SOFTWARE RELIABILITY GROUP

Pending Constraints in Symbolic Execution for Better Exploration and Seeding

Timotej Kapus Frank Busse Cristian Cadar
Imperial College London

Symbolic Execution

- Program analysis technique
- Active research area
- Used in industry
    - IntelliTest, SAGE
    - KLOVER

Angr

Why symbolic execution?

- *No false positives!*
  - Every bug found has a concrete input triggering it
- Can interact with the environment
  - I/O, unmodeled libraries
- Only relevant code executed "symbolically", the rest is fast "native" execution

Why (not) symbolic execution?

- Scalability, scalability, scalability
  - Constraint solving is hard
  - Path explosion

This talk

Introduce pending constraints which enhance the scalability of symbolic execution via aggressively tackling paths that are known to be feasible.

“known to be feasible”

Caching

- Cache assignments from previous solver queries
- Already widely adopted

Seeding

- External, usually valid concrete inputs
- Used to bootstrap symbolic execution
- From test-suites, examples, production data

Symbolic execution example: get_sign

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

get_sign(***x***);

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments

∅

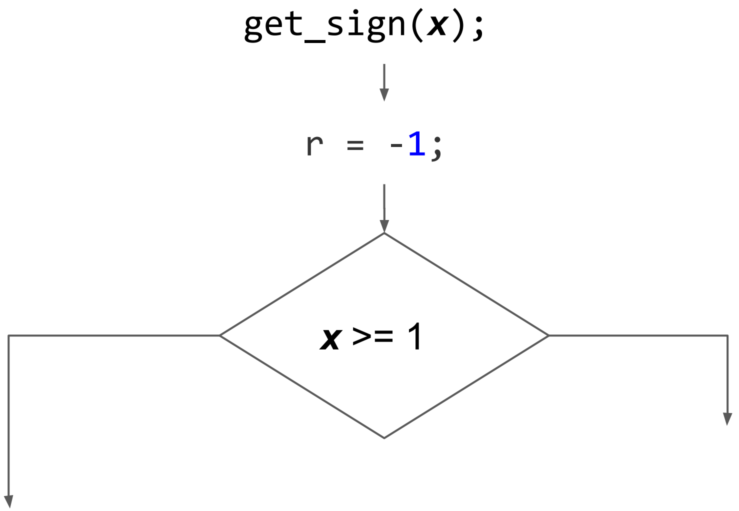
```
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

```
get_sign(x);
    ↓
r = -1;
    ↓
```

Known assignments

∅

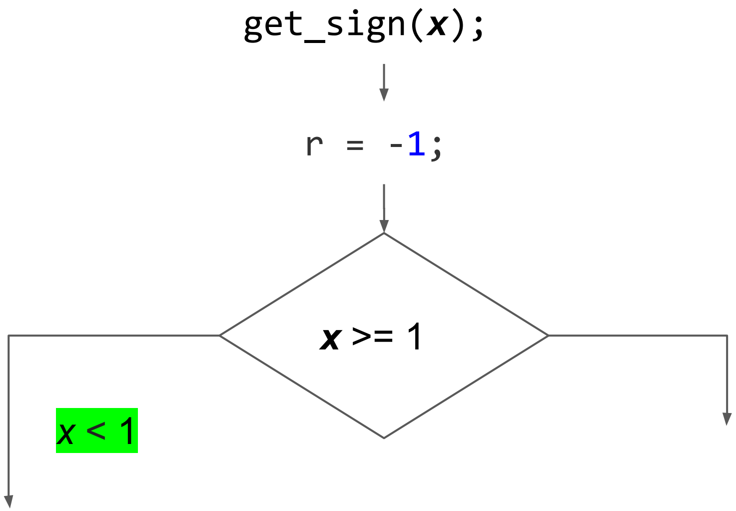
```
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

∅

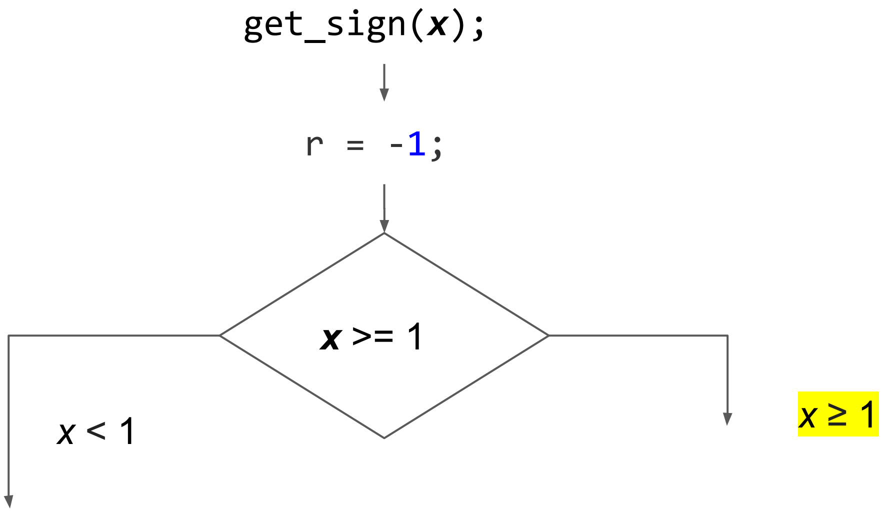
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

$x = -2$

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

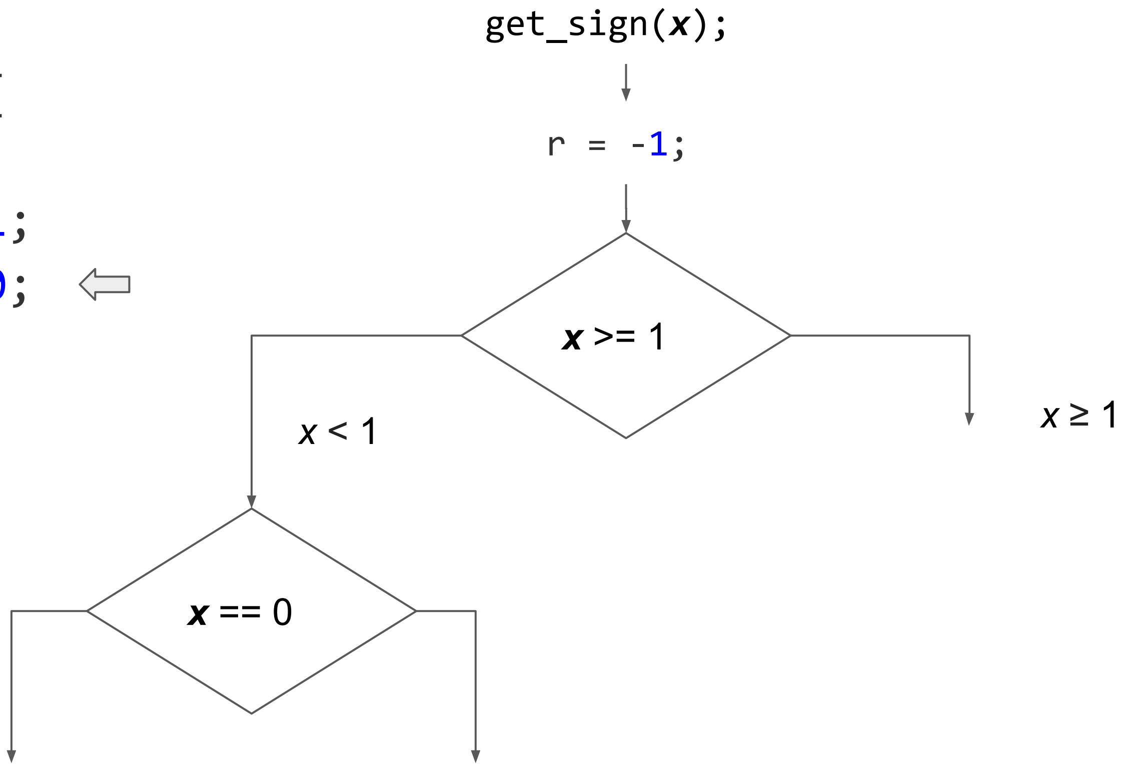


Known assignments

$x = -2$

$x = 7$

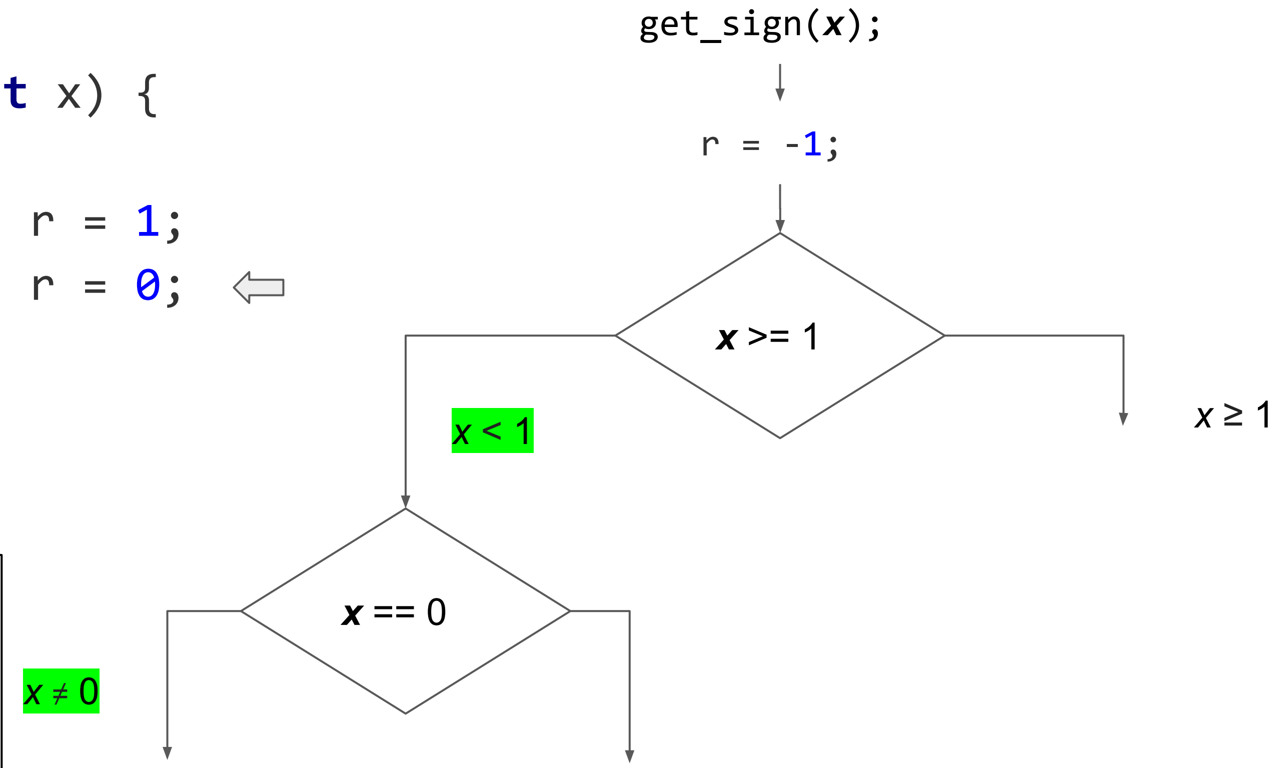
```
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2
x = 7

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

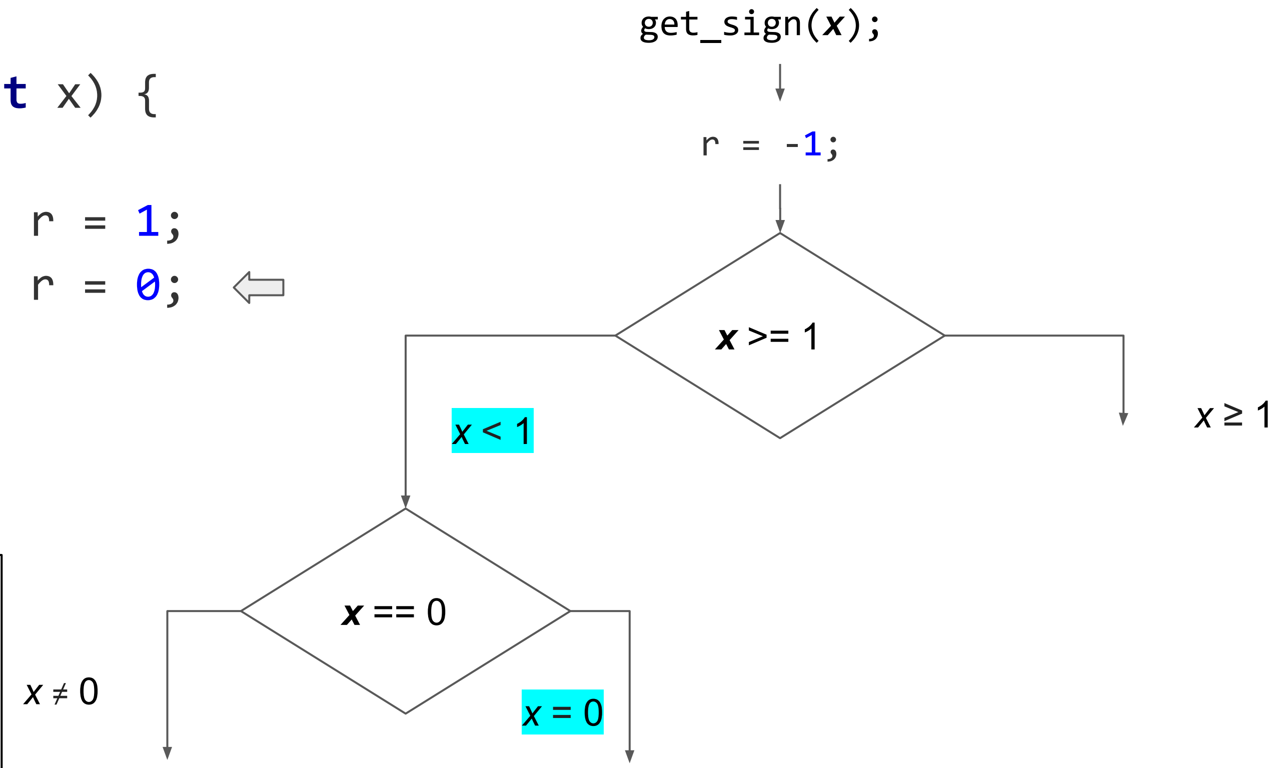


Known assignments

x = -2

x = 7

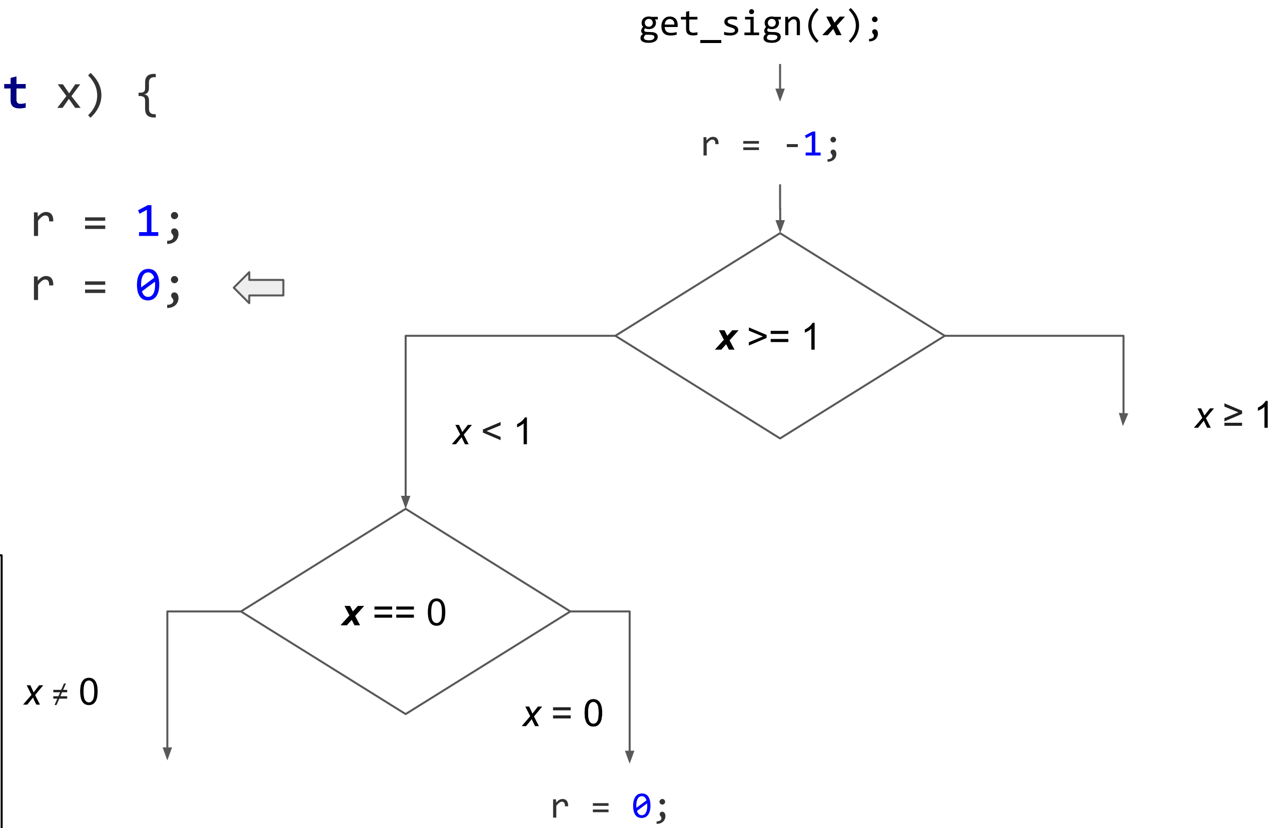
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2
x = 7
x = 0

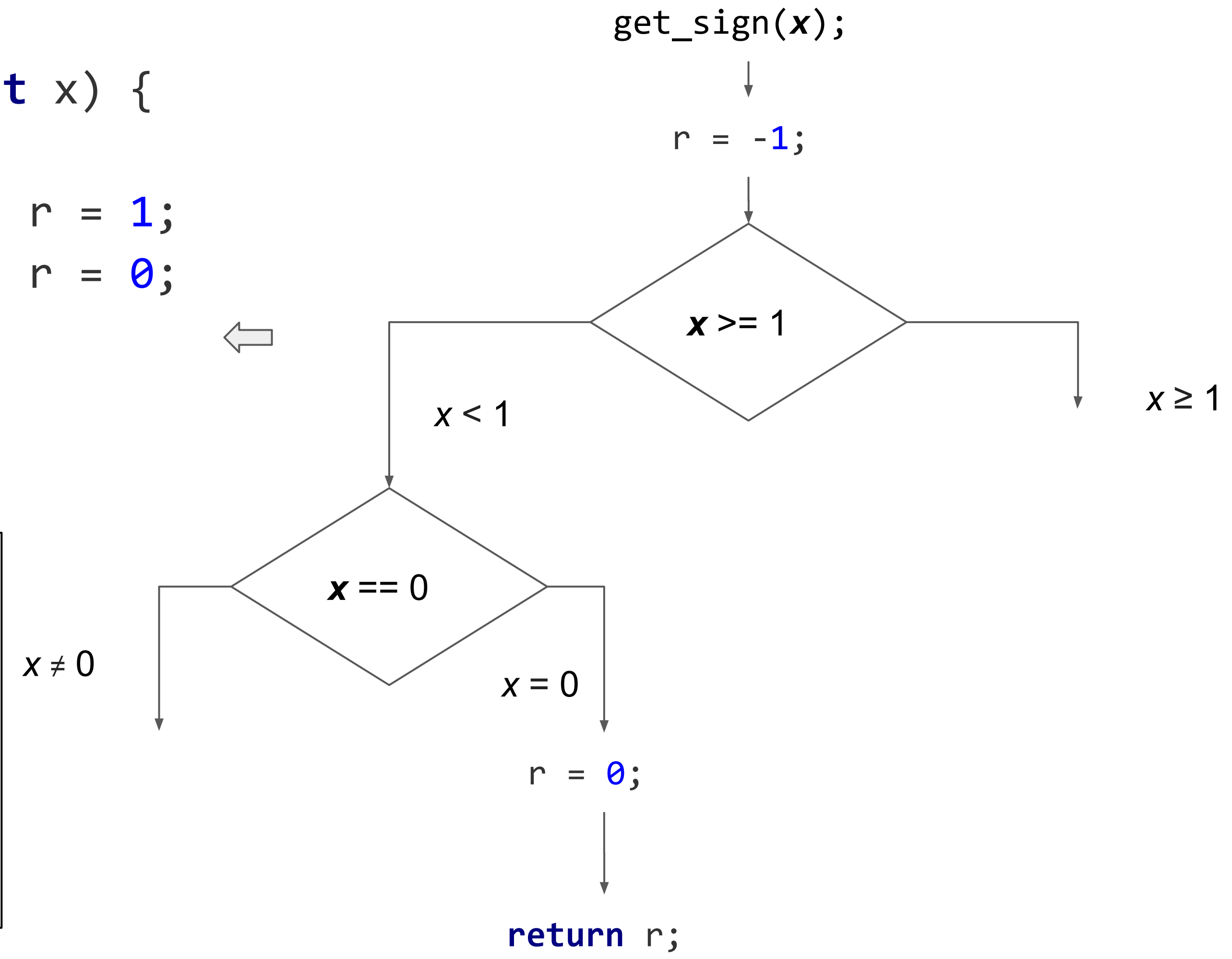
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2
x = 7
x = 0

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



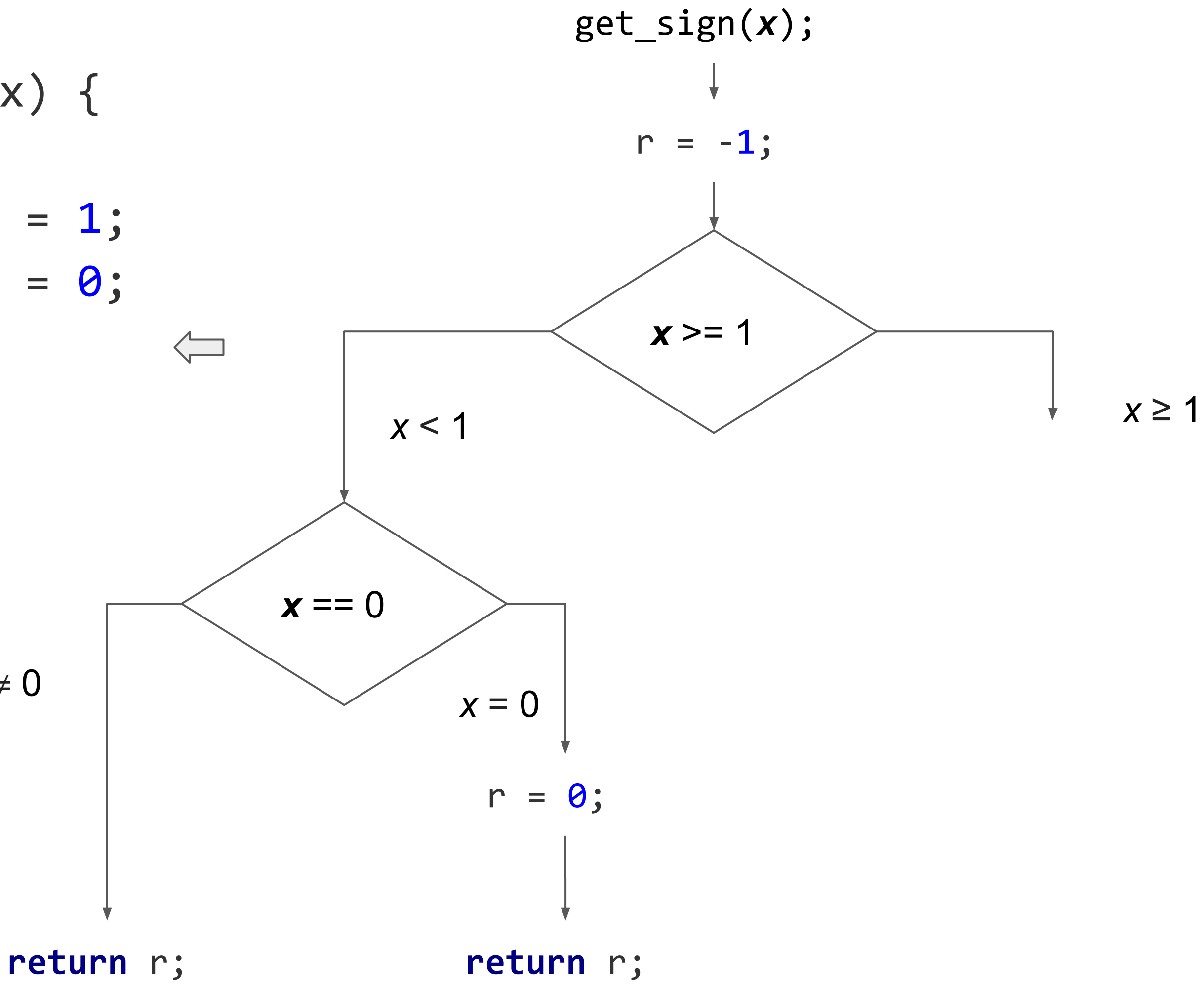
Known assignments

$\boldsymbol{x}$ = -2
$\boldsymbol{x}$ = 7
$\boldsymbol{x}$ = 0

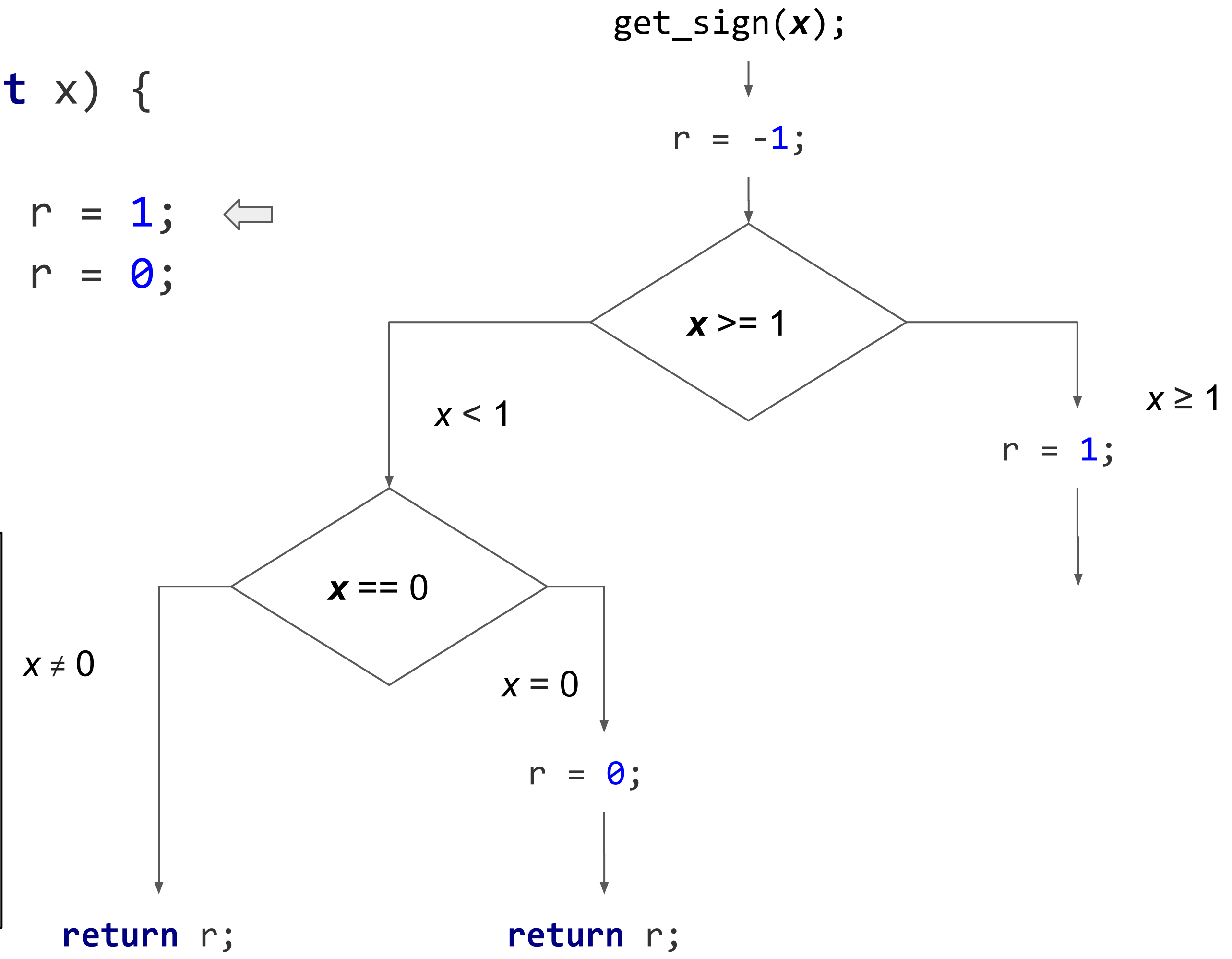
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments

$\boldsymbol{x} = -2$
$\boldsymbol{x} = 7$
$\boldsymbol{x} = 0$



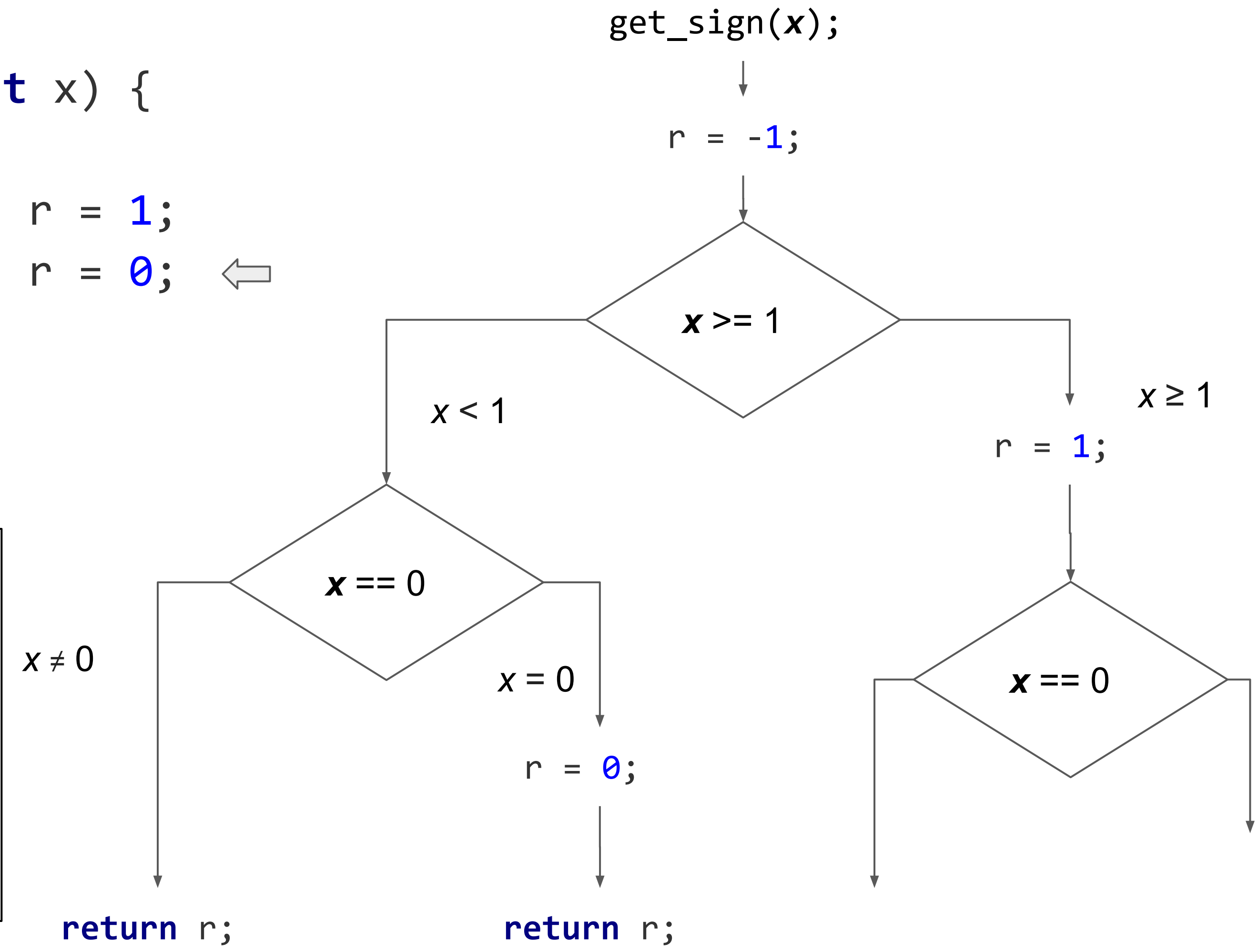
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2
x = 7
x = 0

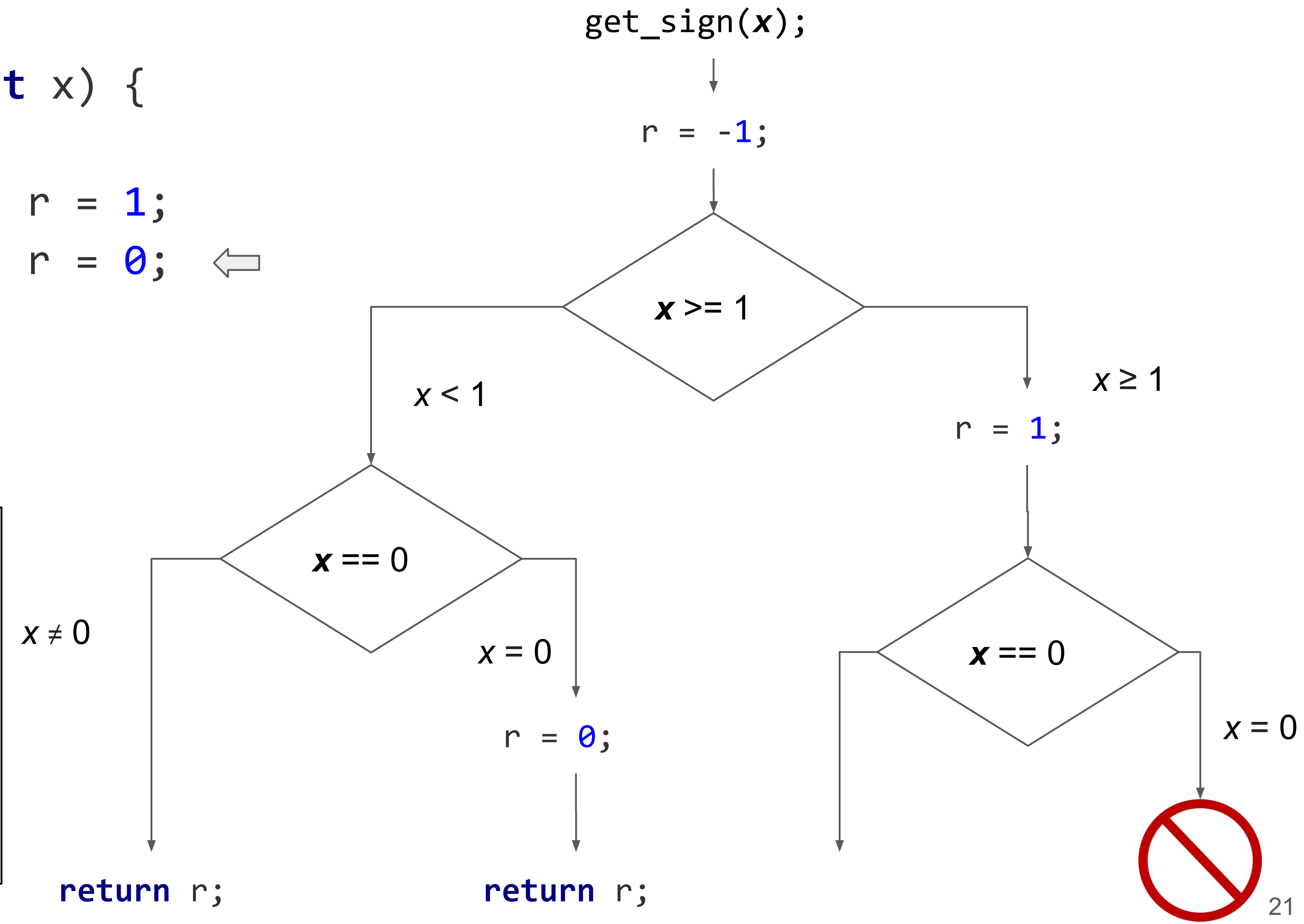
```
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2
x = 7
x = 0

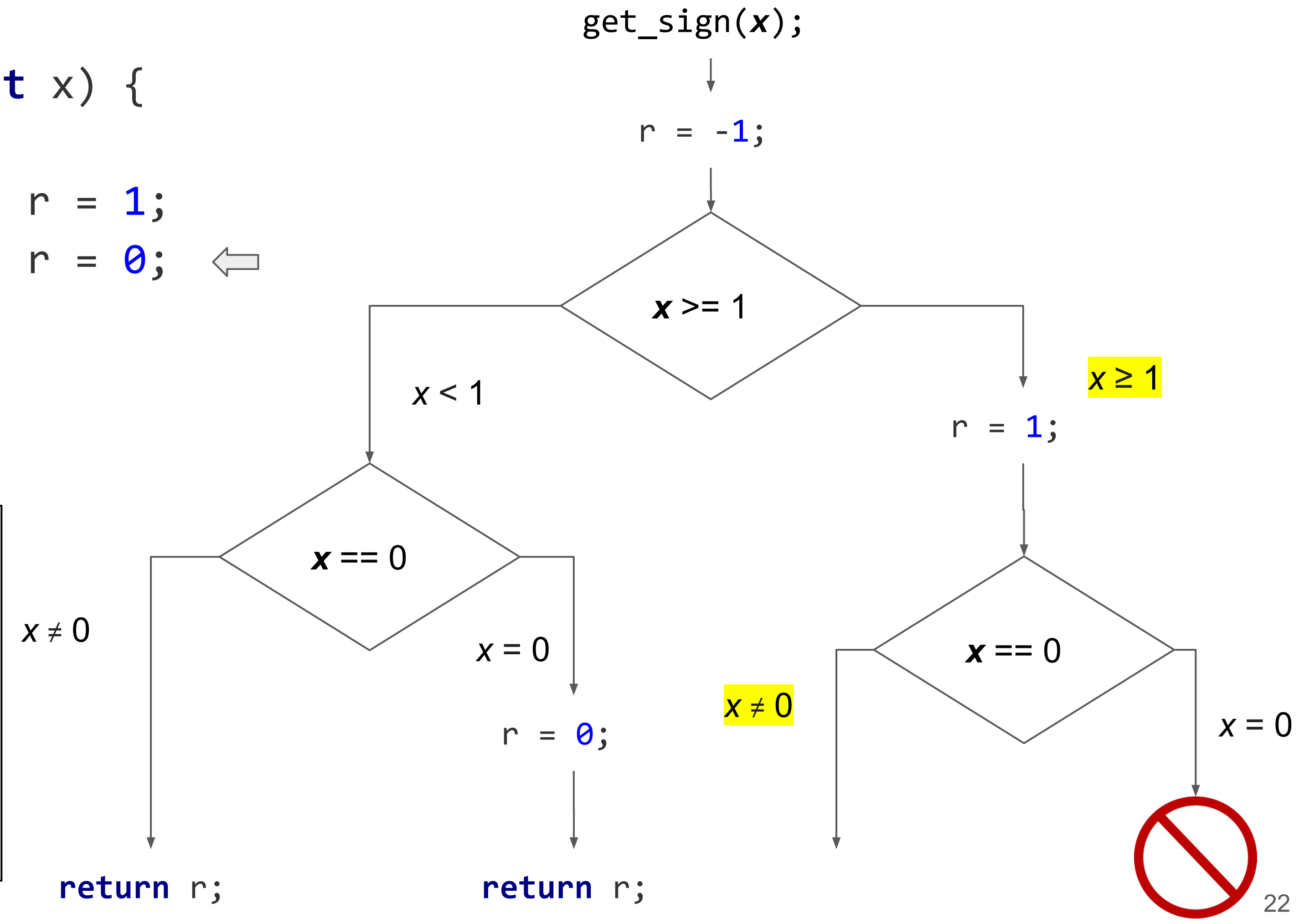
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2
x = 7
x = 0

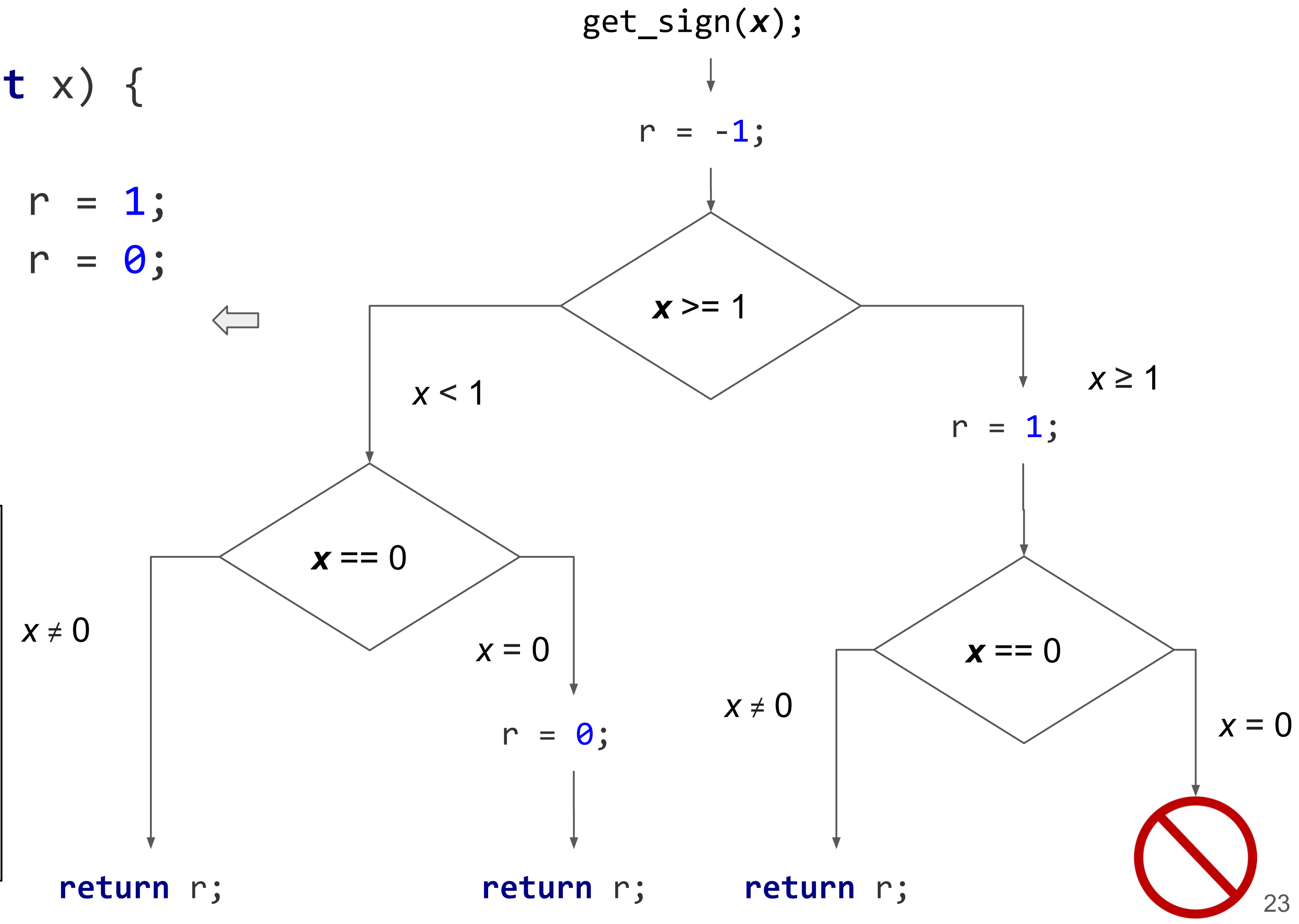
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2
x = 7
x = 0

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

$x = -2$
$x = 7$
$x = 0$

Symbolic execution with pending constraints

- Explore paths that are known to be feasible
- Solve constraints only when necessary to make progress

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

get_sign(***x***);

```
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments

$\varnothing$

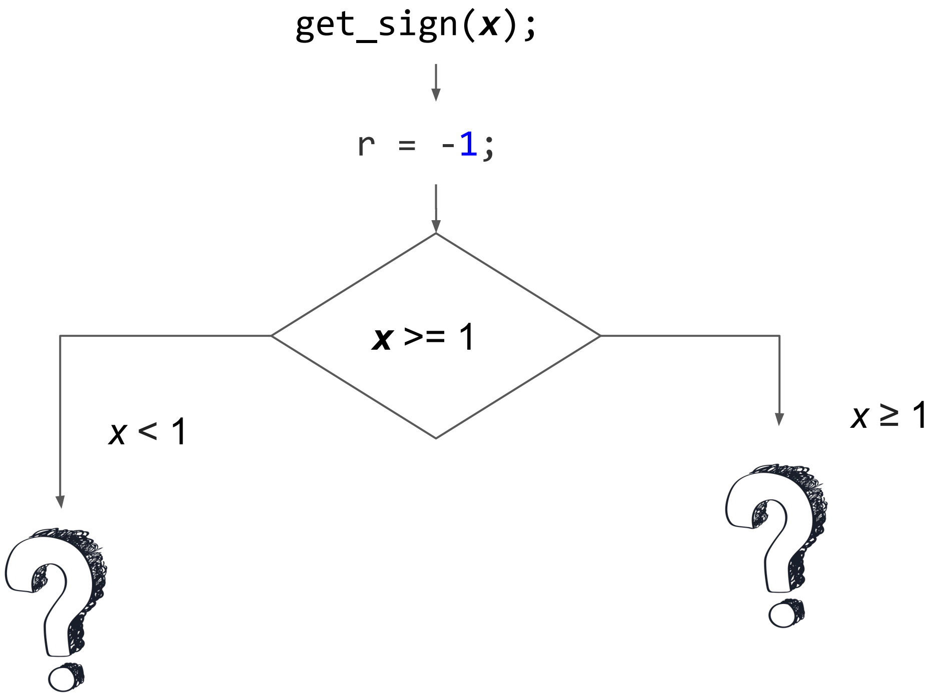
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

```c
get_sign(x);
r = -1;
```

Known assignments

∅

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

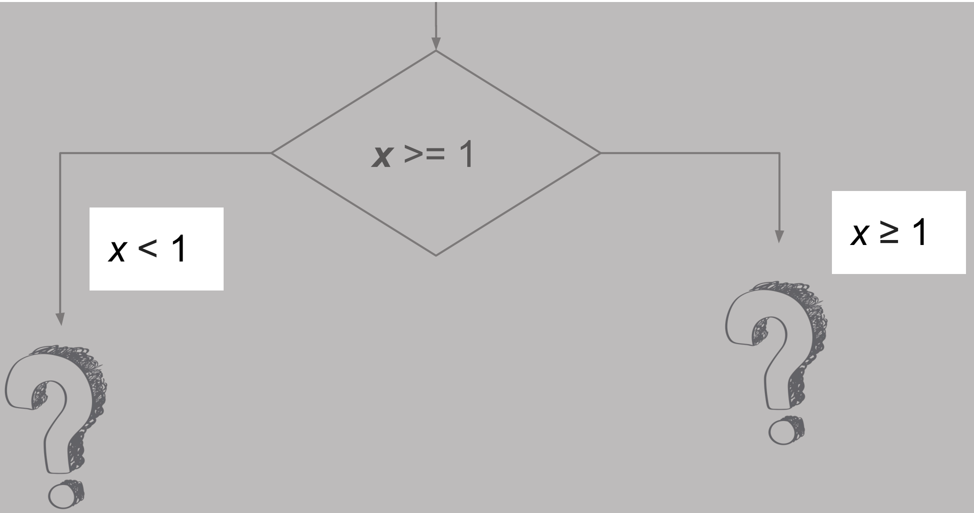
$\varnothing$

```
get_sign(x);
```

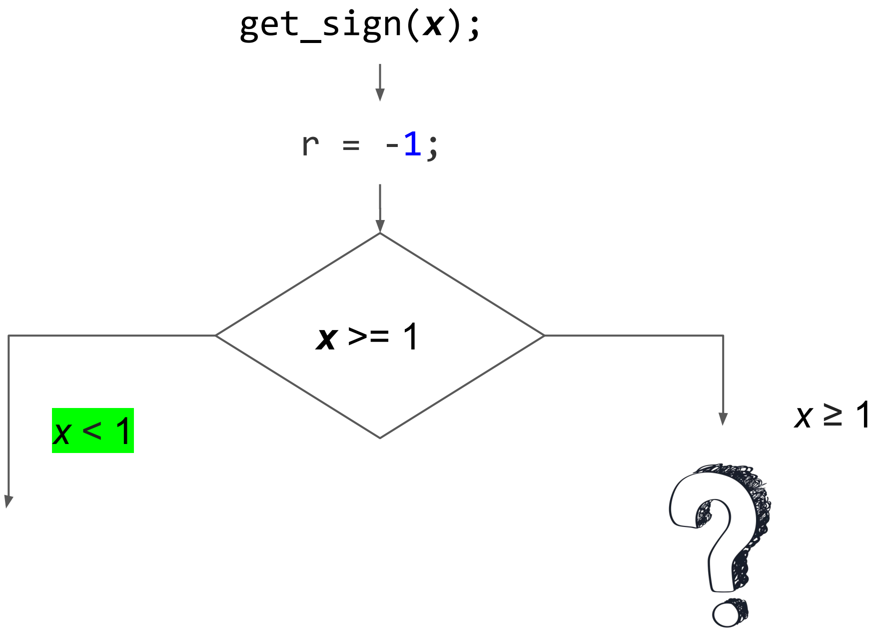
Pending constraints

```
if (x >= 1) r = 1;  ⇦
if (x == 0) r = 0;
```

- Not known to be feasible
- When only pending constraints left
    - Pick one and check



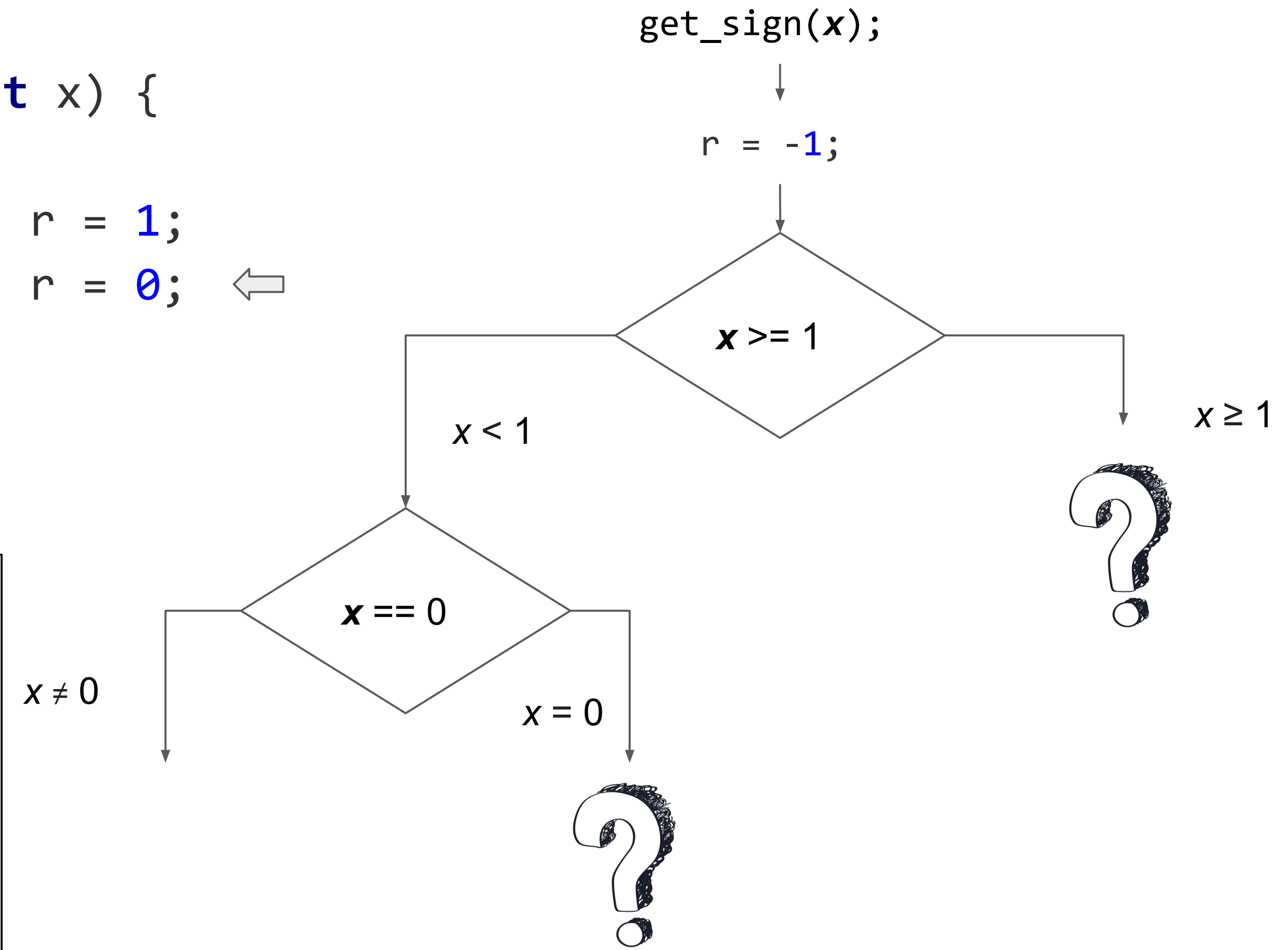
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2

```
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



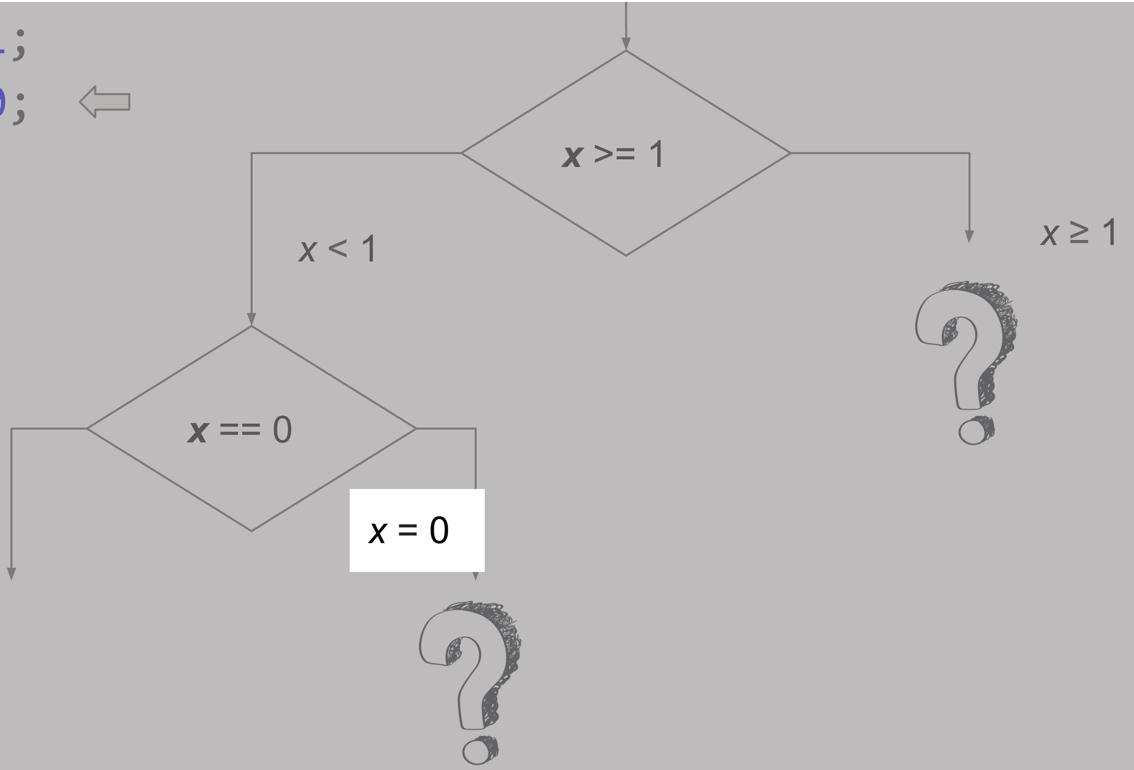
Known assignments

x = -2

`get_sign(x);`

Pending constraints: still not known if feasible

```
    if (x >= 1) r = 1;
    if (x == 0) r = 0;  ⇦
    return r;
}
```



Known assignments

x = -2

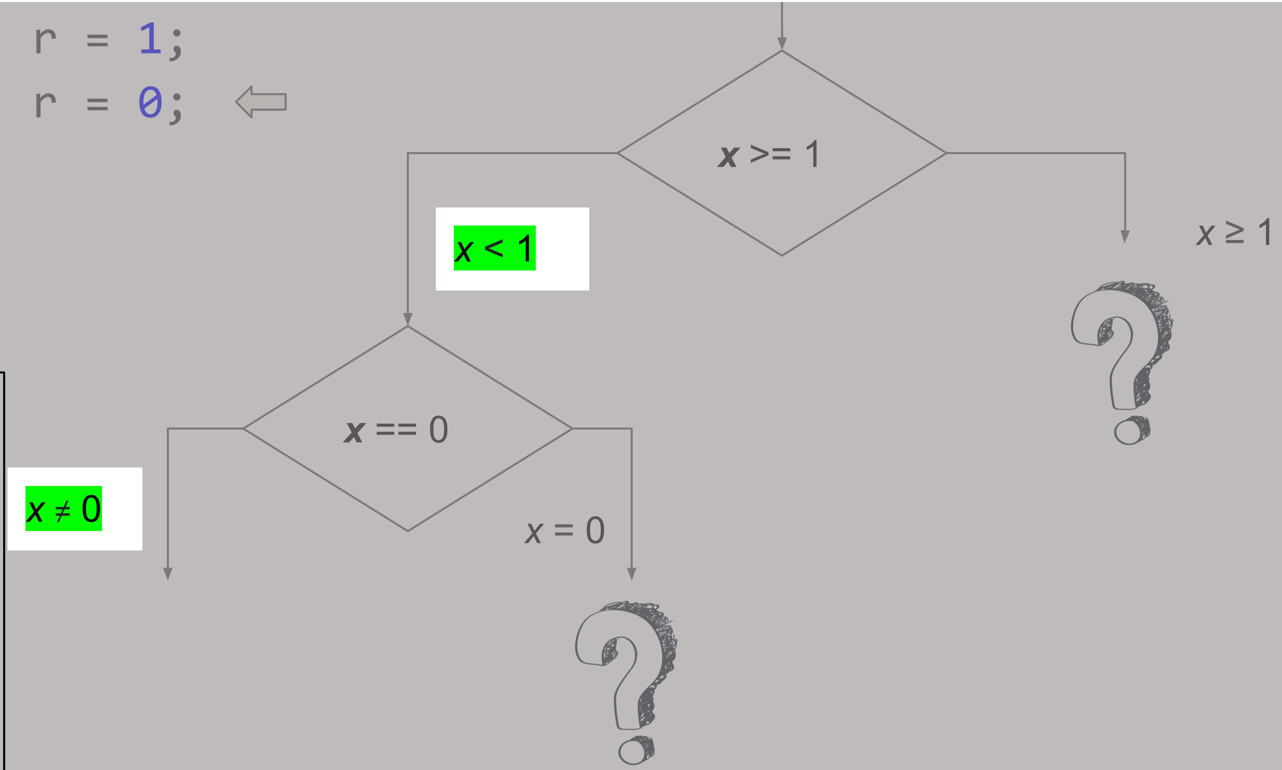
get_sign(x);

Pending constraints: known feasible path!

```
    if (x >= 1) r = 1;
    if (x == 0) r = 0;  ⇦
    return r;
}
```

Known assignments

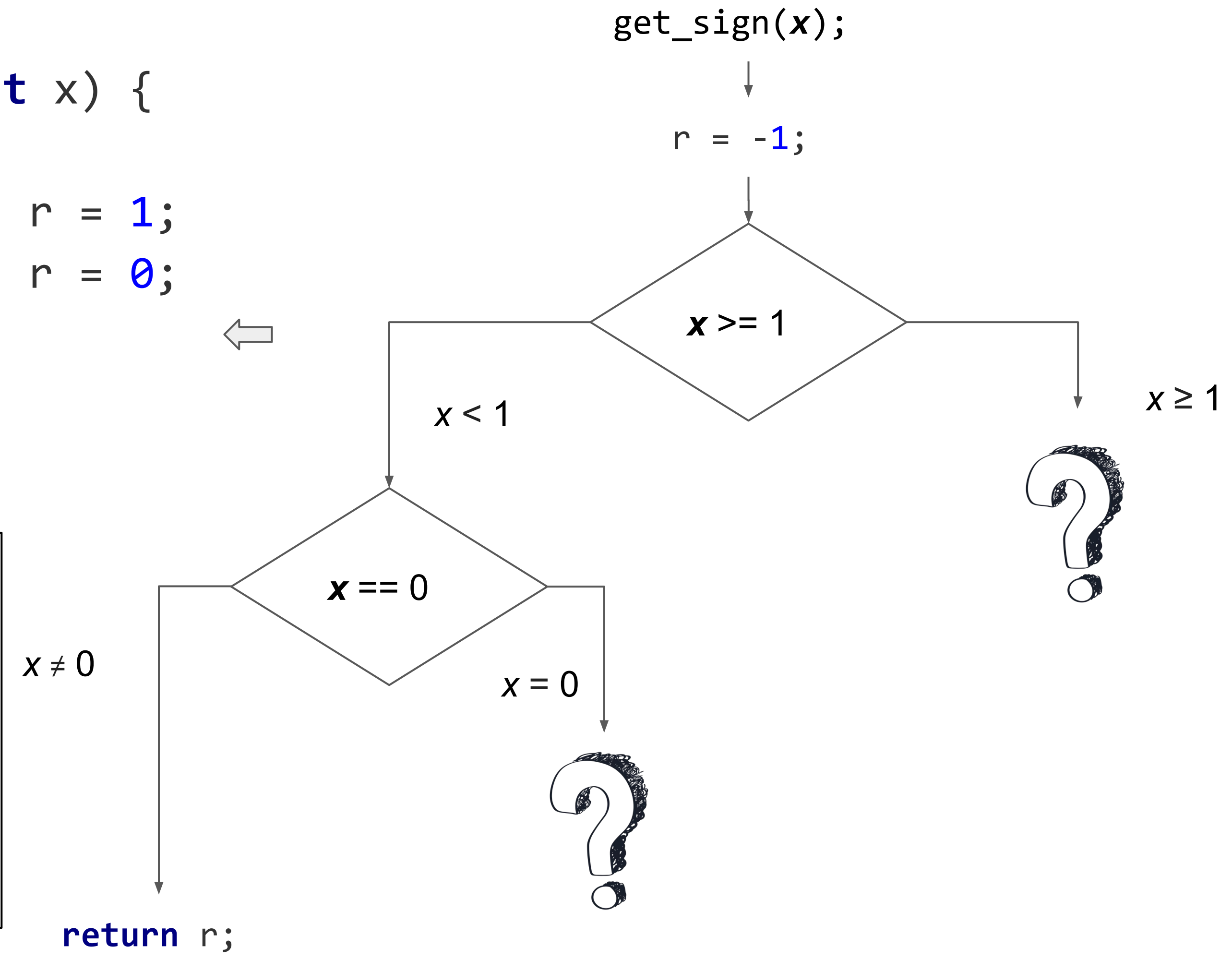
$x = -2$



```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments

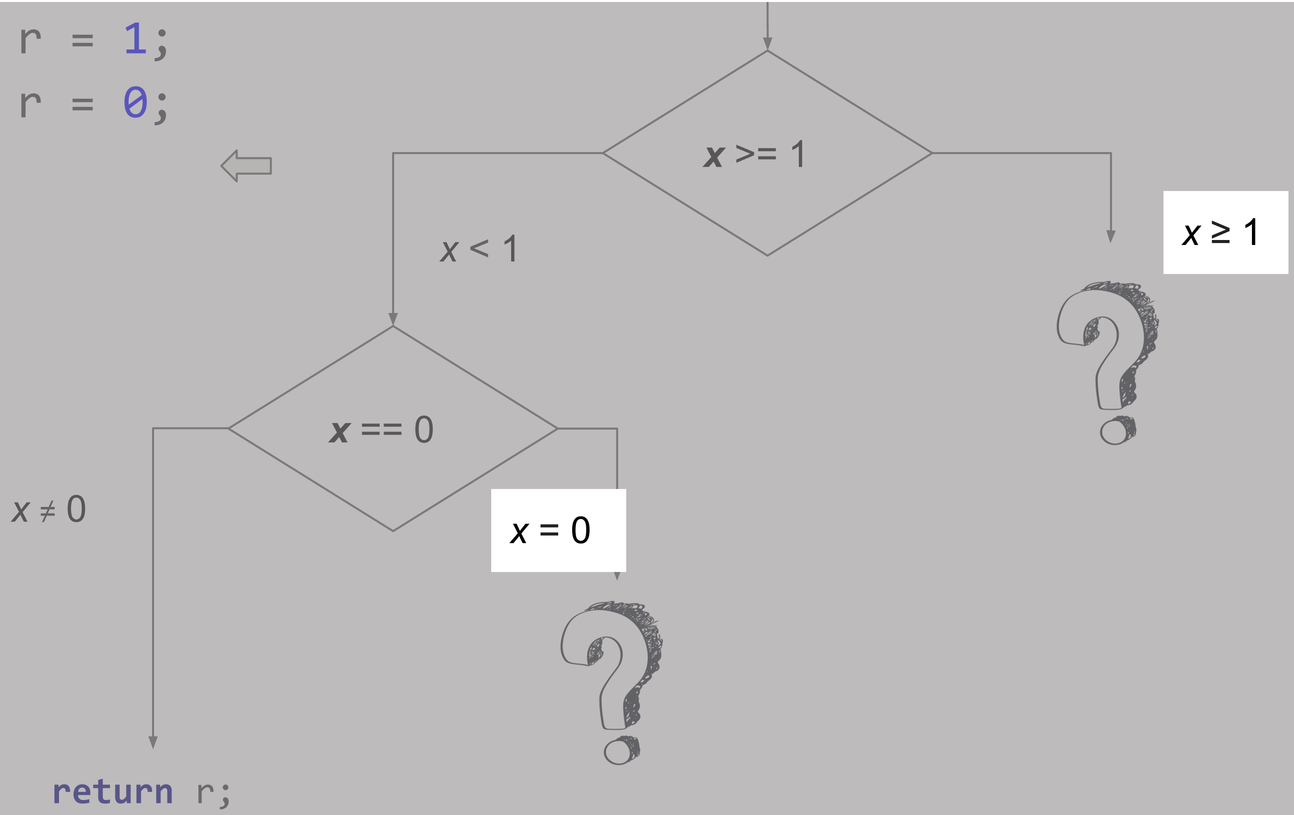
$x = -2$



Only pending constraints: check one

```
get_sign(x);
```

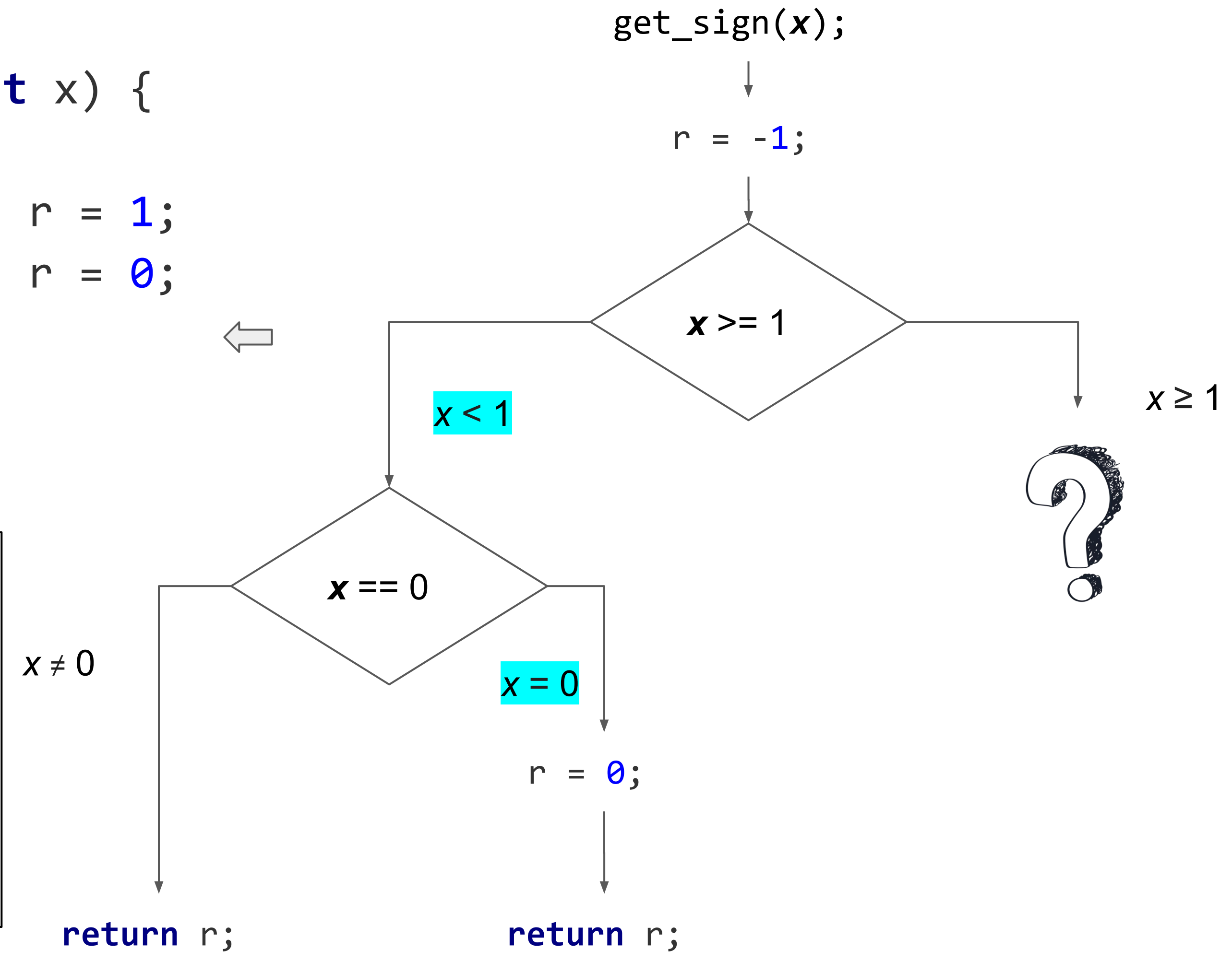
```
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

$x = -2$

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

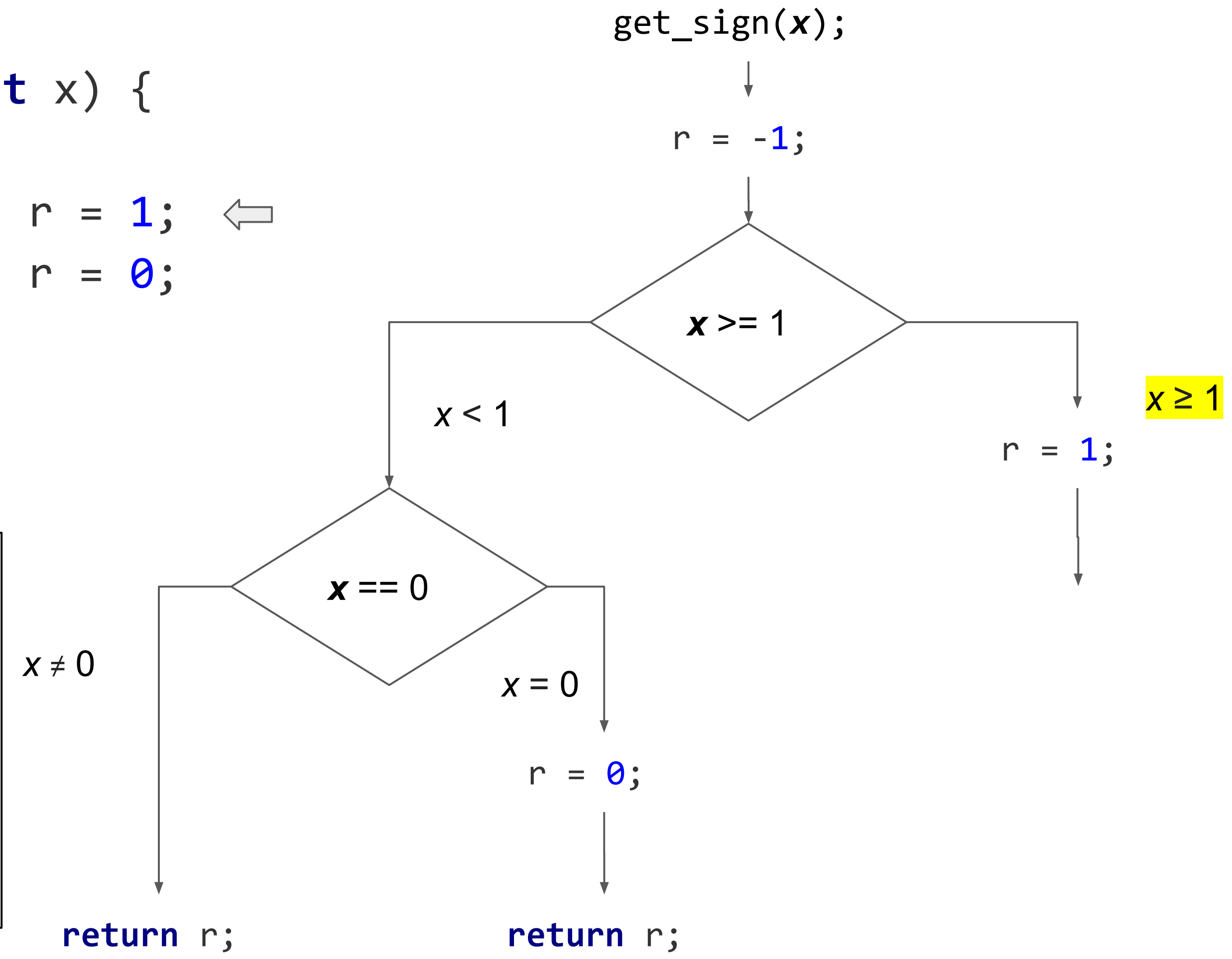


Known assignments

$x = -2$

$x = 0$

```
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



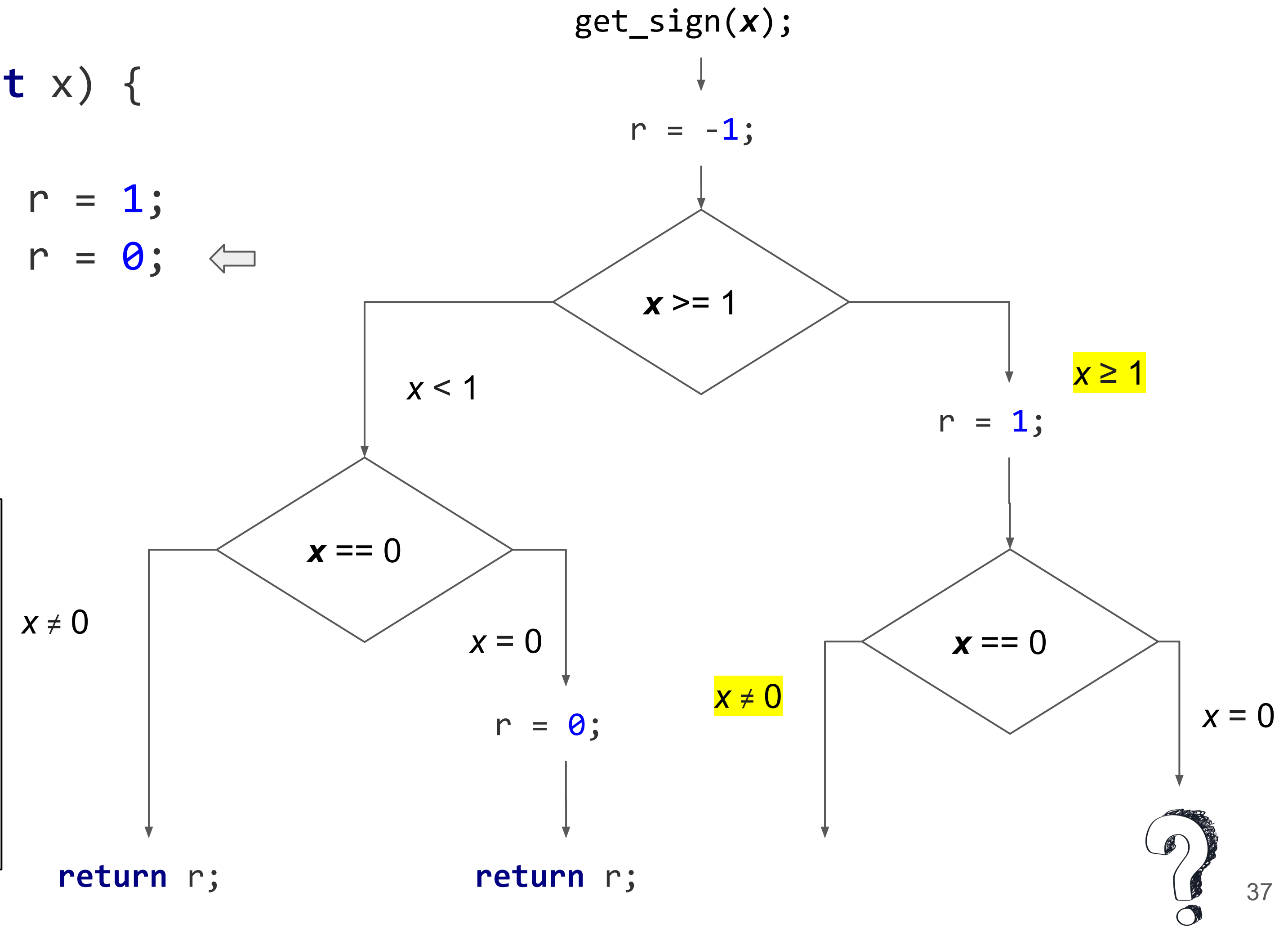
Known assignments

$x = -2$
$x = 0$
$x = 7$

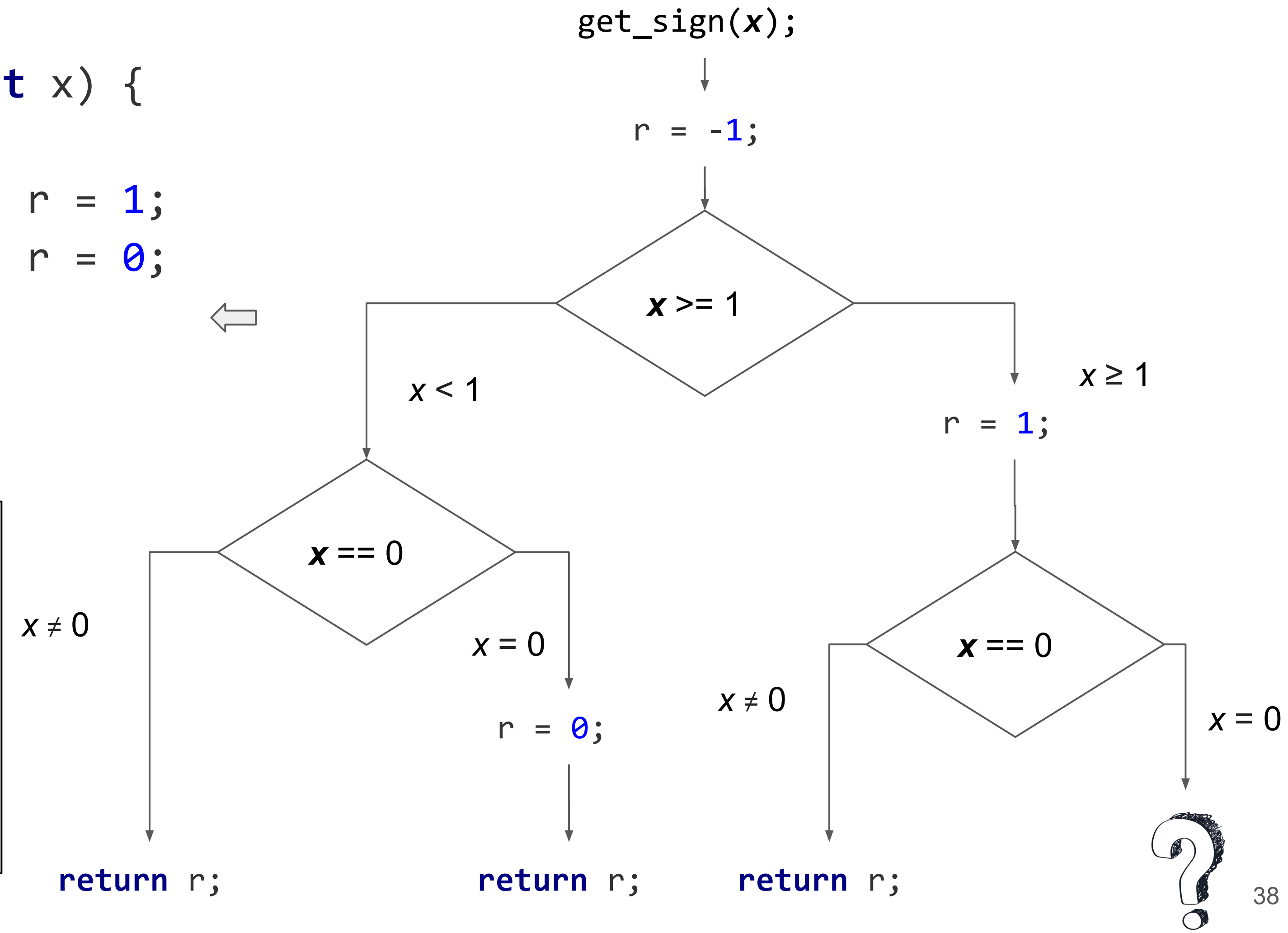
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments

x = -2
x = 0
x = 7



```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



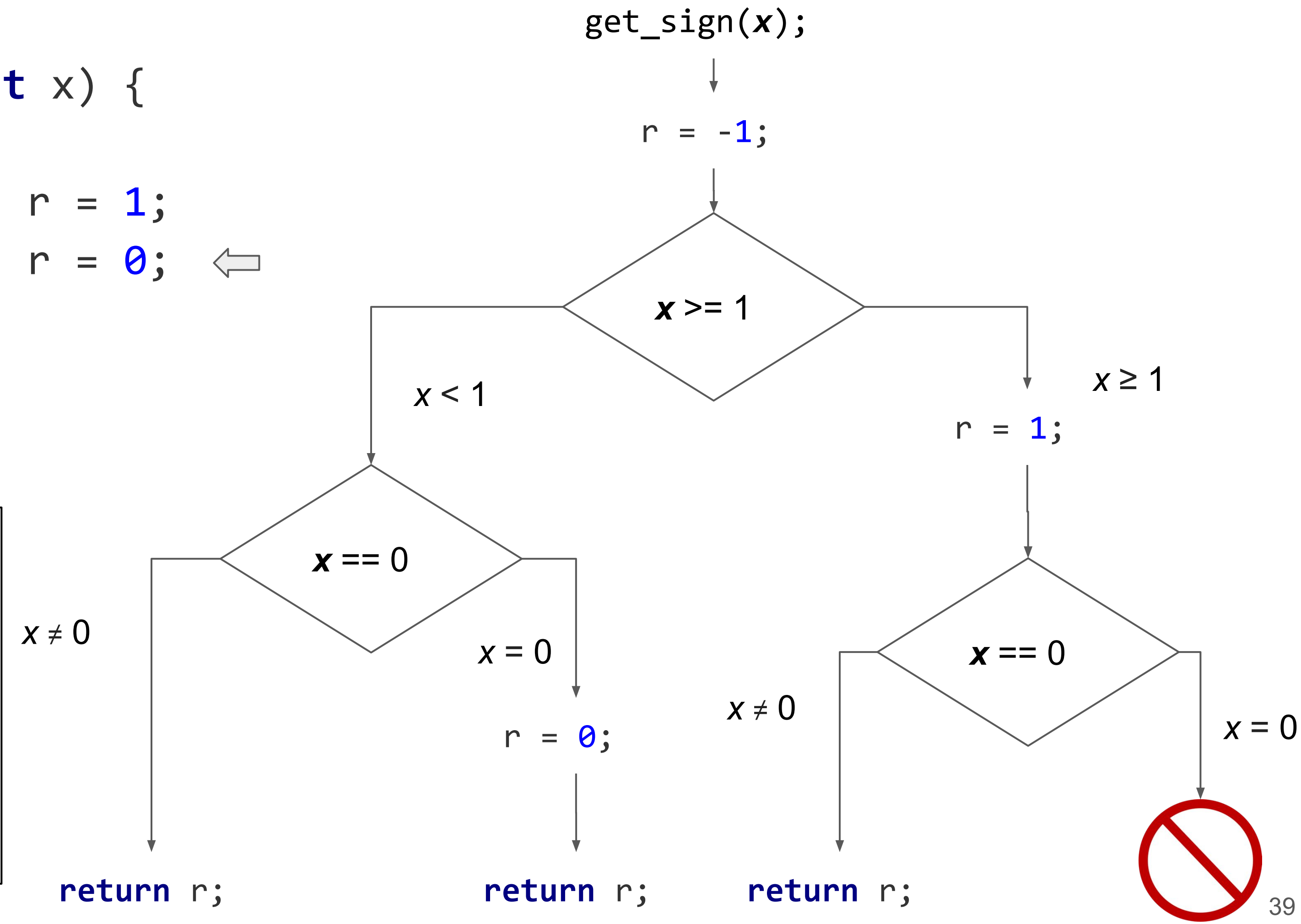
Known assignments

$\boldsymbol{x}$ = -2
$\boldsymbol{x}$ = 0
$\boldsymbol{x}$ = 7

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments

$x = -2$
$x = 0$
$x = 7$

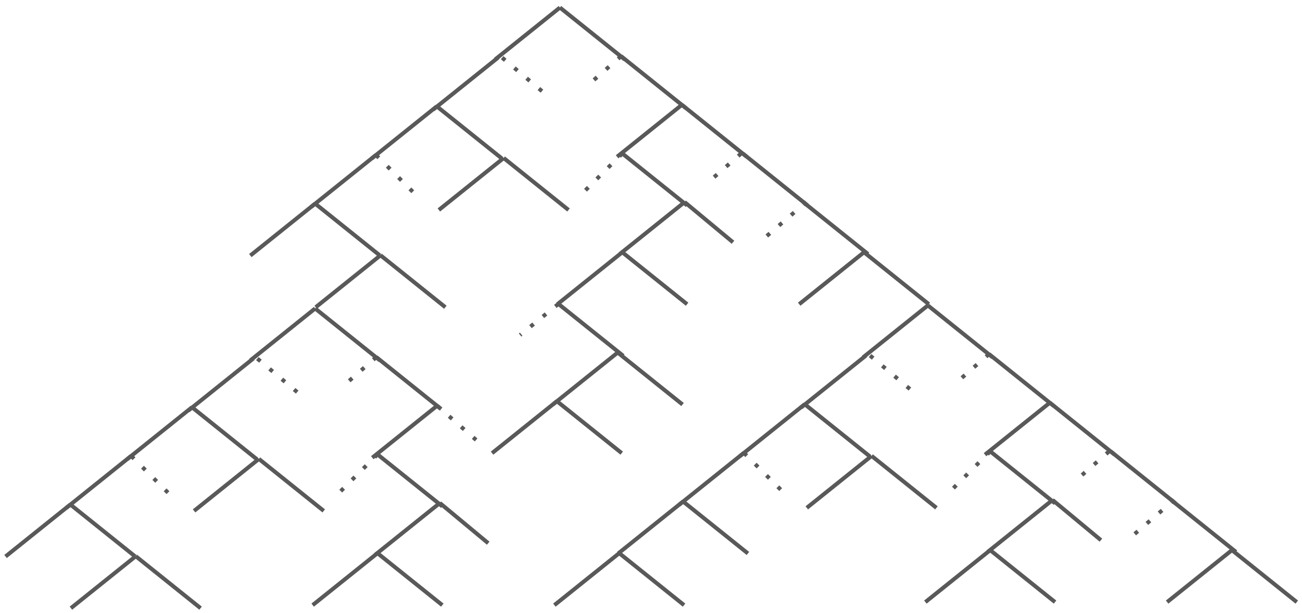


Pending constraints: why would they be useful?

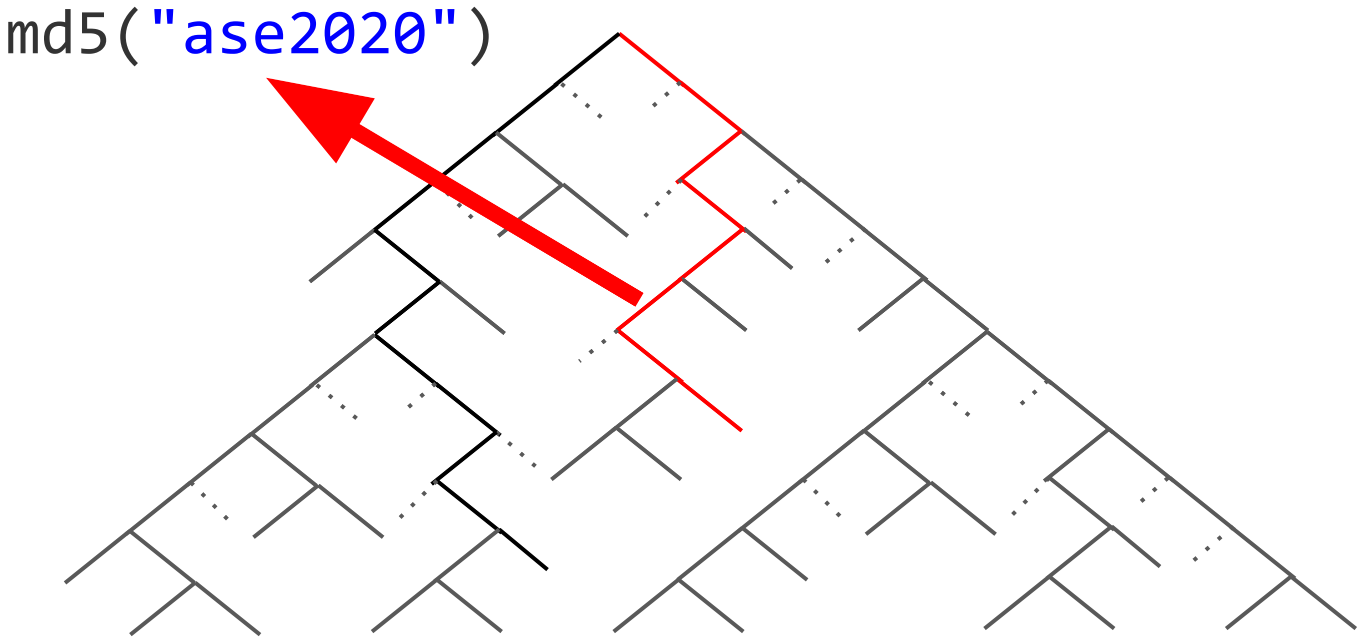
- Reversing md5 hash
  - Very hard for SMT solvers
- `1471037522 = md5("ase2020")[0]`
  - Use as seed

```
char msg[8] = symbolic;
uint32_t *hash = md5(msg, 8);
assert(hash[0] == 1471037522)
```

Suppose this exploration tree for `md5`

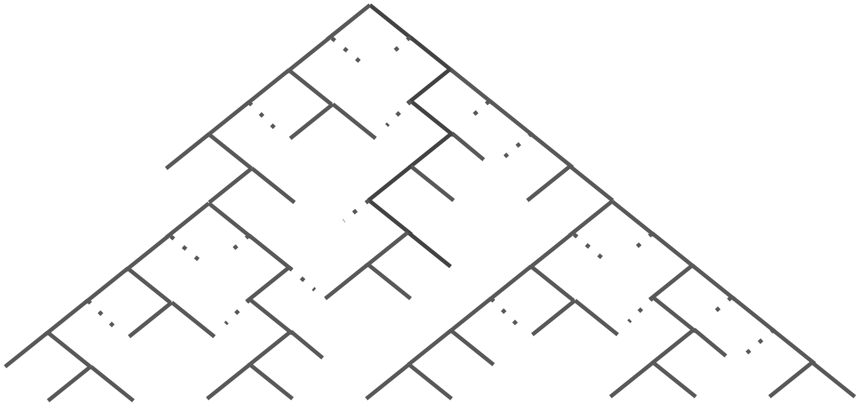


Suppose this path

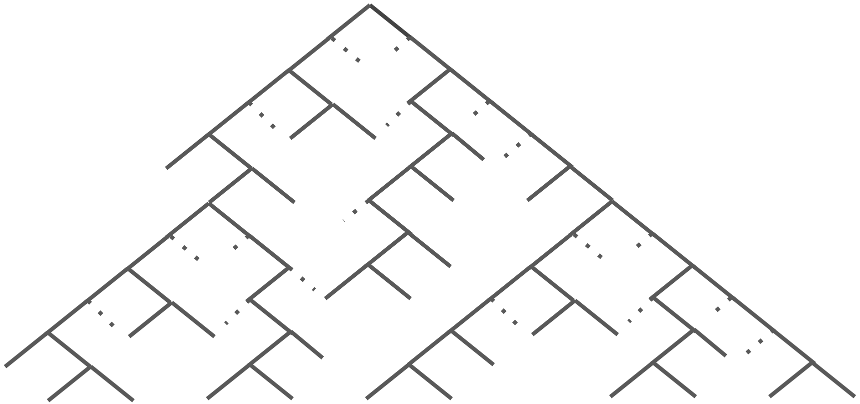


Solver queries: 0

Pending

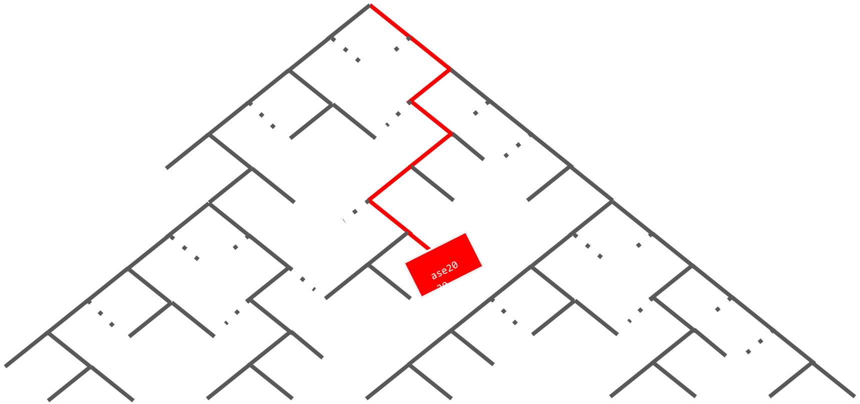


Vanilla

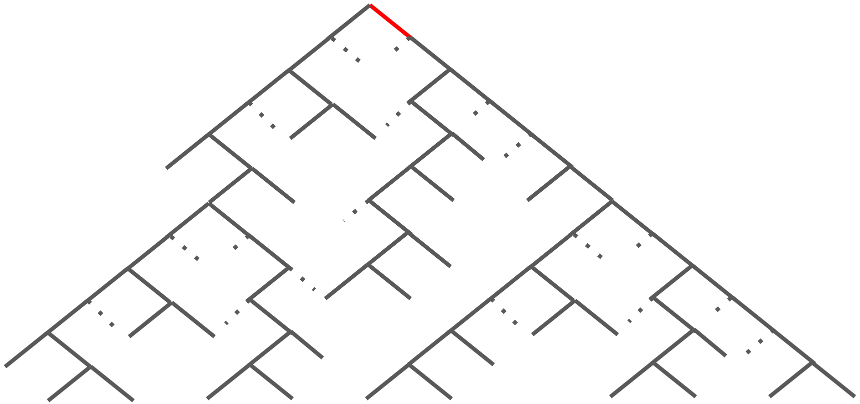


Solver queries: 0

Pending

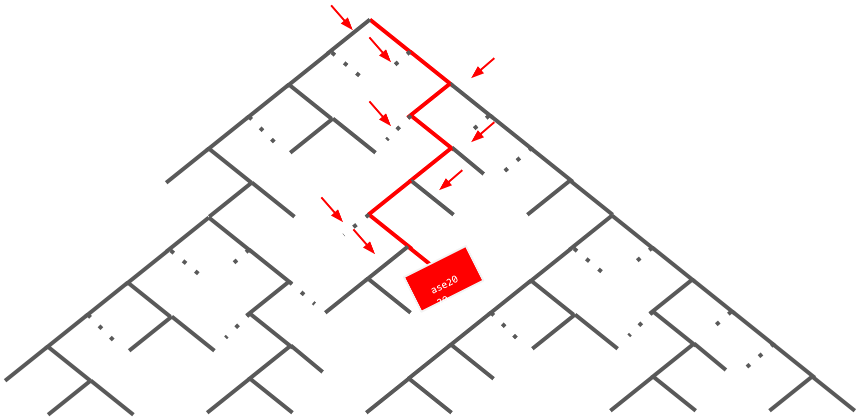


Vanilla

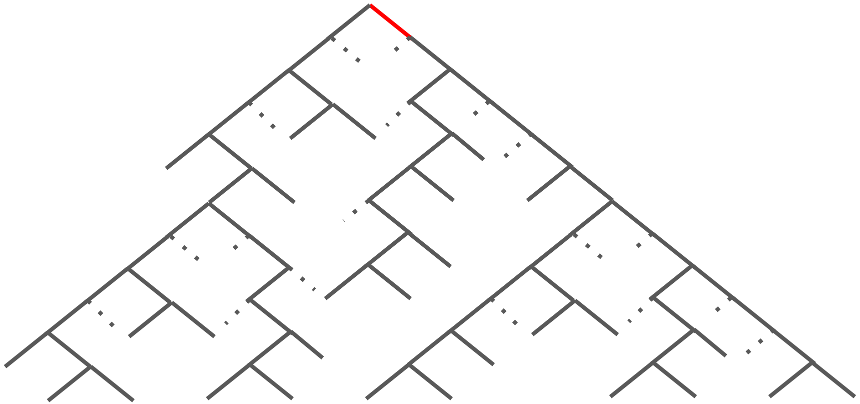


Solver queries: 0

Pending

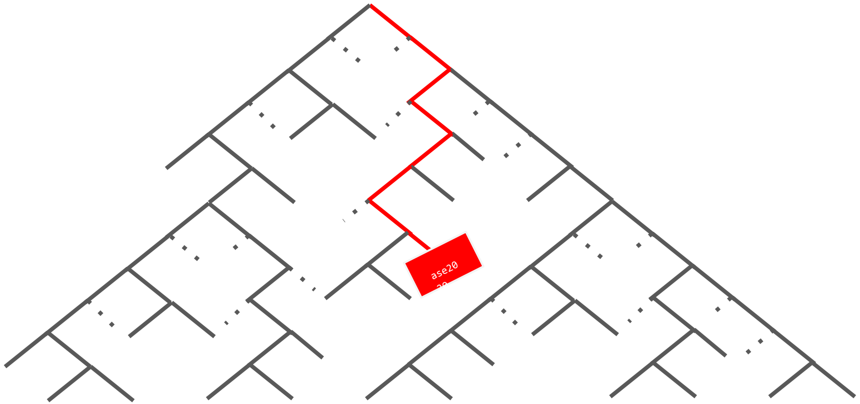


Vanilla

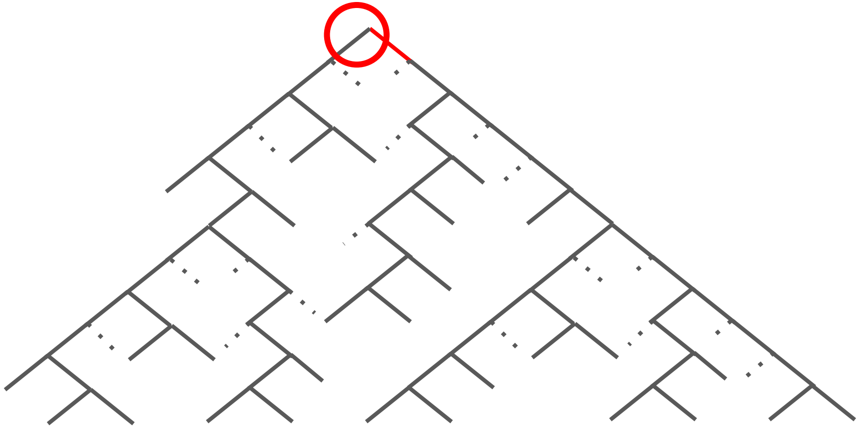


Solver queries: 0

Pending



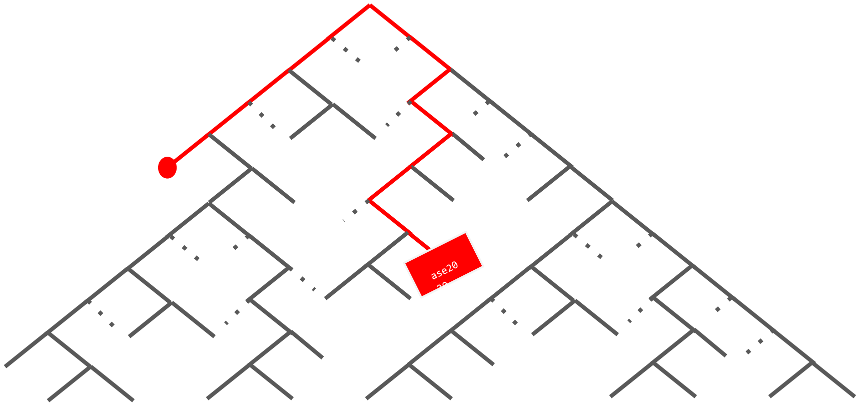
Vanilla

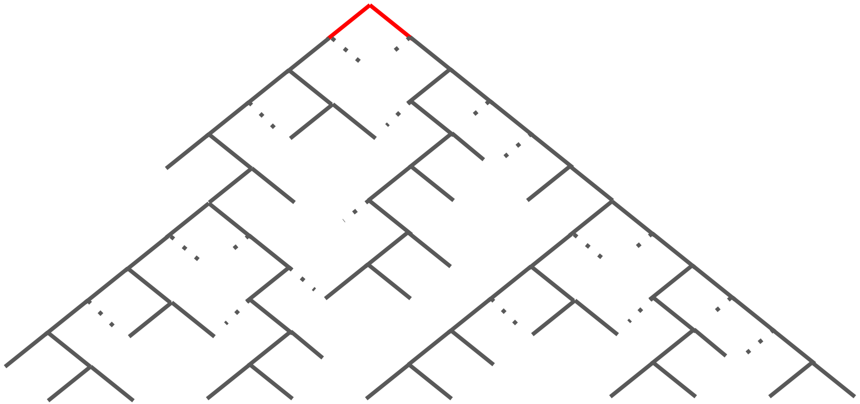


Solver queries: 1

Pending

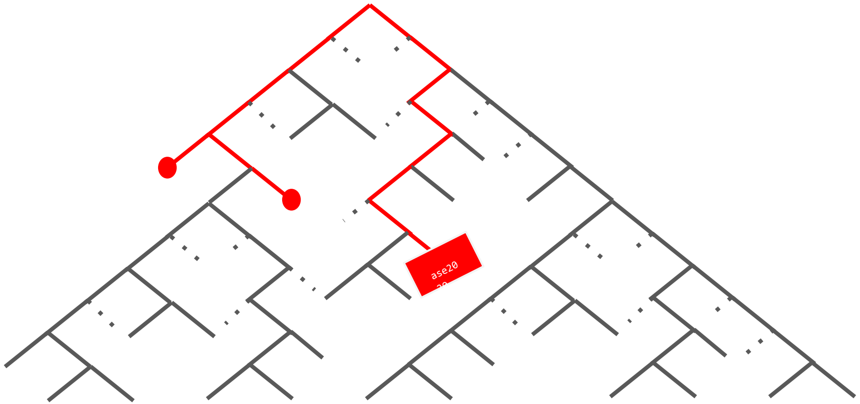
Vanilla



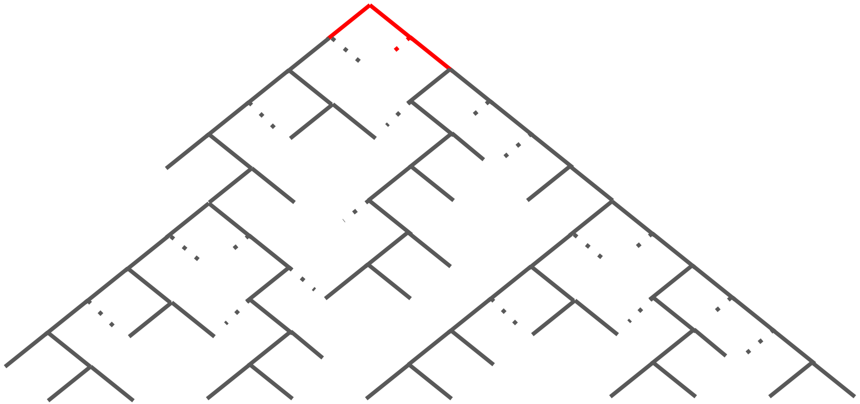


Solver queries: 2

Pending

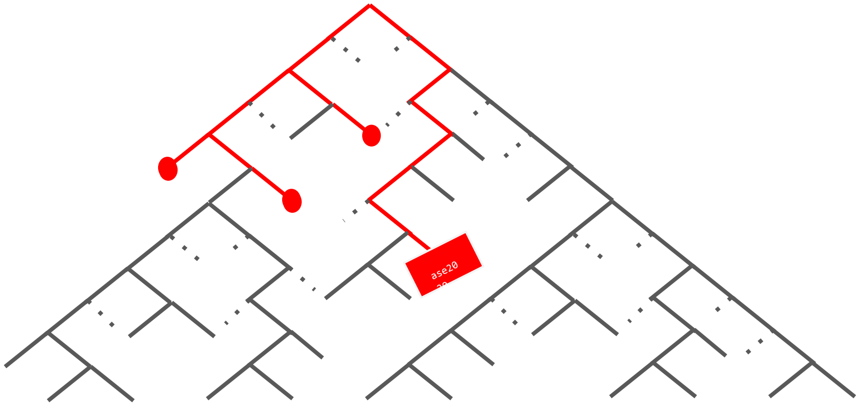


Vanilla

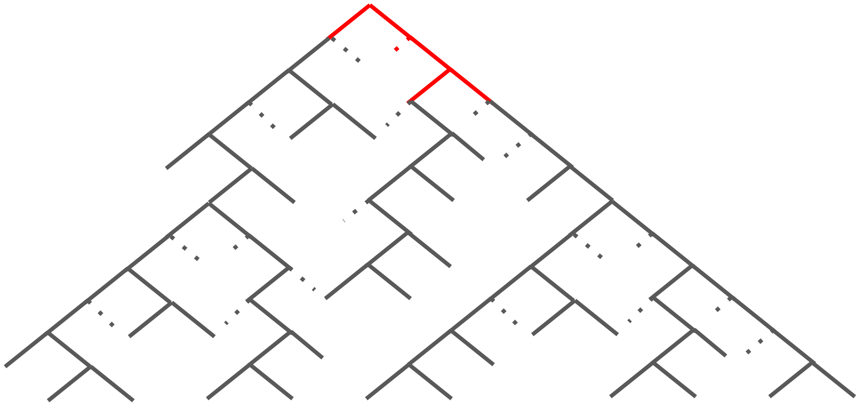


Solver queries: 3

Pending

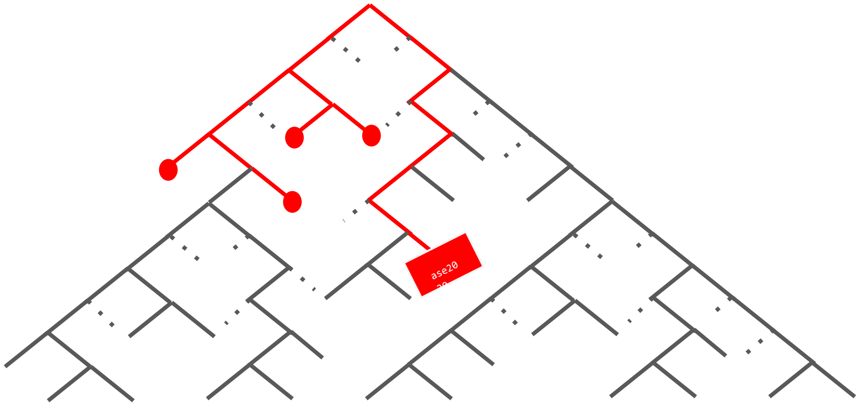


Vanilla

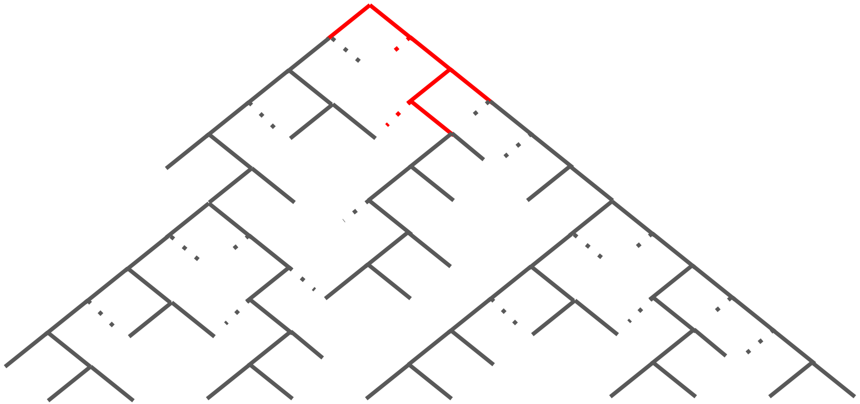


Solver queries: 4

Pending

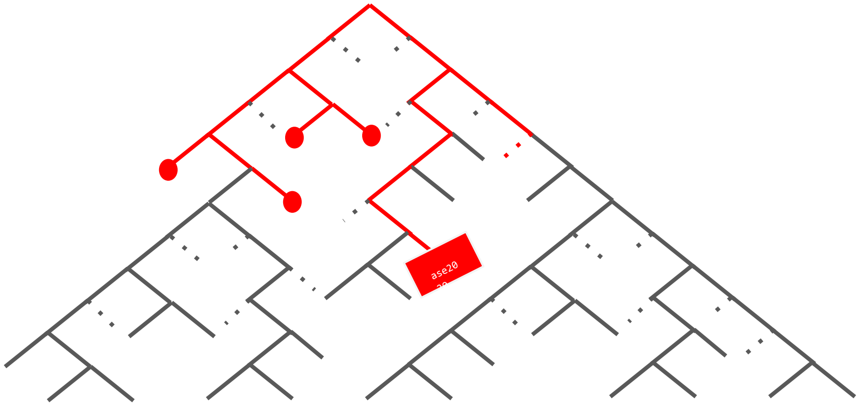


Vanilla

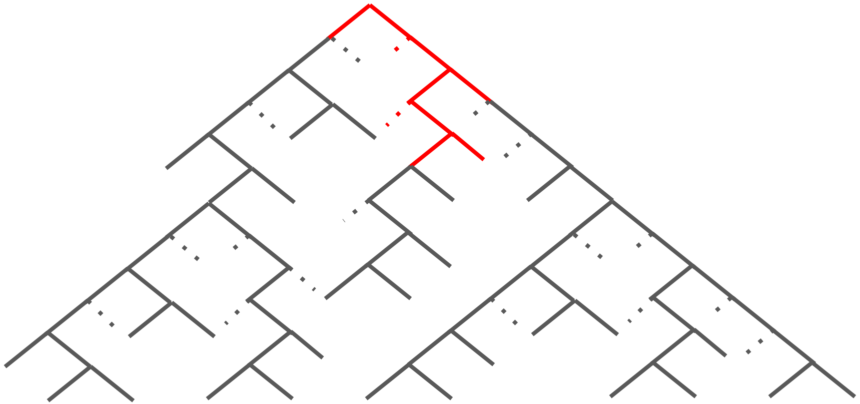


Solver queries: 5

Pending

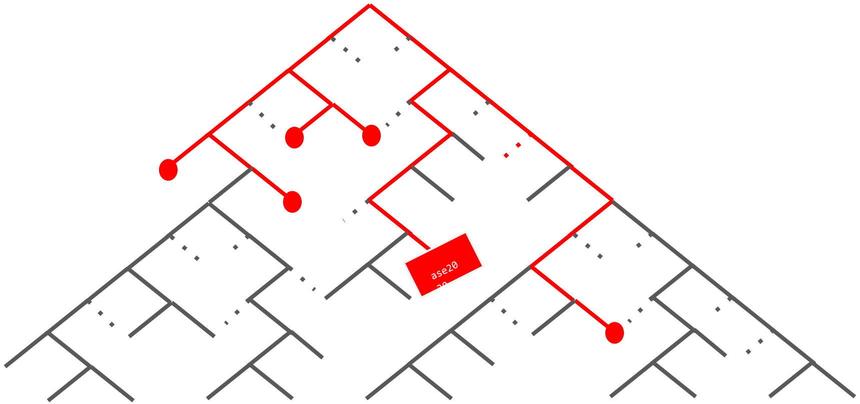


Vanilla

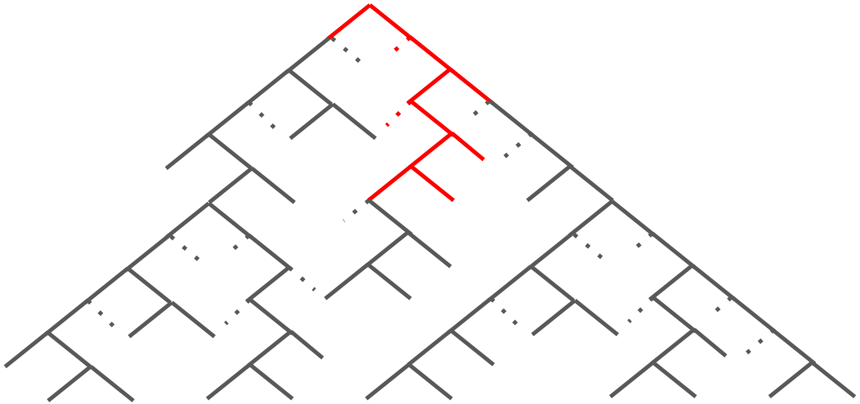


Solver queries: 6

Pending

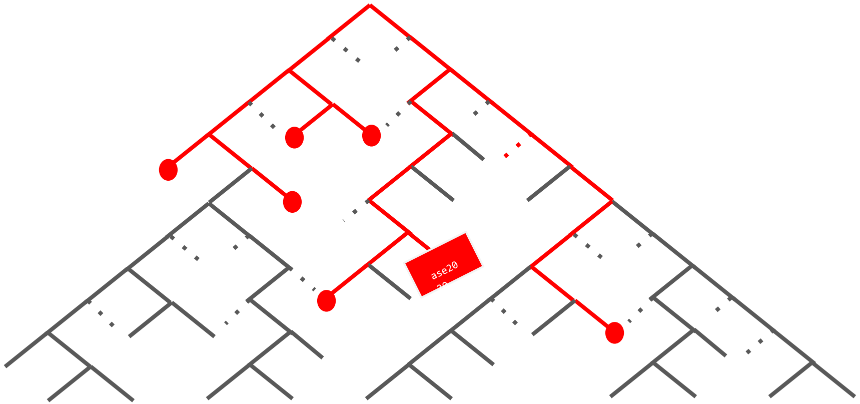


Vanilla

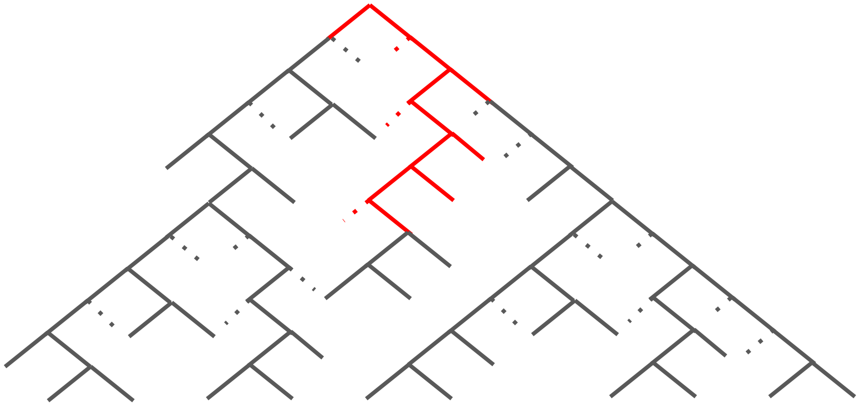


Solver queries: 7

Pending

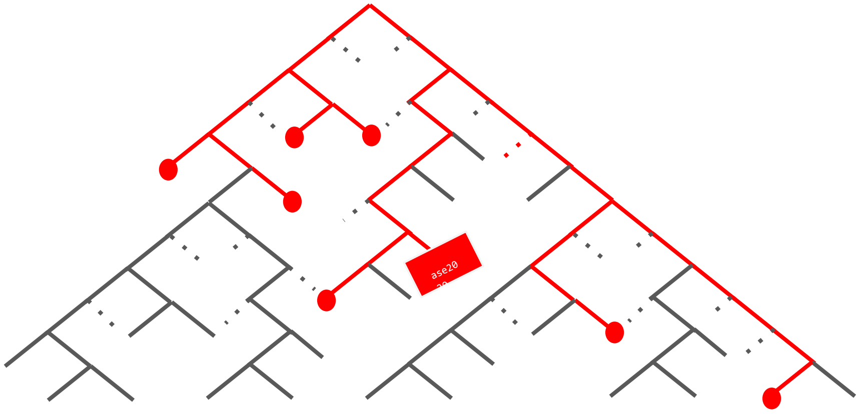


Vanilla

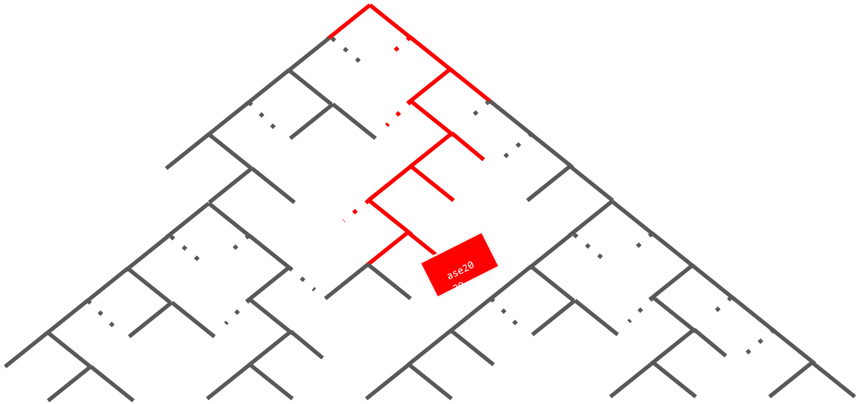


Solver queries: 8

Pending

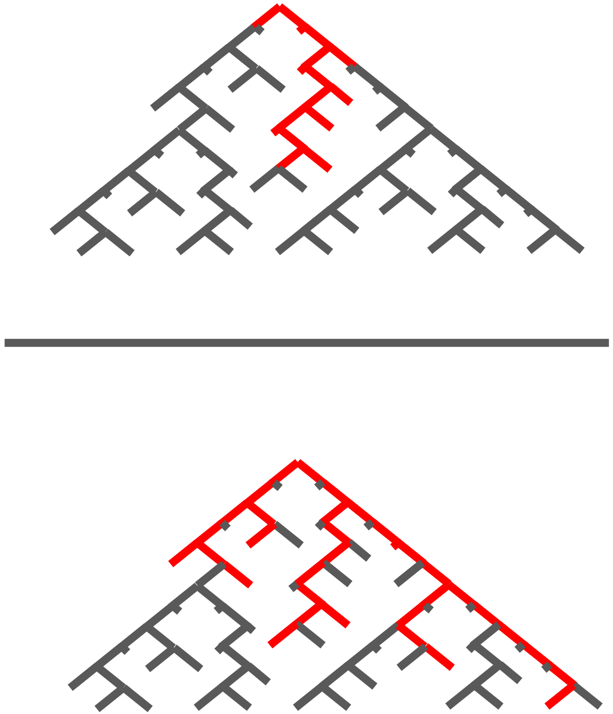


Vanilla



Why pending constraints?

- More efficient use of solver solutions
  - Explore more instructions per query
  - Less time solving infeasible queries
- Prefers deeper search tree exploration
- Empowering search heuristics
  - Control over constraint solving
  - ZESTI



Evaluation

- Based on an implementation in KLEE
- 8 real world applications
- Hard targets for symbolic execution

ImageMagick

bc

m4

make

datamash

Ogg Vorbis

SQLite

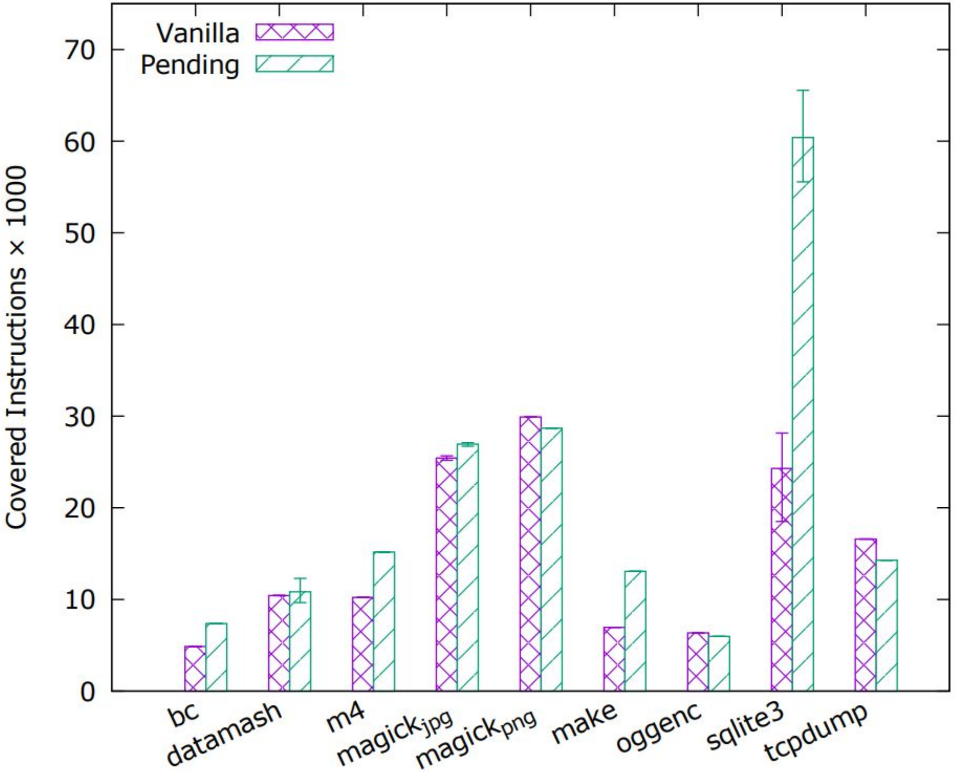
TCPDUMP & LiBPCAP

Experiment design

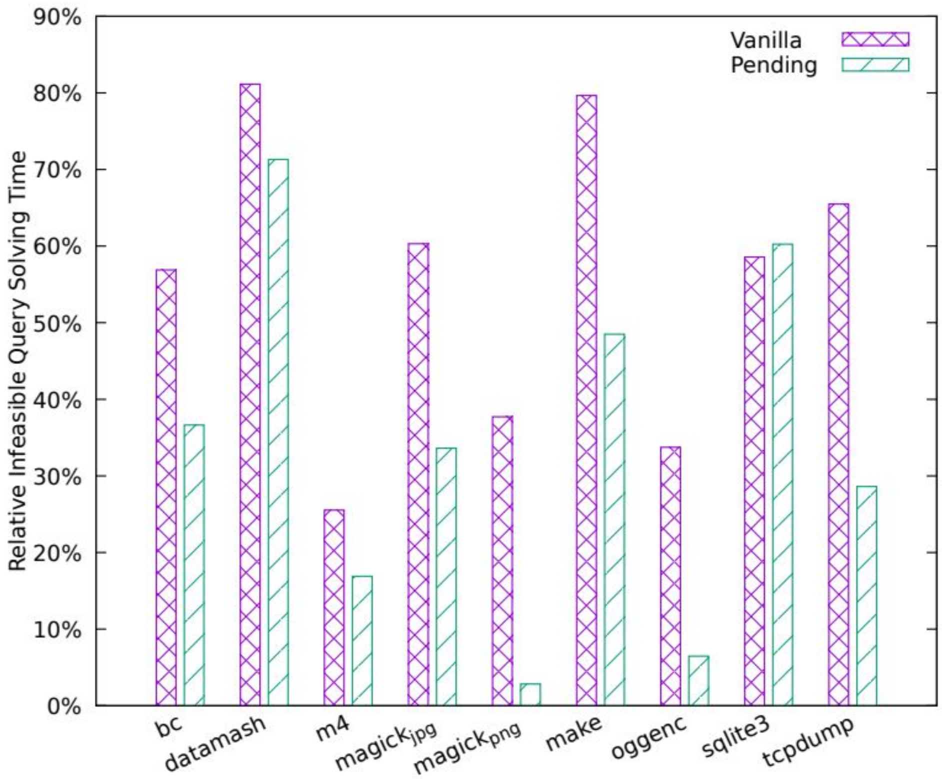
- 2 hour runs
- With and without seeds
- 3 search strategies: **random path**, DFS, depth biased
- 3 repetitions

- Case study on SQLite3 with 24 hour runs

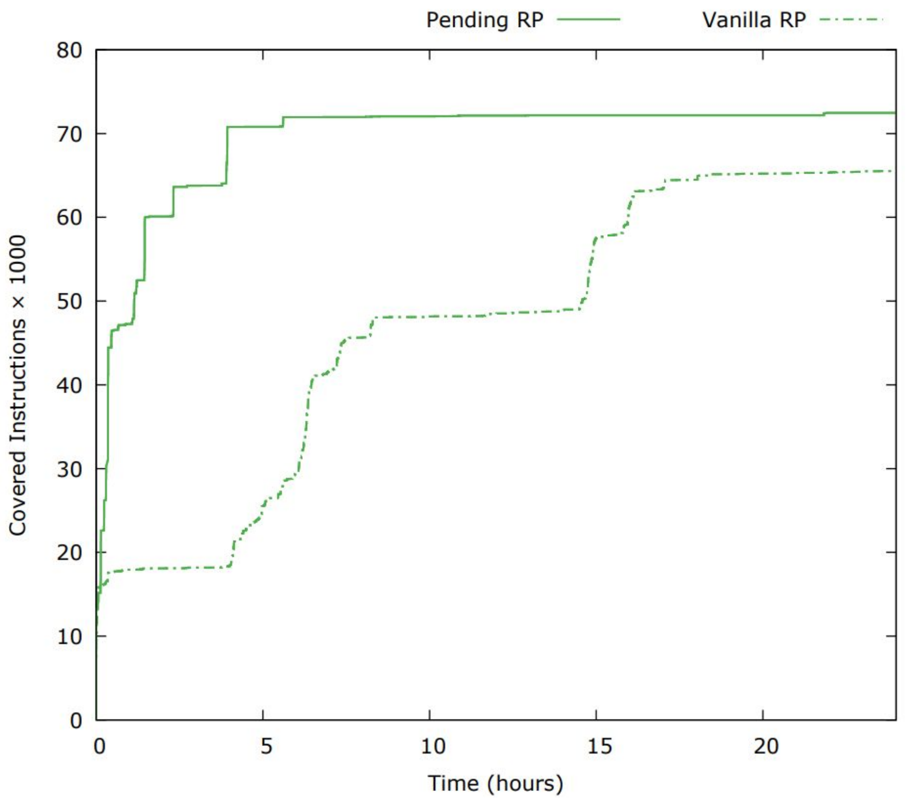
Vanilla vs Pending without seeds (random path)



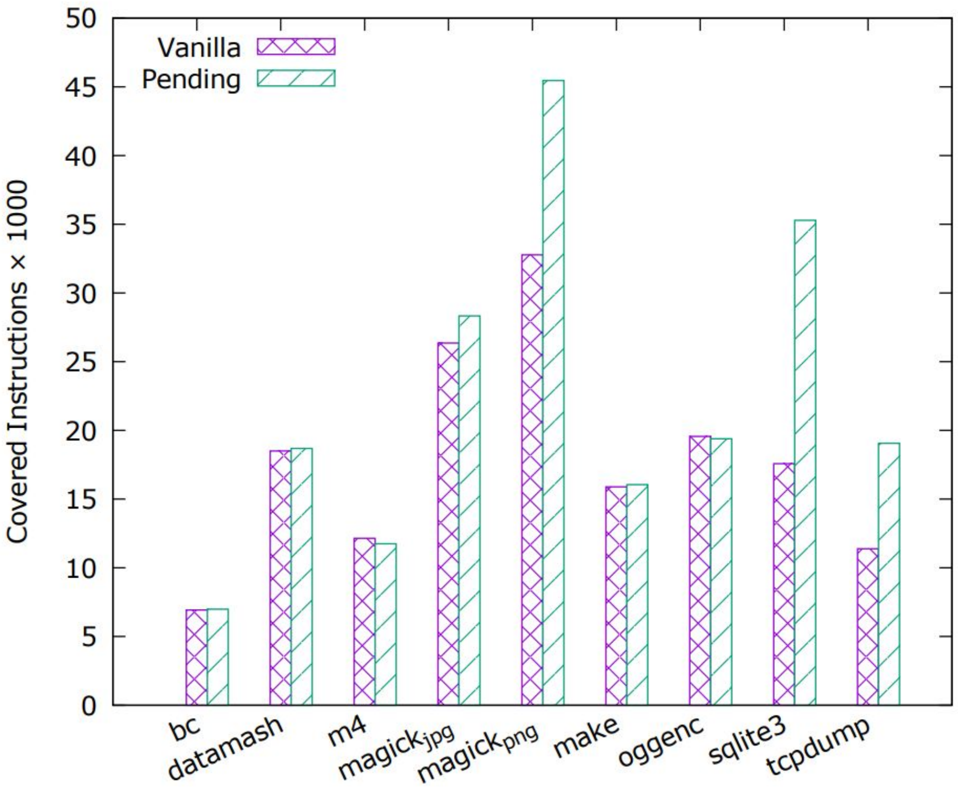
Proportion of time spent solving queries that were infeasible



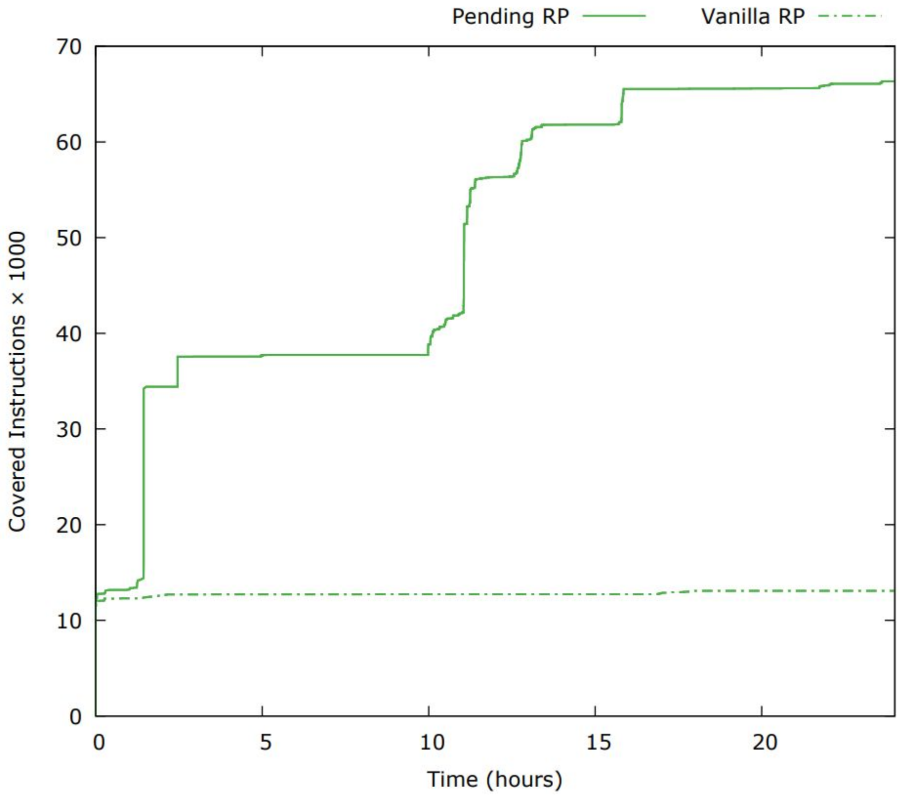
SQLite3: 24 hour run without seeds (random path)



Vanilla vs Pending with seeds (random path)

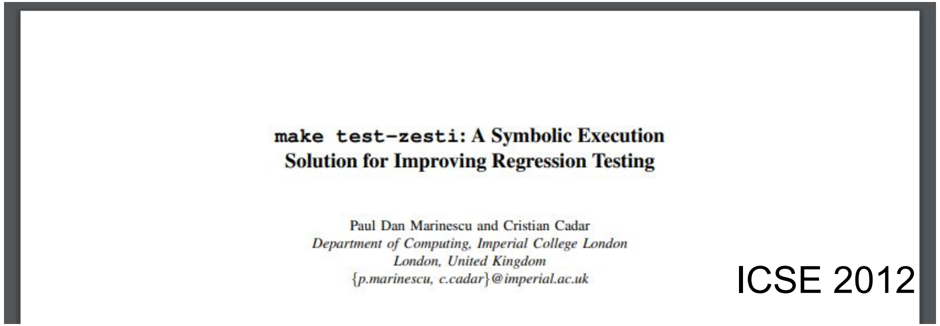


SQLite3: 24 hour run with seed



ZESTI and seeding

- Extension of KLEE for augmenting test suites
- Explores paths “around” a seed
- Easy to implement with pending constraints
- Found 2 bugs in tar, dwarfdump that were fixed

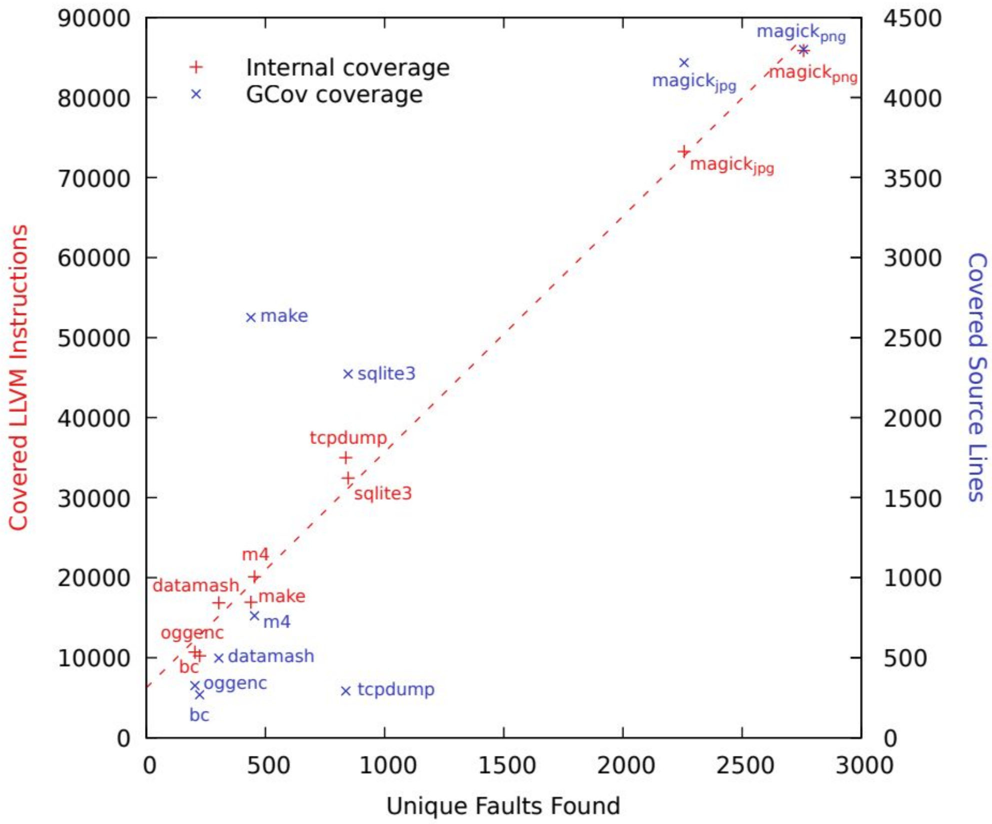

`make test-zesti`: A Symbolic Execution Solution for Improving Regression Testing

Paul Dan Marinescu and Cristian Cadar
Department of Computing, Imperial College London
London, United Kingdom
{p.marinescu, c.cadar}@imperial.ac.uk

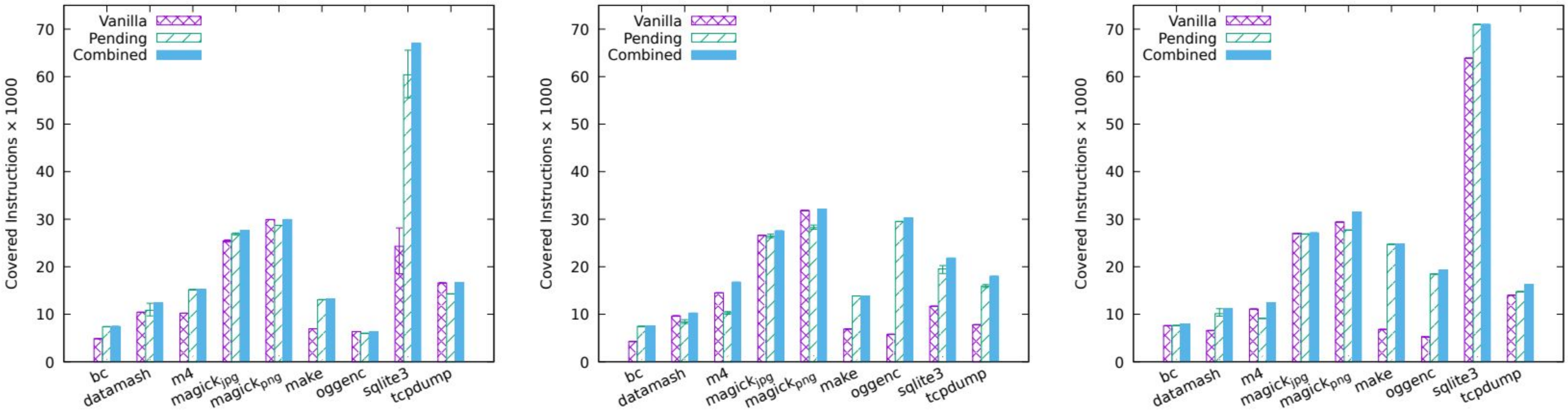
ICSE 2012

Conclusion

- Pending constraints
  - Tackles scalability of symbolic execution by aggressively following paths that are known to be feasible
- Effective in improving coverage for 8 challenging programs


Internal coverage
GCov coverage
Covered LLVM Instructions
Covered Source Lines
Unique Faults Found
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
500
1000
1500
2000
2500
3000
magick$_{png}$
magick$_{jpg}$
magick$_{png}$
magick$_{jpg}$
make
sqlite3
tcpdump
sqlite3
m4
datamash
make
m4
oggenc
datamash
bc
oggenc
tcpdump
bc

Without seeds

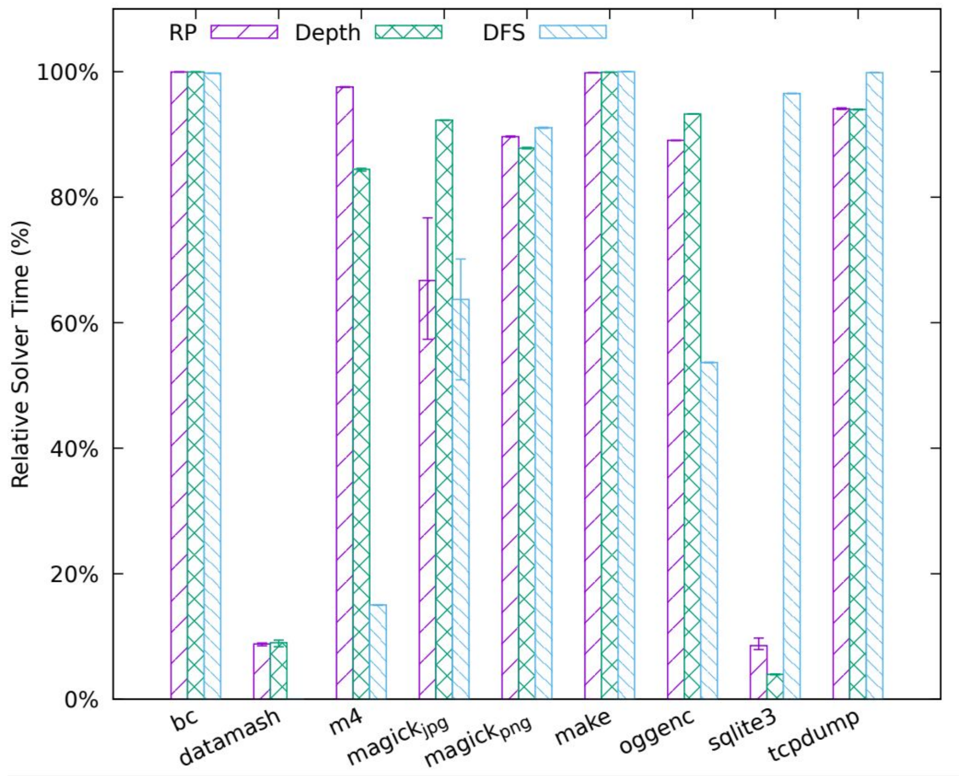


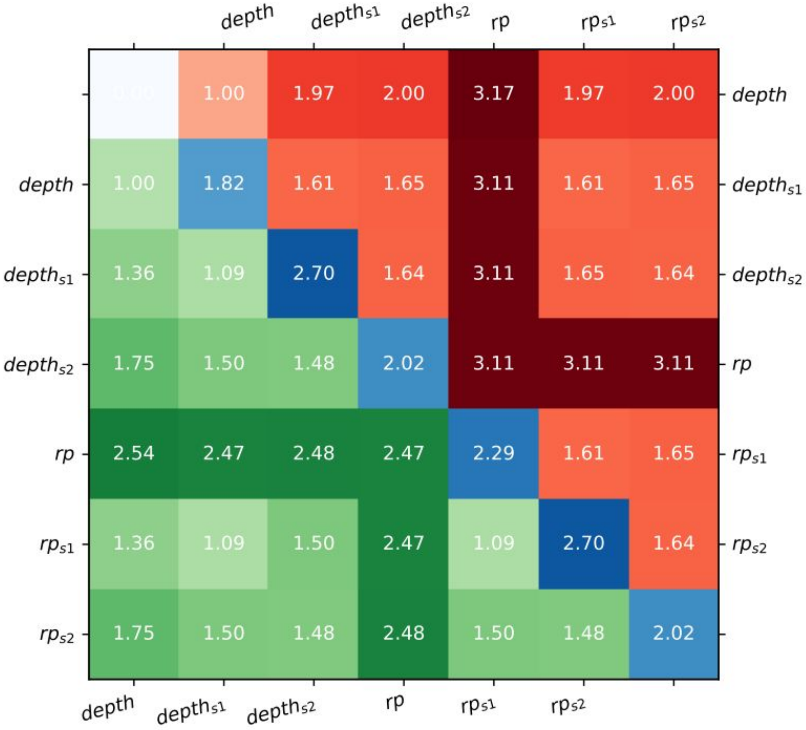
(a) Random Path

(b) Depth-Biased

(c) DFS

Time spent constraint solving by vanilla KLEE




depth
depth$_{s1}$
depth$_{s2}$
rp
rp$_{s1}$
rp$_{s2}$
1.00 1.97 2.00 3.17 1.97 2.00
1.00 1.82 1.61 1.65 3.11 1.61 1.65
1.36 1.09 2.70 1.64 3.11 1.65 1.64
1.75 1.50 1.48 2.02 3.11 3.11 3.11
2.54 2.47 2.48 2.47 2.29 1.61 1.65
1.36 1.09 1.50 2.47 1.09 2.70 1.64
1.75 1.50 1.48 2.48 1.50 1.48 2.02