# SEMANTIC CODE SEARCH VIA EQUATIONAL REASONING

Varot Premtoon, James Koppel, Armando Solar-Lezama

Massachusetts Institute of Technology

PLDI 2020

Priya Srikumar

Cornell University

Fall PLDG 2020

```python
count = 0
for a in cart:
    if a == item
        count += 1
use (count)
```

```
count = 0
for a in cart:
    if a == item
        count += 1
use (count)
```

```
use(cart.count(item))
```

```python
count = 0
for i in cart:
    if debug:
        print(cart[i])
    if cart[i] == item:
        count += 1
use(count)
```

```python
count = 0
for i in range(len(arr)):
    if item != arr[i]:
      continue
    count += 1
use(count)
```

```
count = 0
i = 0
while i < len(cart):
  if cart[i] == k:
    count += 1
   i += 1
use(count)
```

```
for .* in cart
```

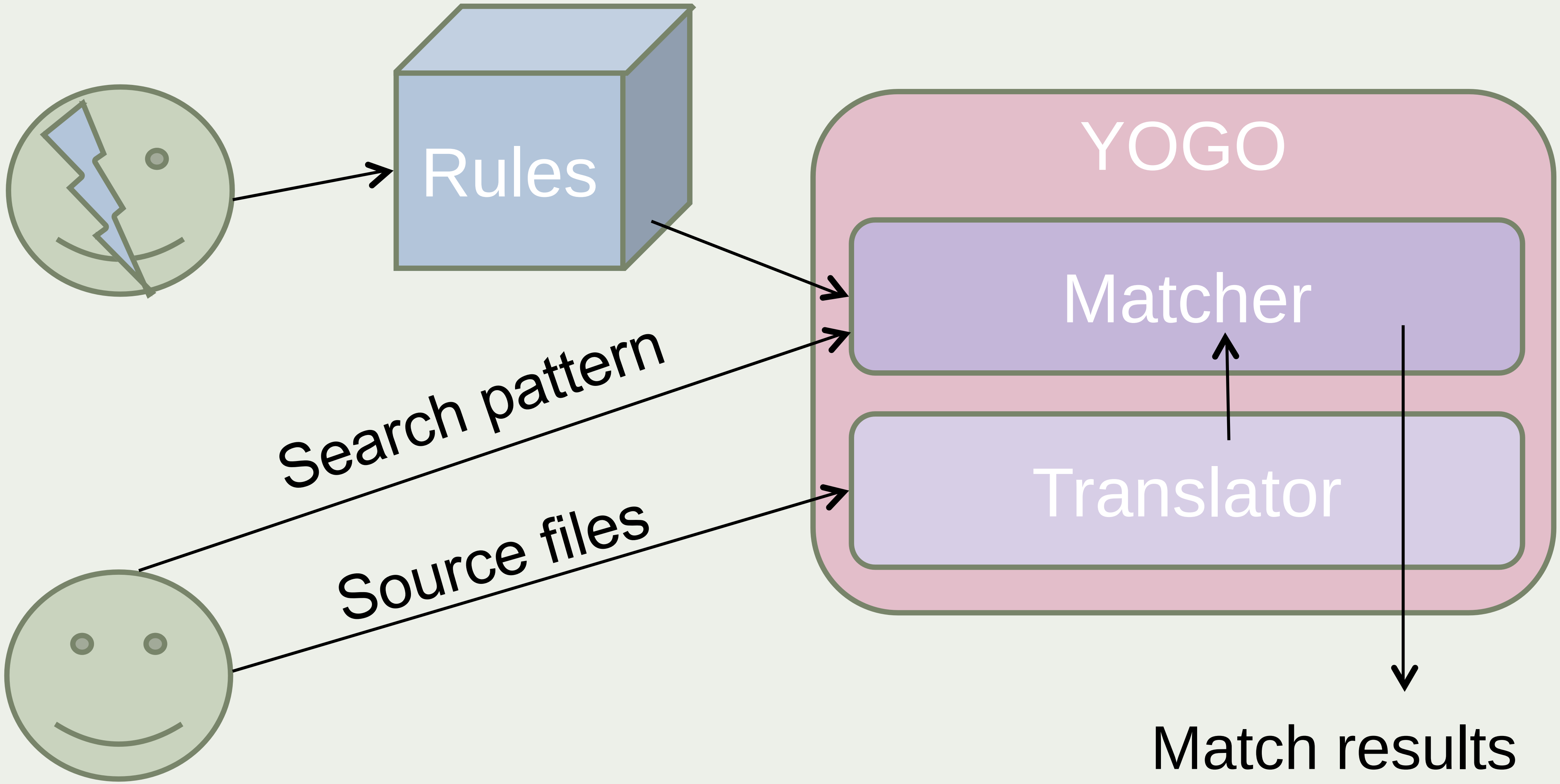
# Semantic code search

Scope

Query type

Performance needs

# You Only Grep Once


Rules
YOGO
Matcher
Translator
Search pattern
Source files
Match results

# Operational evaluation

Search patterns gleaned from common StackOverflow questions

YOGO times out on 1% of the methods searched

False negatives due to tool incompleteness and reasoning limitations

# Graal

13 static analyzers on top of Graal’s bytecode analysis

YOGO analyzed 1.2 million LOC in 2.5 hours on 30 AWS instances

Generalizing one of the analyzers’ queries revealed a bug it missed!

YOGO query was 60 lines; the analyzer was 330 lines

# Usability evaluation

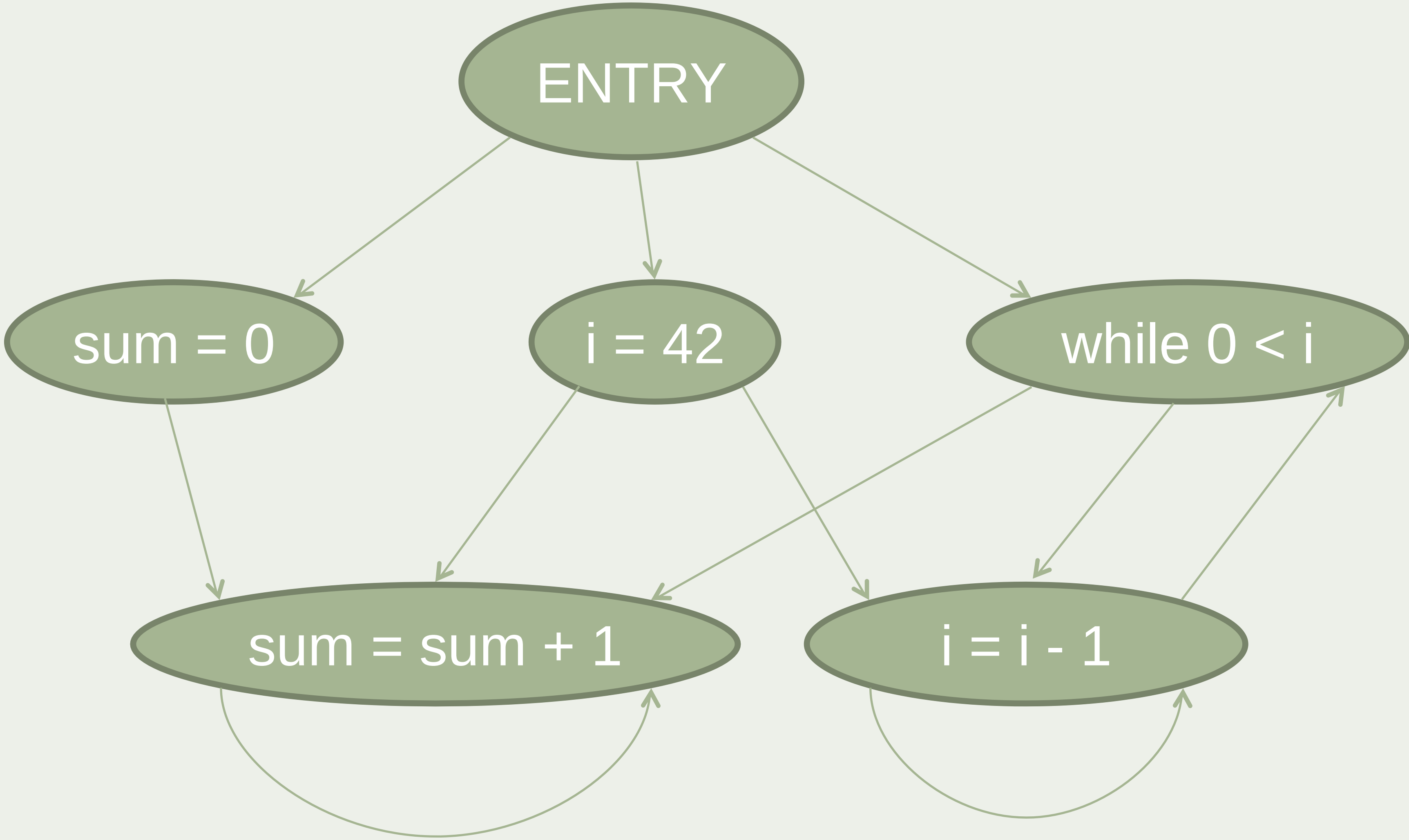
YOGO was straightforward enough to use that the authors

**outsourced Graal query coding; it only took 3 days to learn!**

```
i = 42
while (0 < i)
    i = i - 1
```

```
sum = 0
i = 42
while (0 < i)
    sum = sum + 1
    i = i - 1
```

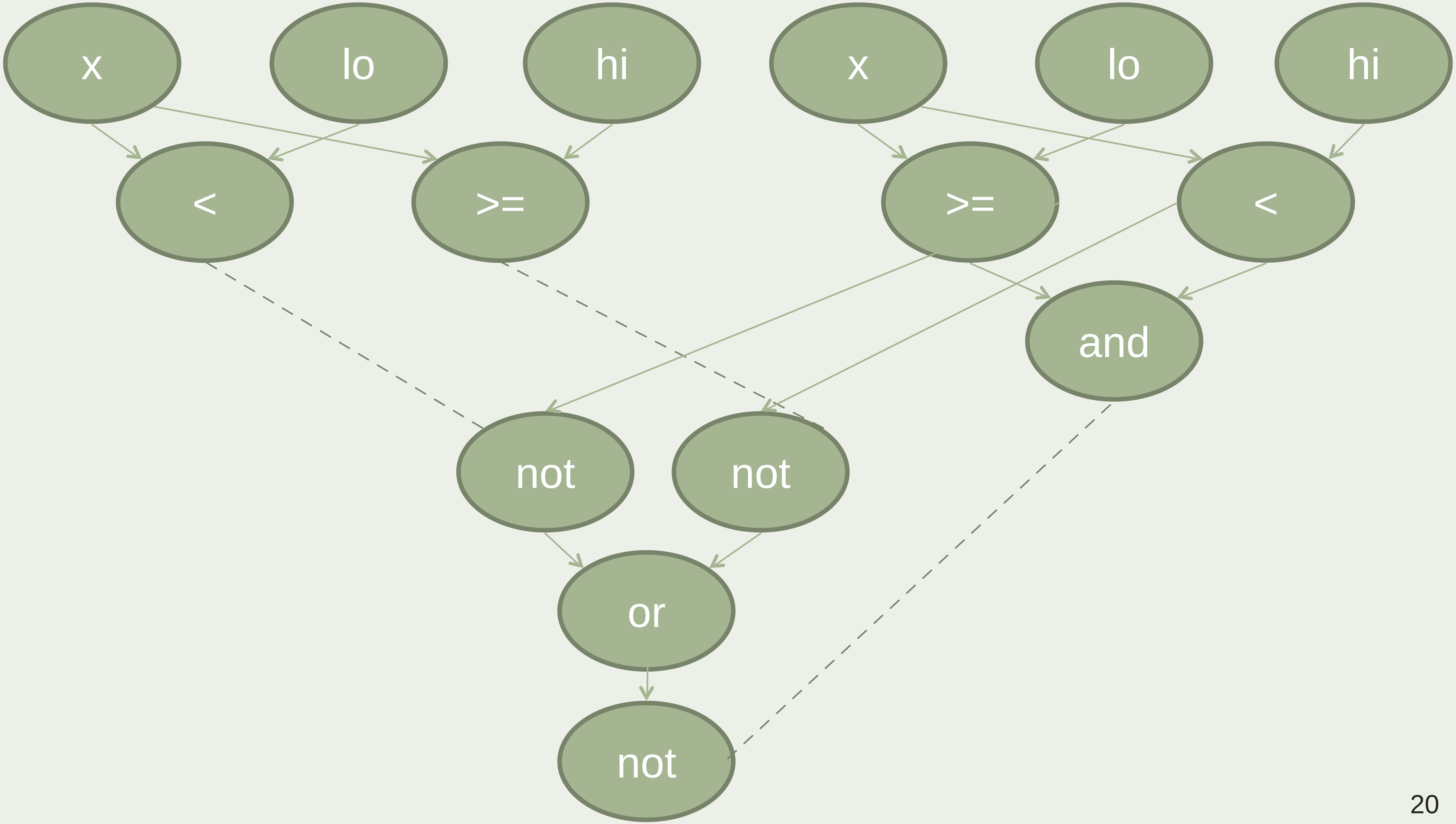
```
i = 42
while (0 < i)
    i = i - 1
```



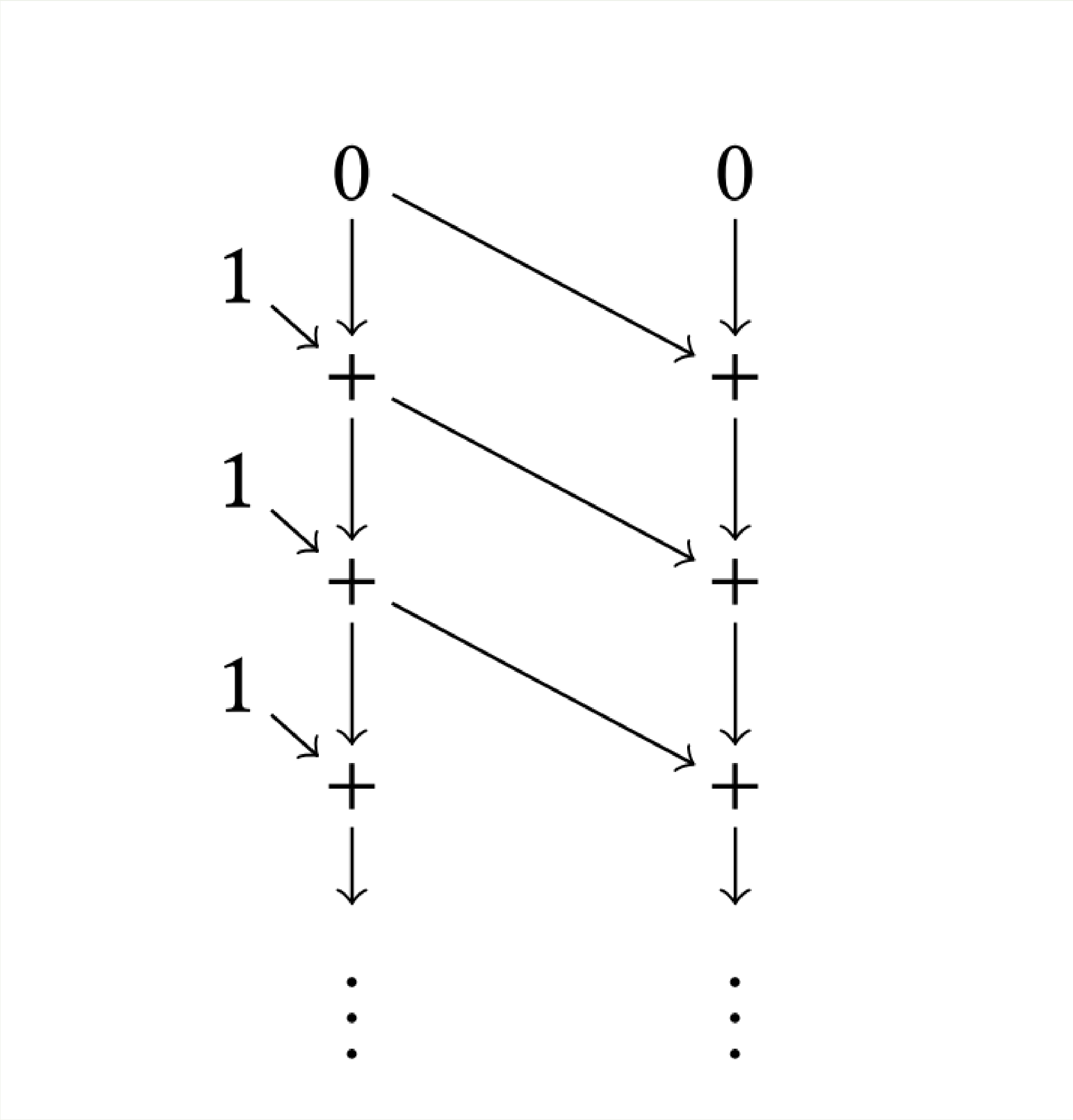
# Program Expression Graphs give us an efficient procedure for code equivalence.

```
x >= lo && x < hi
```

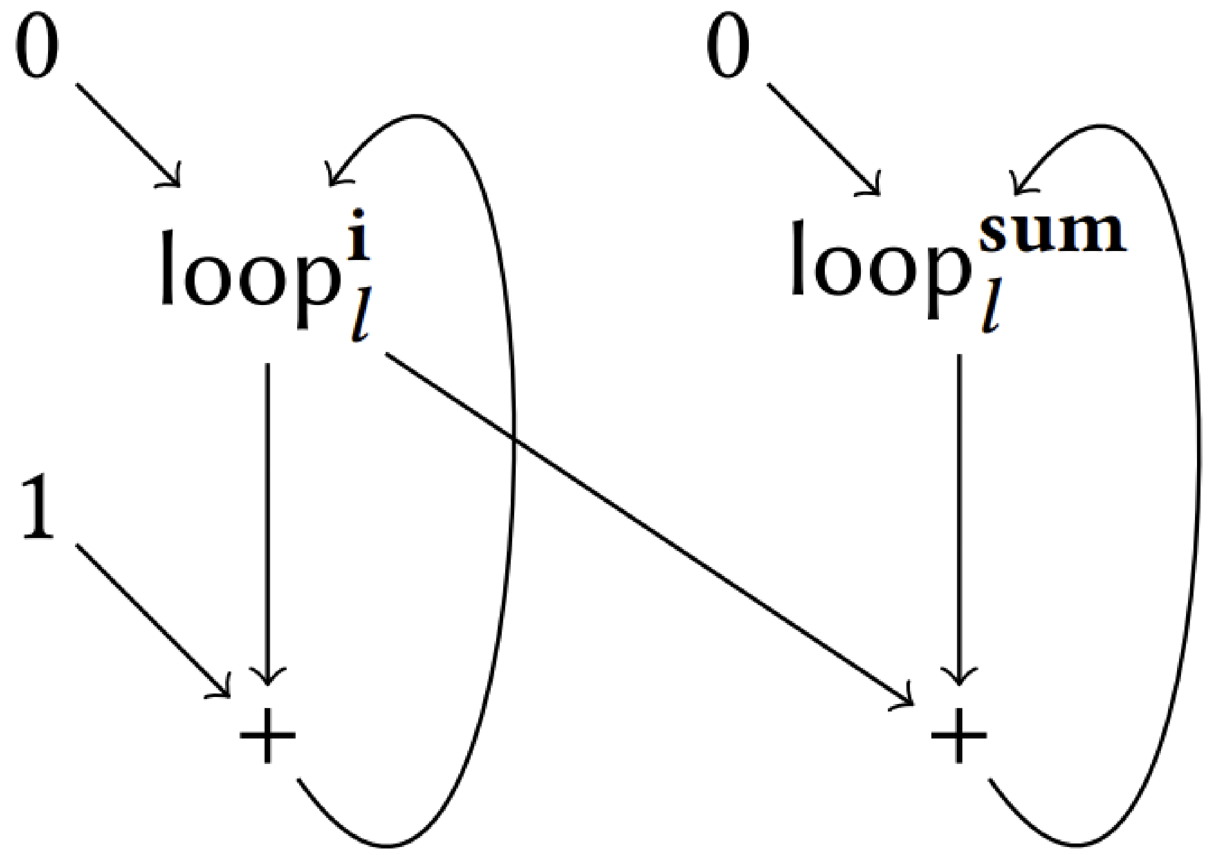
```
(defsearch bound-checking
    (root <- (generic/binop :or (generic/binop :< x lo)
                                (generic/binop :>= x hi))))
```


x
lo
hi
x
lo
hi
<
>=
>=
<
and
not
not
or
not

```
sum = 0;
for(i = 0; ...; i++) {
  sum += i;
}
```


0
0
1
+
+
1
+
+
1
+
+

$$\mathrm{loop}_{l}^{\mathbf{sum}}$$


0
$\mathrm{loop}_l^{\mathbf{i}}$
1
+
0
$\mathrm{loop}_l^{\mathbf{sum}}$
+

| Node | Denotation |
| --- | --- |
| $\text{loop}(e_0, e)$ | $\lambda i.\begin{cases} \mathrm{e}_0 & \mathrm{i} = 0 \\ \mathrm{e(i\text{-}1)} & \text{otherwise} \end{cases}$ |
| $\text{final}(e, l)$ | $\lambda i.l(\min_{j\in\mathbb{N}} e(j) = \mathbf{false})$ |

```
i = 0
while ...:
  i = i + 1
```

# High-level concepts can be identified as dataflow patterns.

| Node | Denotation |
|---|---|
| $\mathrm{seq}(i, k)$ | $(i, i + K, i + 2k, \ldots)$ |

```
i = 0
while ...:
  i = i + 1
```

| Node | Denotation |
| --- | --- |
| $\mathsf{iterV}((t_0, t_1, \ldots))$ | $\lambda i.\begin{cases} \mathsf{t}_i & \mathsf{t}_i \neq \bot \\ \bot & \text{otherwise} \end{cases}$ |
| $\mathsf{iterP}((t_0, t_1, \ldots))$ | $\lambda i.\begin{cases} \mathbf{true} & \mathsf{t}_i \neq \bot \\ \mathbf{false} & \text{otherwise} \end{cases}$ |

$$\textsc{invariant}(l, e)$$

$$\textsc{pure}(f)$$

$$\textsc{independent}(\lambda_1, \lambda_2)$$

```
count = 0
for a in cart:
    if a == item
        count += 1
use (count)
```

$$\text{counter} \leftarrow \text{loop}_l(0, \text{next})$$

$$
\begin{aligned}
\text{counter} &\leftarrow \text{loop}_l(0, \text{next}) \\
\text{next} &\leftarrow \text{cond}(\text{iterV}_l(\text{coll}) = k, \\
&\qquad\qquad\quad \text{counter} + 1, \\
&\qquad\qquad\quad \text{counter}) \\
\text{answer} &\leftarrow \text{final}_l(\text{iterP}_l(\text{coll}), \text{counter})
\end{aligned}
$$

$$\textsc{invariant}_l(k)$$

# Conceptual rewrite soundness

WTS: Code refines behavior of concept specification

Rewrites must be sound wrt their concept’s state specification

Rule writers prove soundness; users enjoy the benefits!

# Key contributions

Coarser notion of equality supports hierarchical abstraction

Higher-level denotations in the DSL = more abstract representations

Not a fixed set of equational rules! User-defined rewrite rules

# Limitations

Rewriting is hard: syntactic differences, interleaving, overlapping

DSL cannot handle arbitrarily-nested expressions i.e. f(g(x))

Impure function calls assume no relation between memory states

Interprocedural patterns not supported; cause graph explosion

# Comparison to other code search methods

Static analysis abstractions are too coarse

SMT methods struggle with loops and function calls

Other searches do not scale well

# Reflections

Code searcher + bug finder

Generality makes abilities hard to characterize

Presents a technique with a lot of reusability