Abstract Interpretation vs “Just Flow Stuff”

Alon Zakai / March 2025

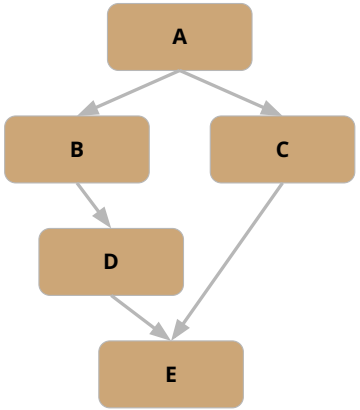
WHO WOULD WIN?

$$\bigsqcup_{i \geq 0} f^i(\bot)$$

$$x \sqsubseteq \gamma(y)$$

Rigorous algorithm studied in academia

One flowey boi



The Optimization Problem (simplified)

```
(module
  (type $struct (struct (field mut i32)))

  (global i32 $global (i32.const 42))

  (func “work” (result i32)
    (struct.get $struct 0
      (struct.new $struct
        (global.get $global)
      )
    )
  )
)
```

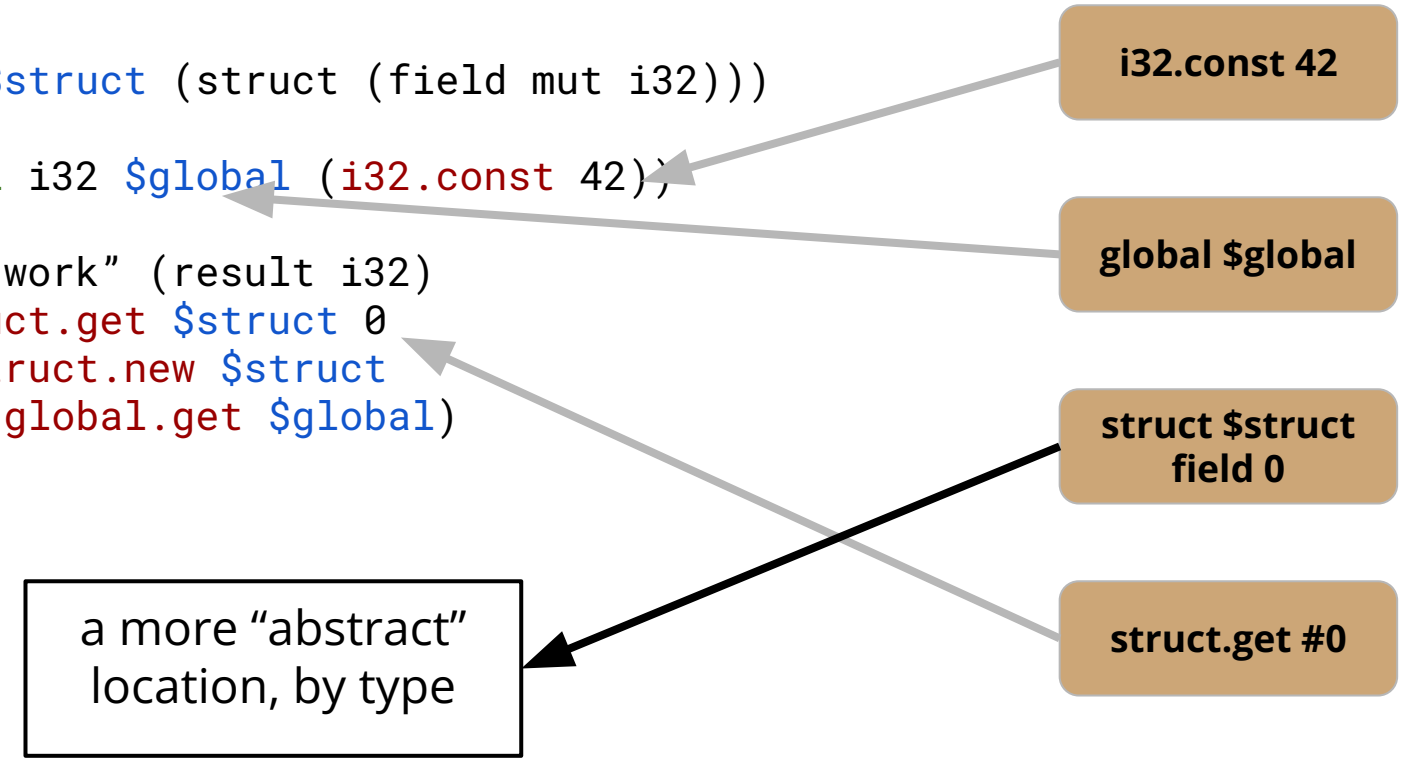
How do we write an optimization pass that does this kind of thing? (note: function code, globals, heap)

Define “Locations” that can contain Values

```
(module
  (type $struct (struct (field mut i32)))

  (global i32 $global (i32.const 42))

  (func “work” (result i32)
    (struct.get $struct 0
      (struct.new $struct
        (global.get $global)
      )
    )
  )
)
```

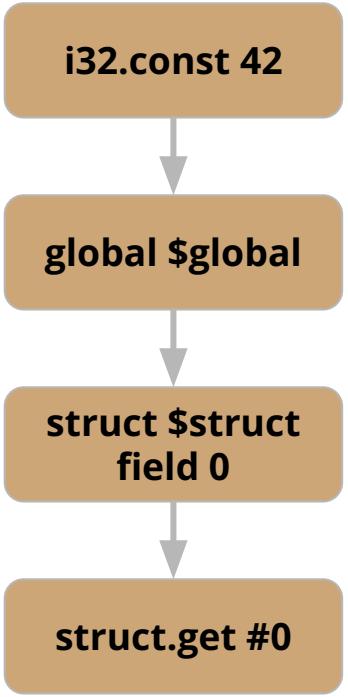


Connect the Locations (where values flow)

```
(module
  (type $struct (struct (field mut i32)))

  (global i32 $global (i32.const 42))

  (func “work” (result i32)
    (struct.get $struct 0
      (struct.new $struct
        (global.get $global)
      )
    )
  )
)
```



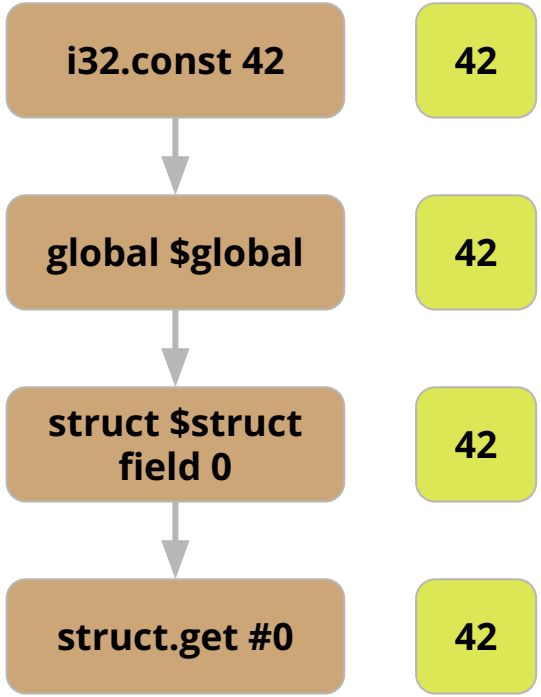
Just Flow Stuff

```
(module
  (type $struct (struct (field mut i32)))

  (global i32 $global (i32.const 42))

  (func “work” (result i32)
    (struct.get $struct 0
      (struct.new $struct
        (global.get $global)
      )
    )
  )
)
```

And so the function returns 42!



Just Flow Stuff: The Simplest Algorithm

1. Define locations that can contain values
2. Connect them where values can flow around
3. Flow:
    a. When a new value arrives at a location **L**, add it
    b. Send the resulting value at **L** to **L**'s targets
    c. Keep going while stuff is changing

Just Flow Stuff: The Simplest Algorithm

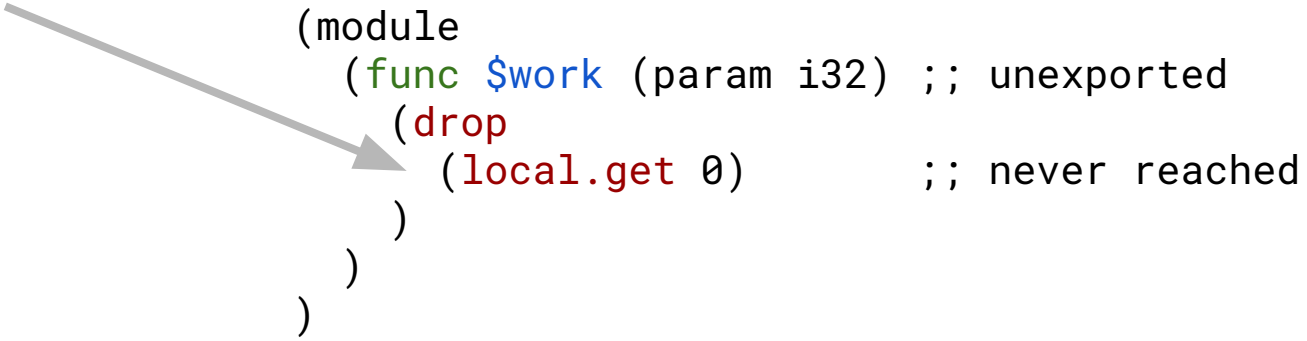
1. Define locations that can contain values
2. Connect them where values can flow around
3. Flow:
   a. When a new value arrives at a location **L**, **add it**
   b. Send the resulting value at **L** to **L**'s targets
   c. Keep going while stuff is changing

a lot to say about this, it turns out... what are the values? how do we add them?

GUFA: Grand Unified Flow Analysis

Exactly what we've seen so far, implemented in Binaryen on these values:

- **None (empty)**



```
(module
  (func $work (param i32) ;; unexported
   (drop
    (local.get 0)        ;; never reached
   )
  )
)
```

GUFA: Grand Unified Flow Analysis

Exactly what we've seen so far, implemented in Binaryen on these values:

- **None (empty)**
- **Single constant**

```
(i32.const 42)
(ref.func $foo)
;; etc.
```

GUFA: Grand Unified Flow Analysis

Exactly what we've seen so far, implemented in Binaryen on these values:

- **None (empty)**
- **Single constant**
- **Immutable global**

```
;; (not important today)

(import "a" "b" (global $g i32))

(func (result i32)
  (global.get $g) ;; while we have no idea
                  ;; what the actual value
                  ;; is, but it is what is
                  ;; in the global (hence
                  ;; two calls to here
                  ;; will compare equal)
)
```

GUFA: Grand Unified Flow Analysis

Exactly what we've seen so far, implemented in Binaryen on these values:

- **None (empty)**
- **Single constant**
- **Immutable global**
- **Cone: wasm type+depth**

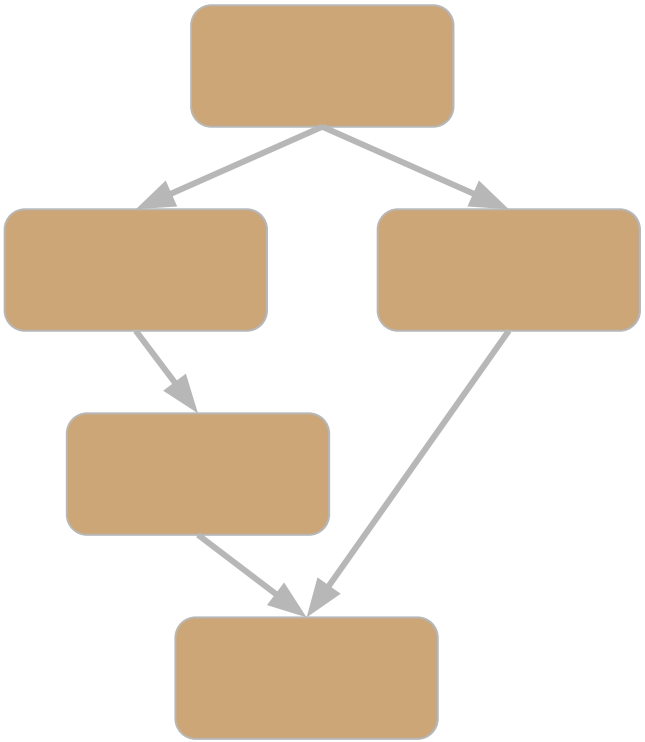
```
;; chain A :> B :> C
(type $A (struct))
(type $B (sub $A (struct)))
(type $C (sub $B (struct)))

(select                ;; Cone(A, 1)
  (struct.new $A)      ;; Cone(A, 0)
  (struct.new $B)      ;; Cone(B, 0)

;; Cone(A, 1) === { A, B }
;; (no C!)
```

```
(select              ;; Cone(i32, ∞) === i32
  (i32.const 0)      ;; i32.const 0
  (i32.const 1)      ;; i32.const 1
```

Ok, we have a graph of locations, and we flow stuff...

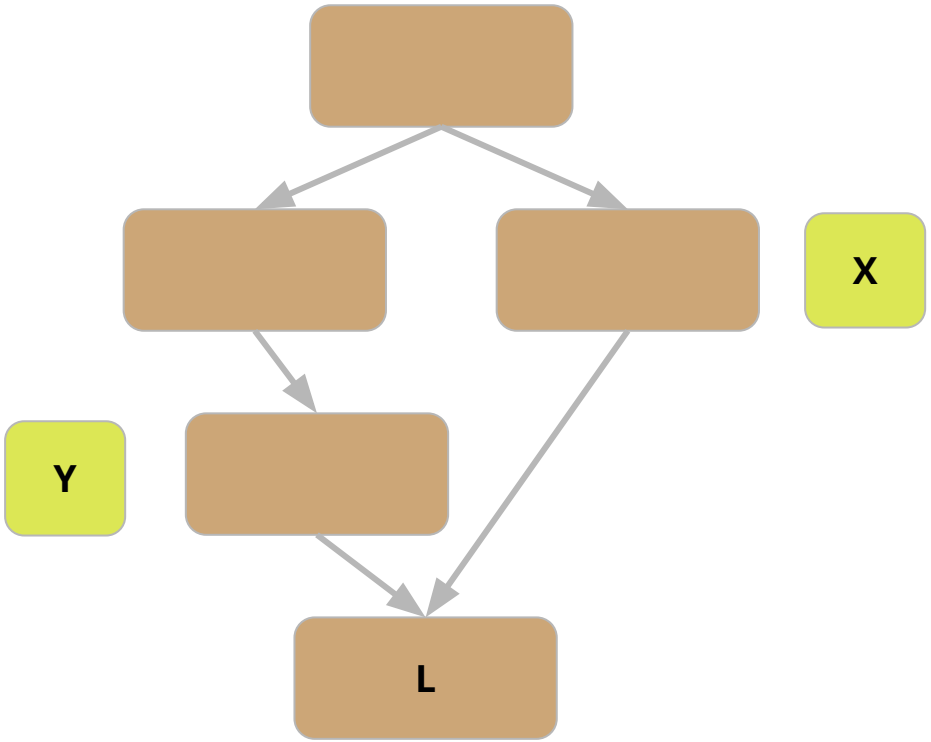


What properties does this have?

1. Does it **converge**? (will it finish running no matter the input)

2. Is it **deterministic**? (will it always return the same output)

Convergence: YES



L receives the values **X** and **Y**. Say **X** arrives first. The first update:

None ⊔ X => X

(⊔ == “join”, union). Second update:

X ⊔ Y => ..

Whatever that is, it is greater than **X**, **Y**: monotonic. + finite # of values => convergence

Determinism..?

First, let us write out the full “add in new content” operation.

For a location **L** with existing content **E** and new content **N**, we update **L**’s content to:

$$(E \sqcup N) \sqcap T_L$$

where T_L is the type of the location **L** - since we trust the wasm type system! - and $\sqcap$ is “meet” (intersection).

Determinism..?

For example:

```
(func (param $ref (ref null $X))
  (block $b (result (ref $X))
    (br_on_non_null $b
      (local.get $ref)))
    ..
```

We naively flow the nullable local to the block, then use the block's type to **filter out nulls**.

(We could instead reason that br_on_non_null sends non-null, but it is simpler to use $b's type.)

What does any of this have to do with Determinism..?!

Recall that the update rule is (for existing content **E**, new content **N**, and known wasm type at that location **T**):

(E ⊔ N) ⊓ T

If **X** appears before **Y**, we have, after two updates:

(((E ⊔ X) ⊓ T) ⊔ Y) ⊓ T

Or, if **Y** is first:

(((E ⊔ Y) ⊓ T) ⊔ X) ⊓ T

What does any of this have to do with Determinism..?!

((X ⊓ T) ⊔ Y) ⊓ T ??? ((Y ⊓ T) ⊔ X) ⊓ T

If these differ then the order matters, and determinism is lost (unless we fix some order, which is generally slower).

These are equal if we have the **distributive** property in math. Do we?

We do **not** have the distributive property :(Consider:

```
X = (ref.func $foo) = $foo    ;; a specific function
Y = (ref.null)      = null    ;; a null
T = (ref func)                ;; any func, no null
```

Note:

```
$foo ⊔ null == (ref null func)
```

Only a cone can contain two constants (a func and a null). This “lossy” operation makes the order matter:

```
(($foo ⊓ (ref func)) ⊔ null) ⊓ (ref func) == (ref func)

((null ⊓ (ref func)) ⊔ $foo) ⊓ (ref func) == $foo
```

Why did that happen? Recall that our values are:

- **None (empty)**
- ***Single* constant**
- **Immutable global**
- **Cone: wasm type+depth**

If we allowed sets of **multiple** constants we'd be ok, but they would need to be of arbitrary size, and this adds overhead (often for little optimization benefit). Tradeoff.

What to do?

Well, we can just fix the nondeterminism by **picking an order**, as mentioned before. Slightly slower.

But we may end up with the “**bad**” order in practice! We want the good one.

Is there some more **principled** approach to all of this, perhaps studied in academia..?

Yes: AI!

...no, not that AI: **A**bstract **I**nterpretation

Analyze by computing a **transfer function** on each location.

For example, an **i32.add**'s transfer function would be:

f(left, right) = left + right

But "**abstractly.**"

[meme: fancy winnie the pooh]

E.g. abstracting over multiple values, if left is a value in **{ 0, 10 }** and right is **{ 20 }**, then **f** returns **{ 20, 30 }**. And we iterate to a fixed point.

Abstract Interpretation

When the values have nice properties (commutativity, etc.) then they form a “lattice.” Abstract Interpretation on a finite lattice is guaranteed to converge to the **minimal** fixed point. Great!

Wait, why doesn’t “Just Flow Stuff” have the same guarantee?

First, as we saw, for practical reasons we lack some of the “nice properties” like distributivity (and perhaps others, for similar reasons).

But there is another reason.

“Just Flow Stuff” vs Abstract Interpretation

Just Flow Stuff combines existing and new, and intersects with the type:

$$\mathbf{E \leftarrow (E \sqcup N) \sqcap T}$$

Abstract Interpretation (here) combines all the sources, then intersects:

$$\mathbf{E \leftarrow (S_1 \sqcup S_2 \sqcup .. S_N) \sqcap T}$$

Just Flow Stuff “accumulates” in place (**E** is on both sides). That is why we end up with nested joins and meets (**((X ⊓ T) ⊔ Y) ⊓ T**) and our breaking of distributivity was dangerous! Accumulation can accumulate “lossiness.”

So let’s go with Abstract Interpretation!

Abstract Interpretation emits the optimal thing, right?

[meme: padme “x, right? …right?”]

Both Algorithms Disappoint

Reminder: Just Flow Stuff computes one of these:

```
(($foo ⊓ (ref func)) ⊔ null) ⊓ (ref func) == (ref func)
```

```
((null ⊓ (ref func)) ⊔ $foo) ⊓ (ref func) == $foo
```

Abstract Interpretation computes this:

```
($foo ⊔ null) ⊓ (ref func) == (ref func)
```

So it always emits the **worse** thing... Wait, how is that possible?

Both Algorithms Disappoint

Abstract Interpretation emits the optimal thing **for a given transfer function and values**. Our values allow only a single constant, so once more, joining two constants (a func and a null) is a “lossy” operation.

Fixing the values is hard (overhead, as we mentioned). But can we fix the transfer function?

Both Algorithms Are Fixable

Yes! Just Flow Stuff was sometimes optimal, if we were lucky and got the right order. That other order added a meet ⊓. We can fix both in that way.

Just Flow Stuff:

$$E \leftarrow (E \sqcup (N \sqcap \underset{\wedge\wedge}{T})) \sqcap T$$

Abstract Interpretation:

$$E \leftarrow ((S_1 \sqcap \underset{\wedge\wedge}{T}) \sqcup (S_2 \sqcap \underset{\wedge\wedge}{T}) \sqcup .. (S_N \sqcap \underset{\wedge\wedge}{T})) \sqcap T$$

Just Flow Stuff vs Abstract Interpretation

Both have the same problem, both have the same solution.

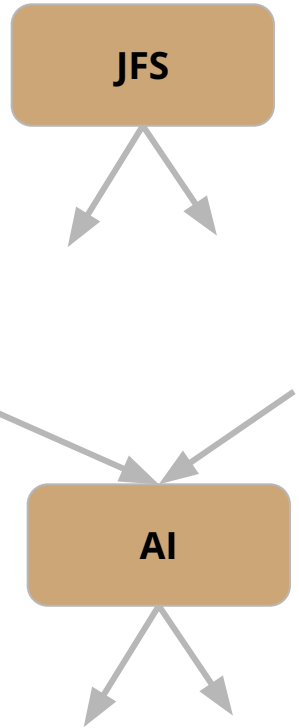
So what is the actual difference?

Tradeoffs: Memory

Just Flow Stuff only flows **forward** (let targets know something changed).

Abstract Interpretation requires **bidirectional links**, to go back and read all sources.

- More memory for the graph
- Reads of sources often not cache-friendly



Tradeoffs: Memory

How big is the efficiency downside of Abstract Interpretation?

I converted GUFA to use that approach in a branch. Diffs:

	Time (seconds)	Memory (MB)
Java (calcworker)	9%	14%
Dart (complex)	11%	18%
C++ (clang)	6%	11%

Tradeoffs: Generalization

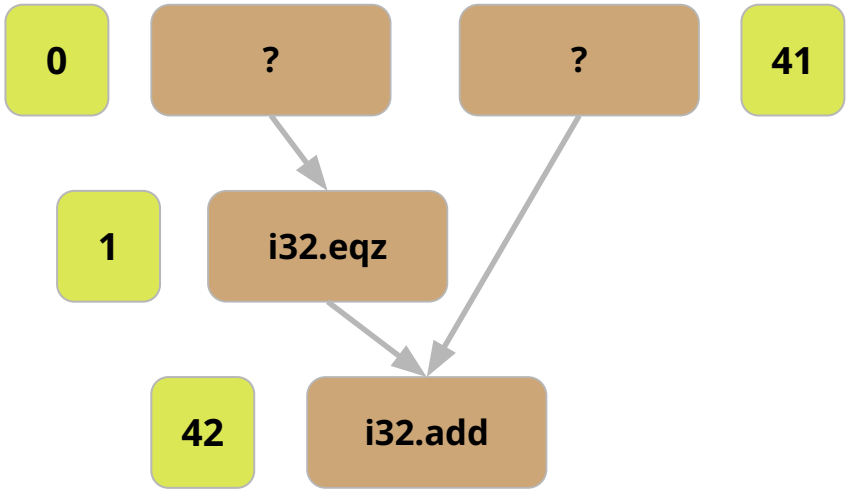
Abstract Interpretation is **generalizable**: Sees all sources at once, and can define arbitrary transfer functions **`f(`S_1`, `S_2`, .., `S_n`)`**.

If all we do is flow values around we don't need that, but if we want anything more - even an **`i32.add`** - then we need Abstract Interpretation.

However...

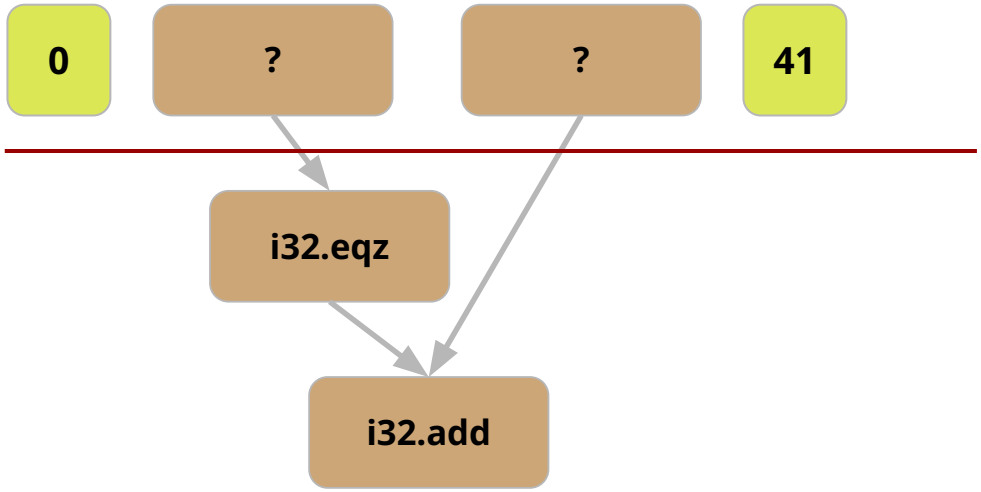
Tradeoffs: Generalization

Abstract Interpretation computes as it goes:



Tradeoffs: Generalization

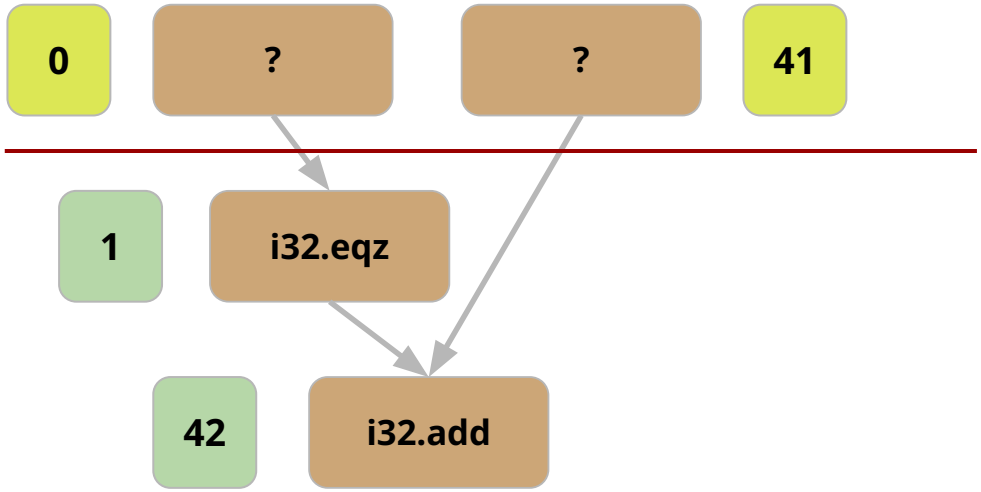
Abstract Interpretation computes as it goes:



Just Flow Stuff, by itself, stops here

Tradeoffs: Generalization

Abstract Interpretation computes as it goes:



Just Flow Stuff, by itself, stops here

But the main optimization pipeline uses the constants Just Flow Stuff inferred, for the same results as Abstract Interpretation (however: efficiency? cycles?)

Tradeoffs: Parallelization

Abstract Interpretation has a natural way to run in parallel. Consider two locations **A** and **B**, then their updates are:

$$A_{new} = f_A(B_{old}, X_{old}) \qquad (= (B_{old} \sqcup X_{old}) \sqcap T_A)$$

$$B_{new} = f_B(Y_{old}, A_{old})$$

Note how they read from each other's old state. But that is not a problem if we keep the old state fixed as we generate the new state in parallel.

Tradeoffs: Parallelization

Given the same graph, Just Flow Stuff has rules like this:

```
A <- (B ⊔ A) ⊓ T_A    ;; add B to A
A <- (X ⊔ A) ⊓ T_A    ;; add X to A
B <- (Y ⊔ B) ⊓ T_B    ;; add Y to B
B <- (A ⊔ B) ⊓ T_B    ;; add A to B
```

We can do each update as a *compare-and-swap*, at the risk of contention?

Another approach is to parallelize inside functions (as other passes do), interleaving global updates?

Tradeoffs: Parallelization

There is a simpler way to parallelize Just Flow Stuff: Take one rule,

$A \leftarrow (B \sqcup A) \sqcap T_A$;; add B to A

Rather than update **A** in place, we can add **B** to a "**mailbox**" **M** for **A**:

M_A.append(B)

Then we can operate on the mailboxes in parallel. In fact, we know which index in the mailbox to use for each source (index in list of **A**'s sources):

$M_A[I_{B,A}] = B$

Then Just Flow Stuff does this in each (parallel) update:

$$A_{new} \leftarrow (M_A[I_{B,A}] \sqcup M_A[I_{A,A}]) \sqcap T_A = (B_{old} \sqcup A_{old}) \sqcap T_A$$

since the values in the mailbox are exactly the values from the old iteration.

And that is exactly the rule for Abstract Interpretation!

[meme: pam from the office, “they’re the same picture”]

Conclusion

Just Flow Stuff *is* Abstract Interpretation, where the transfer function just flows values (no computation), and updates are incremental/in-place.

- **Just Flow Stuff** is **10-15%** more efficient in time and memory
- **Abstract Interpretation** is **generalizable** to more things, and **parallelizable**

Worth generalizing GUFA if/when we consider adding new optimizations.

Thanks to **@tlively** for the discussion that led to this talk!

Questions?