikipedia.org/wiki/Hash_consing

Pessimistic vs Optimistic

- **Iter** & Peepholes are ***Pessimistic***
  - Program is correct before & after
  - Every Peephole is locally correct
  - Types strictly "lift" over time: become higher in lattice
- **GCP** is ***Optimistic***
  - All Types initialized to Top – Program is NOT correct
  - Types strictly "fall" over time
    - Until all conflicts are resolved
  - When done, types (and program) are correct
- For types, **GCP** is monotonically better than **Iter**

Pessimistic vs Optimistic

- Value() computes new Type from input Types
  - Both for **Iter** and **GCP**
  - i.e. Types are both strictly lifting or strictly falling
- Value() calls must be *monotonic*
  - Output falls IFF some input falls
  - eq: If **all** old input Types **isa** new input Types,
    Then old output Type **isa** new output Type
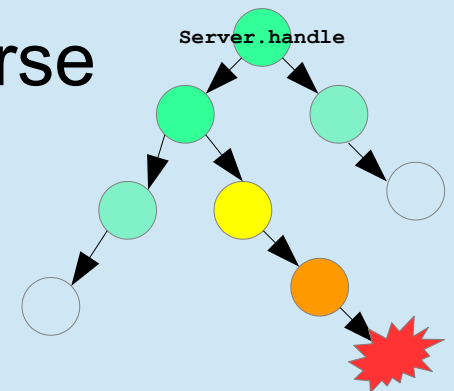
Global Code Motion / Global Value Numbering: Click

Combining Analysis, Combining Optimizations: Click, Cooper

Inlining

- Some execution hits 10K & triggers compile

- Hunt "up stack" for suitable method

  – Get some context around hot method

- Parse bytecodes, build SSA

  – Hunt "down stack" for inlines as-you-parse

  – CHA - Class Hierarchy Analysis

    - Peepholes as-you-parse to feed CHA

  – Mostly: Small & hot

  – Always: Trigger, intrinsics, trivial get/set, Unsafe

  – Exclude medium+ already compiled (i-cache blowout)

Inlining

- Decision made at bytecode-parse time
  - Often too early

- Lacks detailed optimizer knowledge:
  - In depth frequency analysis
  - More precise 'this' type
    - Or constant arguments (often null)
  - So some hit & miss on inlining
 - Which is compensated by over-inlining

Global Code Motion

- Builds a Real ™ CFG

- Unwinds the "Sea" and puts Nodes into Blocks

- Global latency-aware freq-aware scheduler

  - Code moved out of loops

  - Into low-freq branches, esp Deopt paths

- Requires anti-dependencies

  - Which is another pass, and uses precise alias info

- IR is "fragile": code motion is a thing now

Global Code Motion, Global Value Numbering: Click

Graph Coloring Register Allocator

- Probably single slowest phase in compiler
  - %time goes here?
- Responsible for <u>greatest speedup across all codes</u>
- Robust to "over-inlining"
  - Spill code costs almost always cheaper than prolog/epilog call costs of not inlining
  - Which means C2 can crank uplining without hitting a performance "knee" due to spill costs
    - Not true of many many static compilers…
- Deserves its own hour-long talk

Some Java Specific Optimizations

- All bytecode checks exposed as normal IR
  - Literally **no difference** between user null-check and built-in null-check
    - Both use same perf counters, same compiler opts
- Then make compiler robust to safety checks
  - 90% removed during bytecode parsing (Peepholes)
  - 5% as hardware check against memory op
  - 5% as explicit branch
  - Compilers of that era did not remove nearly as many
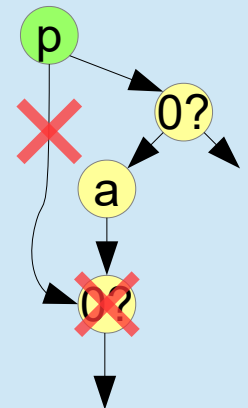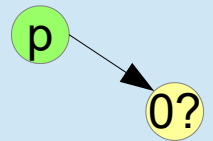
Some Java Specific Optimizations
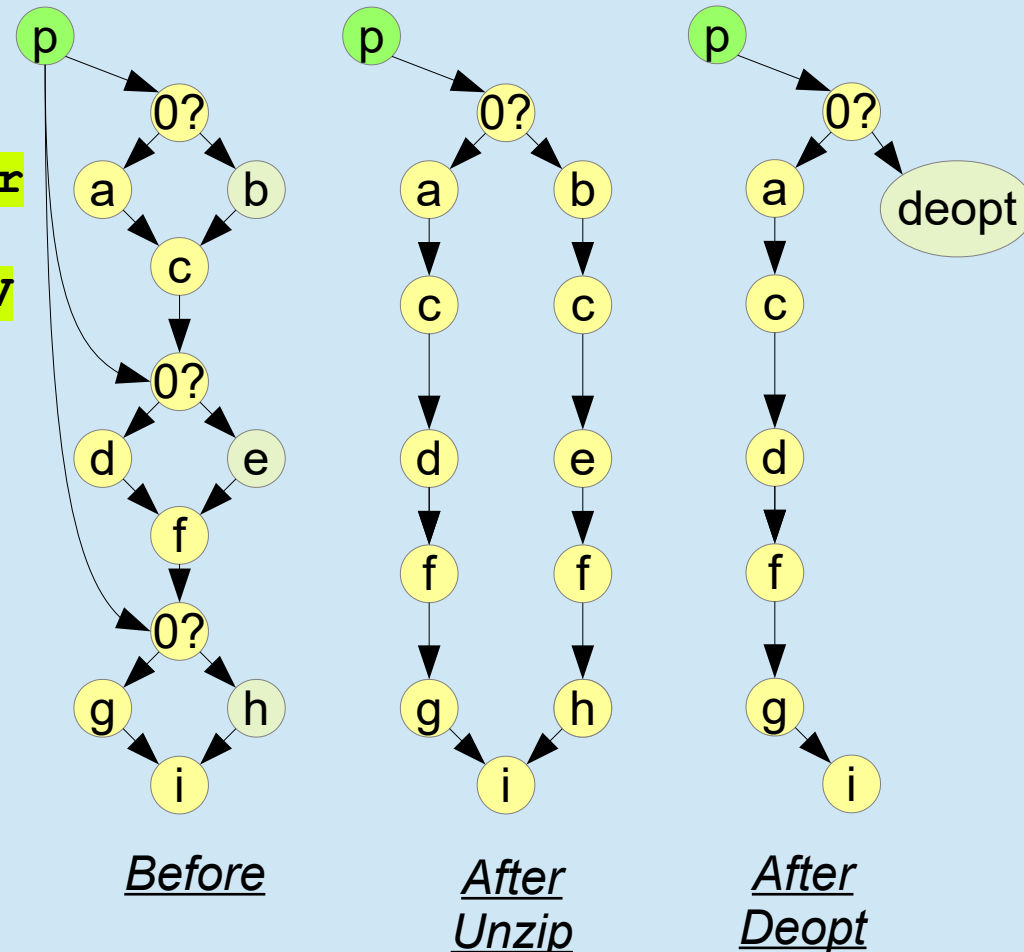
- Unzipping repeated null checks
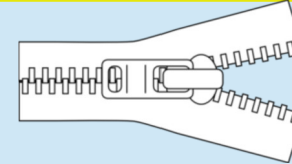
- Test fails? Unzip!
  - A lot? `jne ptr,label; ld ptr`
  - A little? `ld ptr // may SEGV`

- Test never failed (yet)?
  - Deopt on fail: exit JITd code to interpreter
  - No {b,c,e,f,h} code
  - No merge at {i}

Before

After Unzip

After Deopt

Some Java Specific Optimizations

- Calls: CHA turns 90% into static calls

  - Which enables inlining

  - Maybe with a guard test

- Remaining 10% using Inline Cache

  - Which 90% (of 10%) always hit → v-call in 2 clks

    - Compare klass in 1 clk, static call in 2nd clk

  - 10% (of 10%) make official v-call: `ld;ld;ld;jr`

    - Takes ~30clks on many processors

# Some Java Specific Optimizations

- Range Check Elimination
  - (1) Identify bounded loops & induction variables
  - (2) Peel. Removes null-checks
  - (3) Insert pre- and post- loops for edge cases
  - (4) Remove checks from inner loop
  - And **Iter**, to optimize the now-clean loop
  - (5) And for small bodies, unroll by powers of 2
    - Always capped by loop size, beware I-cache blowout!
    - And run **Iter** on the unrolled body, and repeat (5)

Some Java Specific Optimizations

- Checkcast, instanceof, arraystore
  - Plus guarded inlining
- Compare RHS klass depth in LHS klass display
  - Nearly always folds into 1-clk `ld;cmp;jne`

Fast subtype checking in the HotSpot JVM:
*Click, Rose*

Heroic Optimizations

- Many safety checks rarely fail
    - But must work fast/well if they do
    - Array-store, range checks, most null checks, guarded inlines, most classes never overload, ...
    - Paths never taken (yet)
- Heroically Assume The Best!
        (but prepare for the worst)
- Quick correctness test, then go for fast case
   - Plus "breadcrumbs" for failure recovery
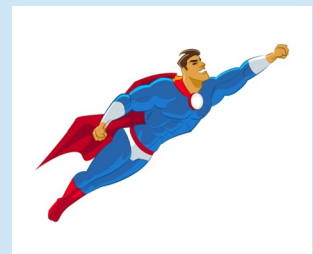
Heroic Optimizations

- Fast/common case: some quick check

- Fail case: use recorded state to deopt

  - Unwind to interpreter; literally rewrite compiled stack frame into nested interpreter frames

  - Record a failure bit…

  - Re-profile in interpreter/C1

  - Re-JIT… but with failure bit.
    Do not be so heroic 2nd time around

 - Hard part: **tracking the heroic bits** through multiple layers of inlining
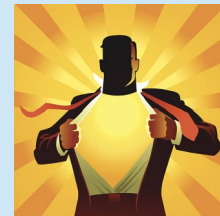
Deoptimization

- The Backup Plan!
  - Recovery option for every failure…
  - … go back to the interpreter
  - Do not try to do **everything**
  - Just what is hot & can be compiled
- Track JVM state, same as X86 tracks state
  - Tracked at "Safepoints", not everywhere
  - Mapping from regs, stack, constants to JVM stack
  - (not really related to GC safepoints)
- But still must generate good code...
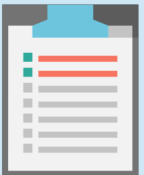
Deoptimization

- From the compiler's point-of-view...

  – A Safepoint is like a rarely taken Call

    • That reads all JVM state including memory effects

  – So treat it like that!

- Safepoints are just a Node

  – Use→Def to every JVM state (forces keep alive)

  – Very low exec frequency (code scheduler, spill code)

  – Effect: Bits needed for interpreter but not JITd code get spilled to stack and moved into off-paths

- Still great code in fast-path, but can unwind
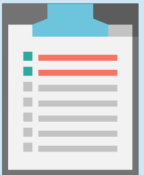
Debugging C2

- Same problem the Heroic Opts target:
  - Common fast case, rare complex slow case
  - Bugs persist in rare complex cases, how to find?
- Make rare case common!
  - And prepare for a 10x slowdown (weekend QA runs)
  - +BlahBlahBlahALot debugging flags
- Many 1000's of compiles
  - Binary search with -XX:CIStart/CIStop,
    bunch of other options for gating e.g. inlining

Summary

- Sea of Nodes!
  - All program semantics in Nodes & Edges
    - Nodes are tiny and Edges are raw pointers for **speed**
  - Graph Rewrite Rules ***Rule***
    - Nearly all "easy" optimizations done with Peepholes
- Types: Fast & Precise
  - Defines what a compiler can "talk" about
  - Theory: Boolean Algebra, Complete, Distributive, Complemented, Bounded (Ranked) Lattice
  - Defined via meet & dual; build using hash-cons

Summary

- C2 has many Java-specific optimizations
  - Fast-path/slow-path
    - Heroic versions of same: deopt instead of slow-path
  - Null check unzipping
  - Range Check Elimination
  - Subtype checks, CHA
- Aggressive Inlining
  - Done perhaps too early, with too little info
- Expensive Graph-Coloring Register Allocator
 - Makes up for the sins of over-inlining

Summary

- C2 has withstood the Test of Time

    - Including its creator leaving

- Typical optimizing compiler lifetime is 20+ years

    - So expect a **lot of cruft** in there

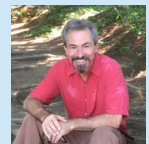- But the Bones of C2 remain, and remain simple

Summary

- C2 has withstood the Test of Time

    – Including its creator leaving

- Typical optimizing compiler lifetime is 20+ years

    – So expect a **lot of cruft** in there

- But the Bones of C2 remain, and remain simple

Hacking C2 was a Labor of Love!
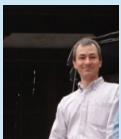
I hope you enjoy hacking it too!

Resources

- Modern Compiler Textbook:

  - *Engineering: A Compiler: Keith Cooper, Linda Torczon*

- Optimistic vs Pessimistic; Type theory

  - *Combining Analysis, Combining Optimizations: Click, Cooper*

  - *https://en.wikipedia.org/wiki/Lattice_(order)*

- Design of the C2 IR:

  - *A Simple Graph-Based IR: Click, Paleczny*

  - *From Quads to Graphs: An IR's Journey: Click*

  - *Global Code Motion, Global Value Numbering: Click*

  - *The Java HotSpot Server Compiler: Click, Paleczny, Vick*

  - *Fast subtype checking in the HotSpot JVM: Click, Rose*

Q&A

Cliff Click