The Trier Mutex Analyzer

DAEDALUS Project, 2002

Helmut Seidl

Varmo Vene

Markus Müller-Olm

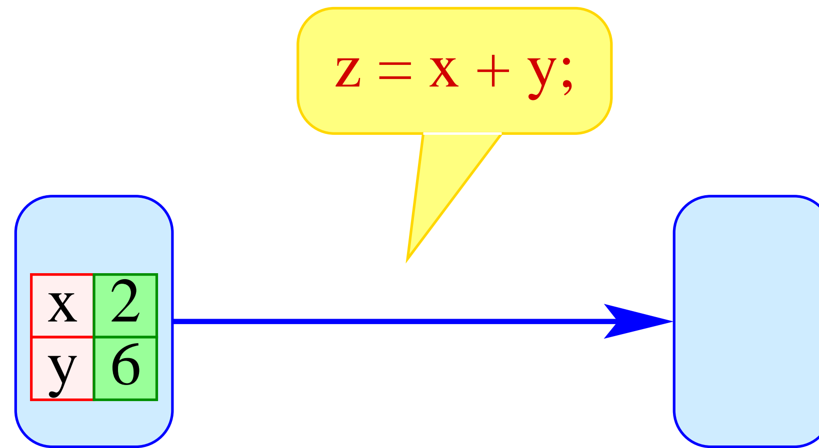
Challenge:

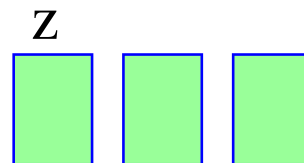
Design and implementation of an analyzer for
multi-threaded C programs that ...

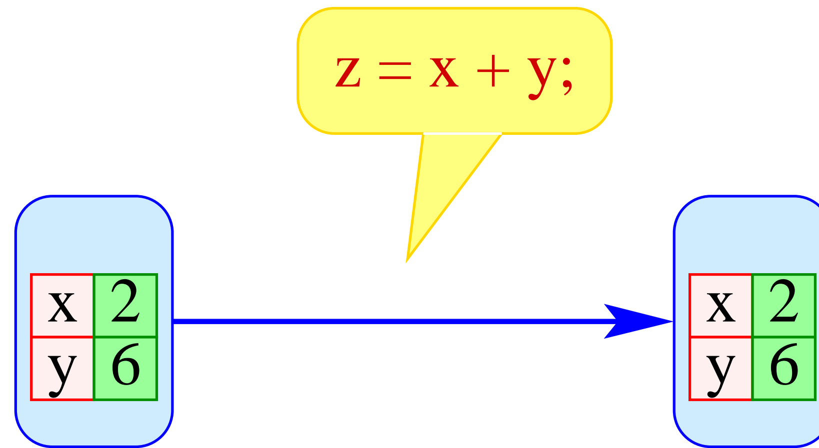
+ deals with (arrays of) function pointers;
+ deals with heap-allocated data;
+ tracks flow- and context-sensitive information ...
+ detects global invariants $\rightarrow$ Mutual Exclusion
+ is sufficiently efficient :-))

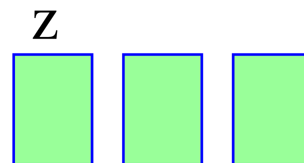
1. The Key Idea:

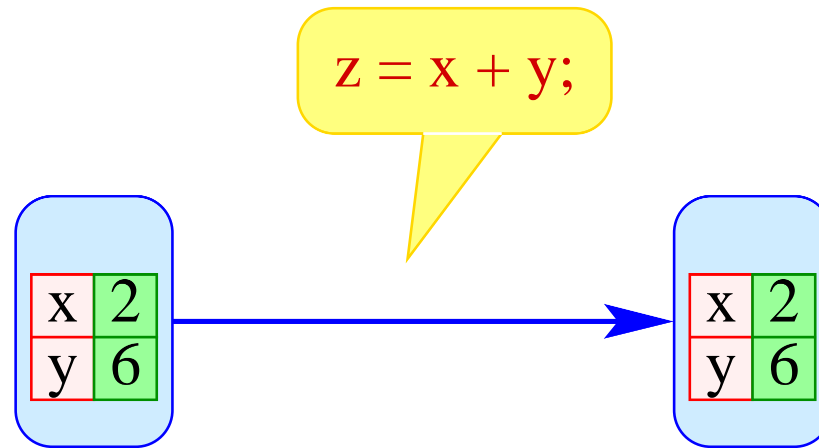
- track values of local variables through (classical) interprocedural analysis;
- approximate globals and heap by safe invariants ...

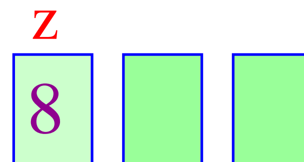

z = x + y;
x
2
y
6


z

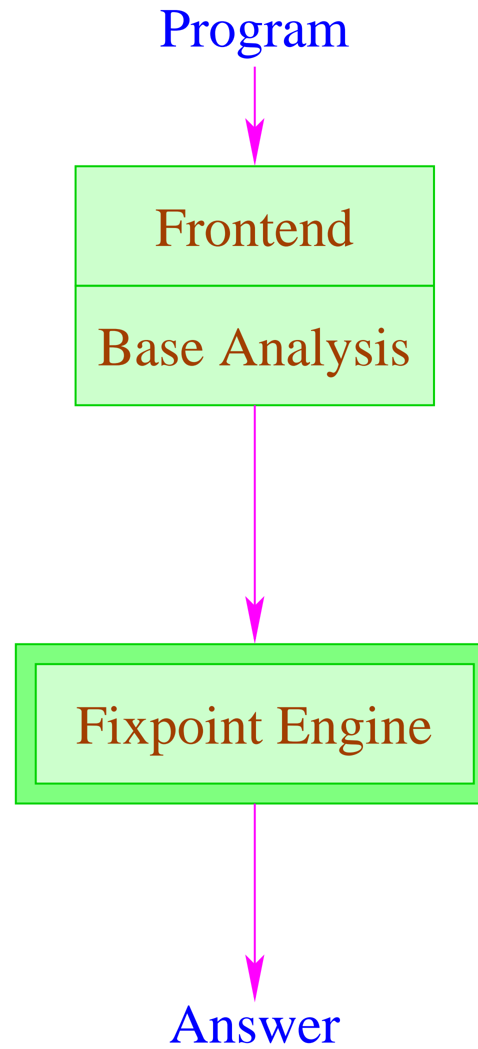

z = x + y;
x 2
y 6
x 2
y 6


z

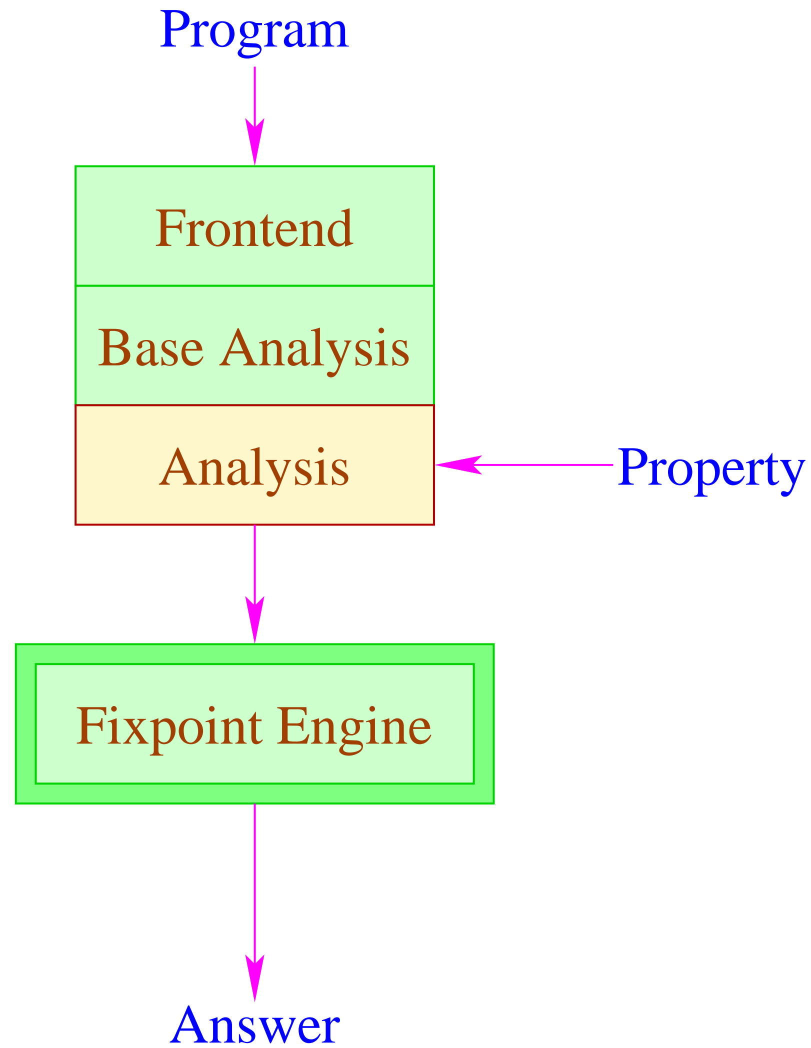

z = x + y;
x
2
y
6
x
2
y
6


z
8

Our System:



Our System:



Problem:

- Our fixpoint engine is local;
- It explores explores the program demand-driven:

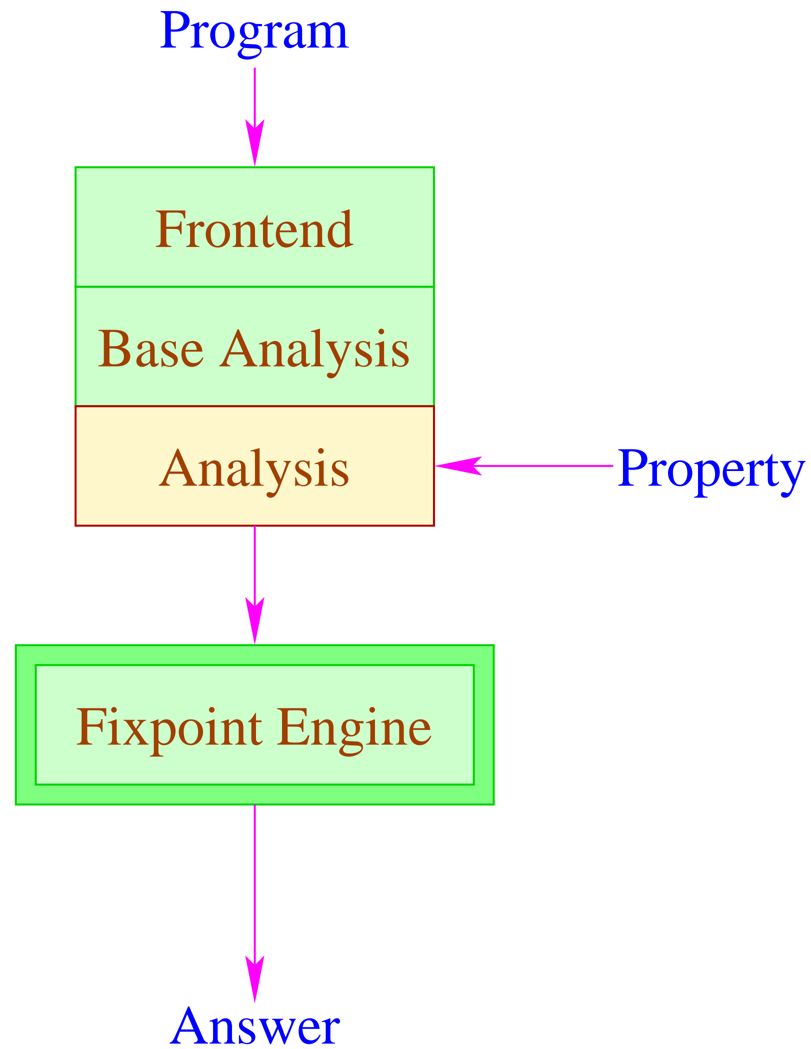
  $\rightarrow$ functions are only analyzed for actually occurring arguments :-)

  $\rightarrow$ global variables formally depend also on un-realizable function calls :-(

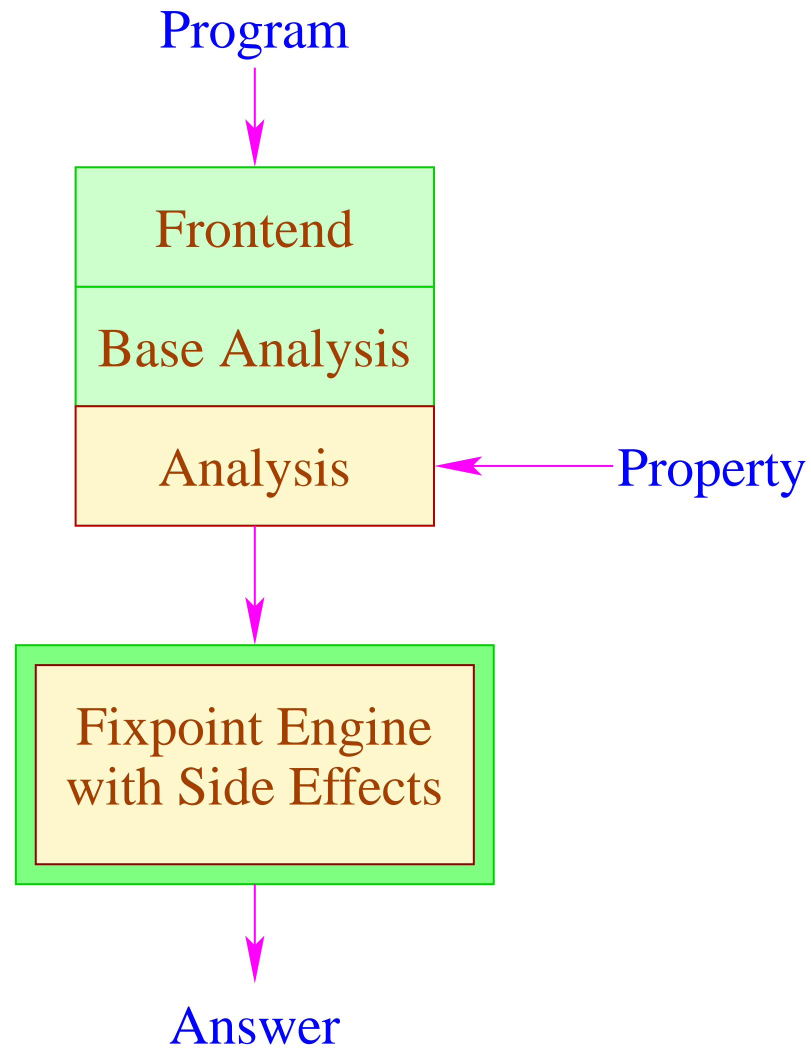
  $\rightarrow$ local solving does not succeed :-((

$\Longrightarrow$ solving with side effects

Our System:



Our System:



2. Some Details: Base Analysis

Demand:

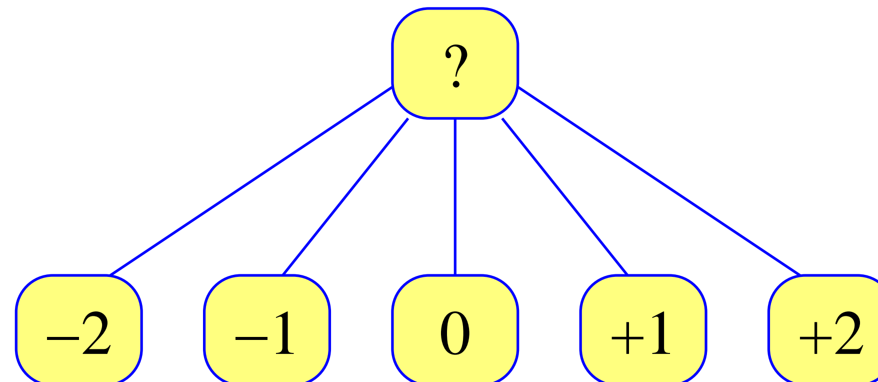
- tracking of storage layout;
- tracking of addresses;
- tracking of int values;
- tracking of mutex locks;
- tracking of strings.

2.1. int Constants

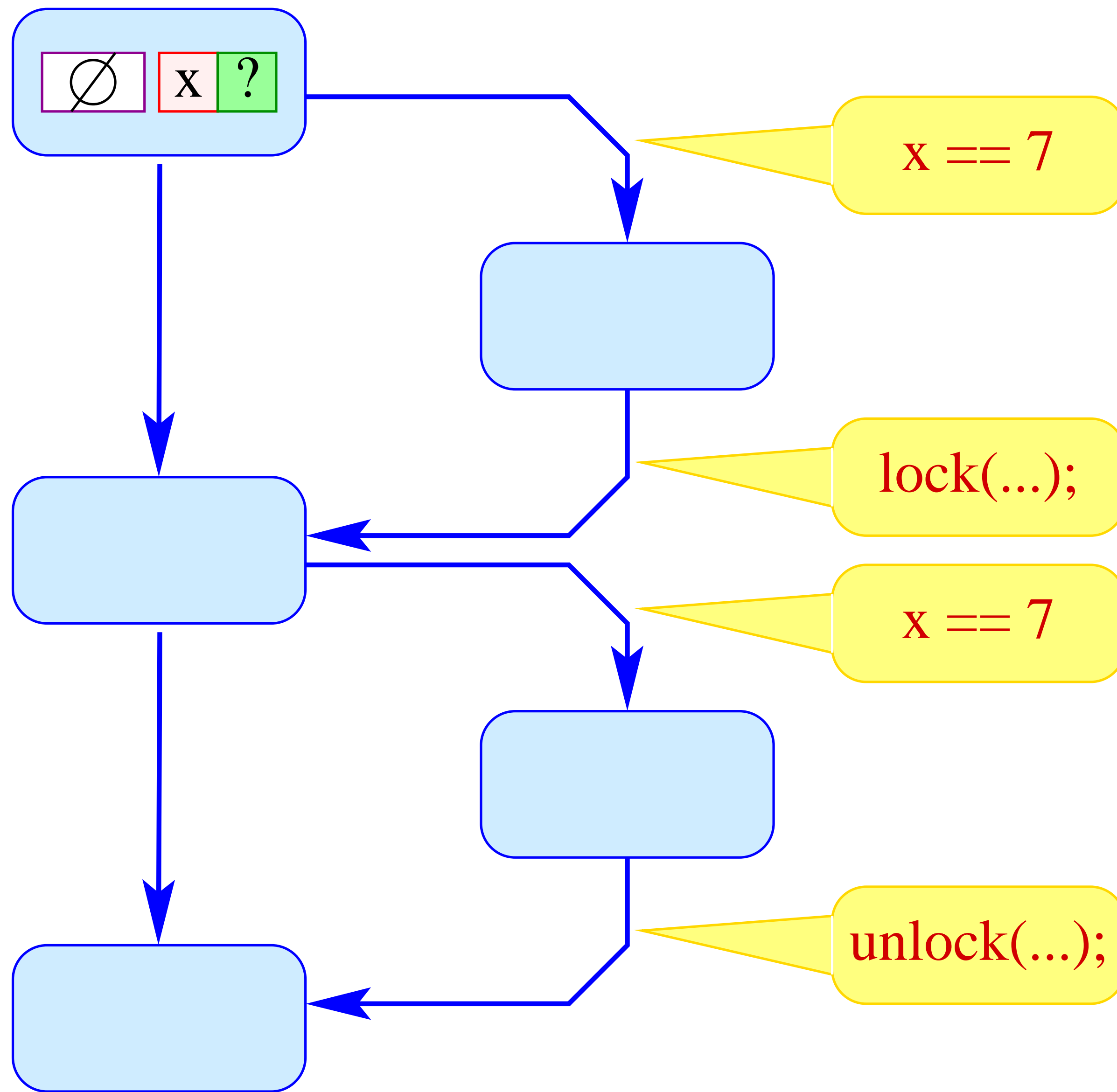
Example:

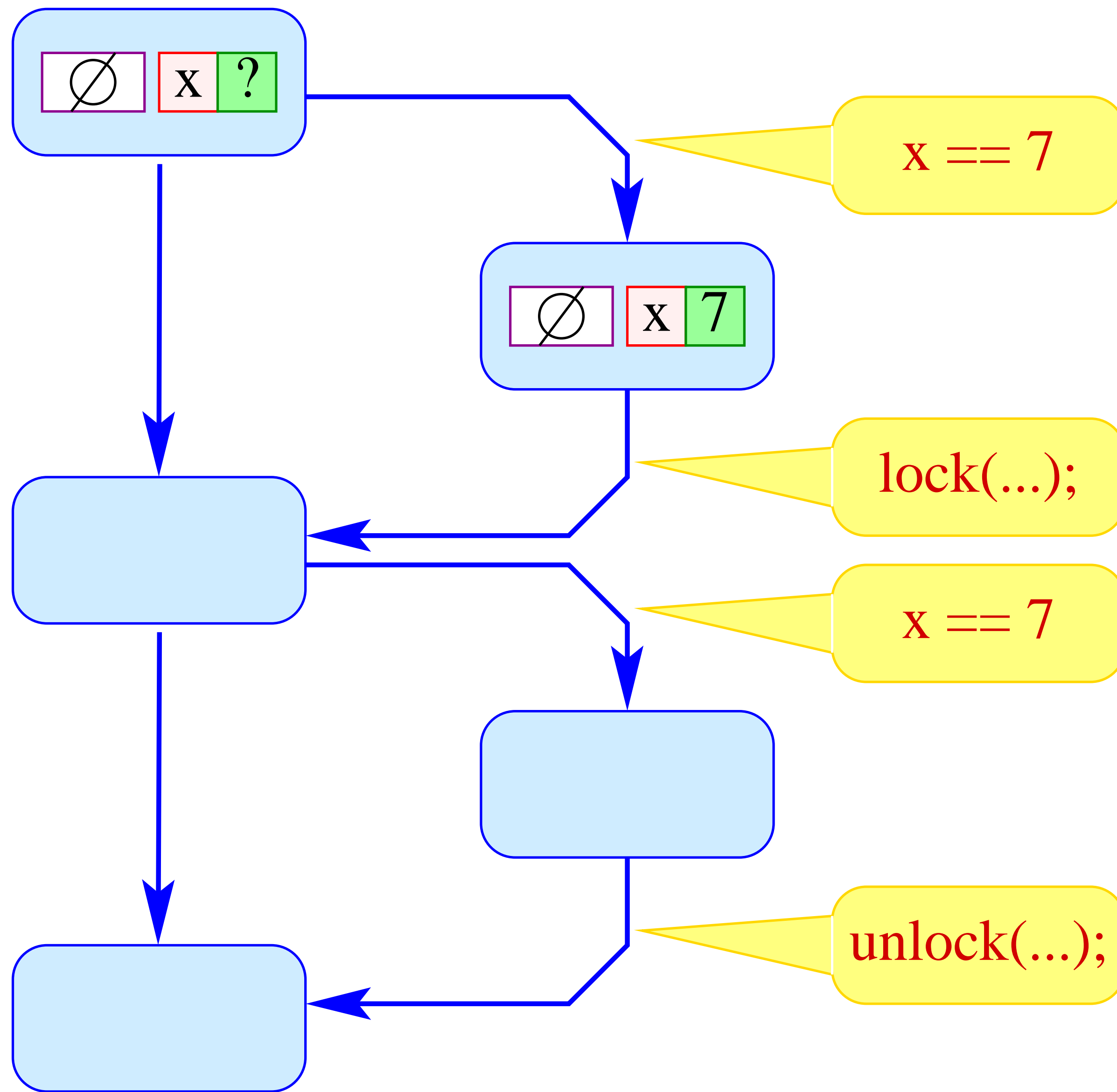
```
if (x == 7) pthread_mutex_lock (&me);
...
if (x == 7) pthread_mutex_unlock (&me);
```

The Classical Analysis:

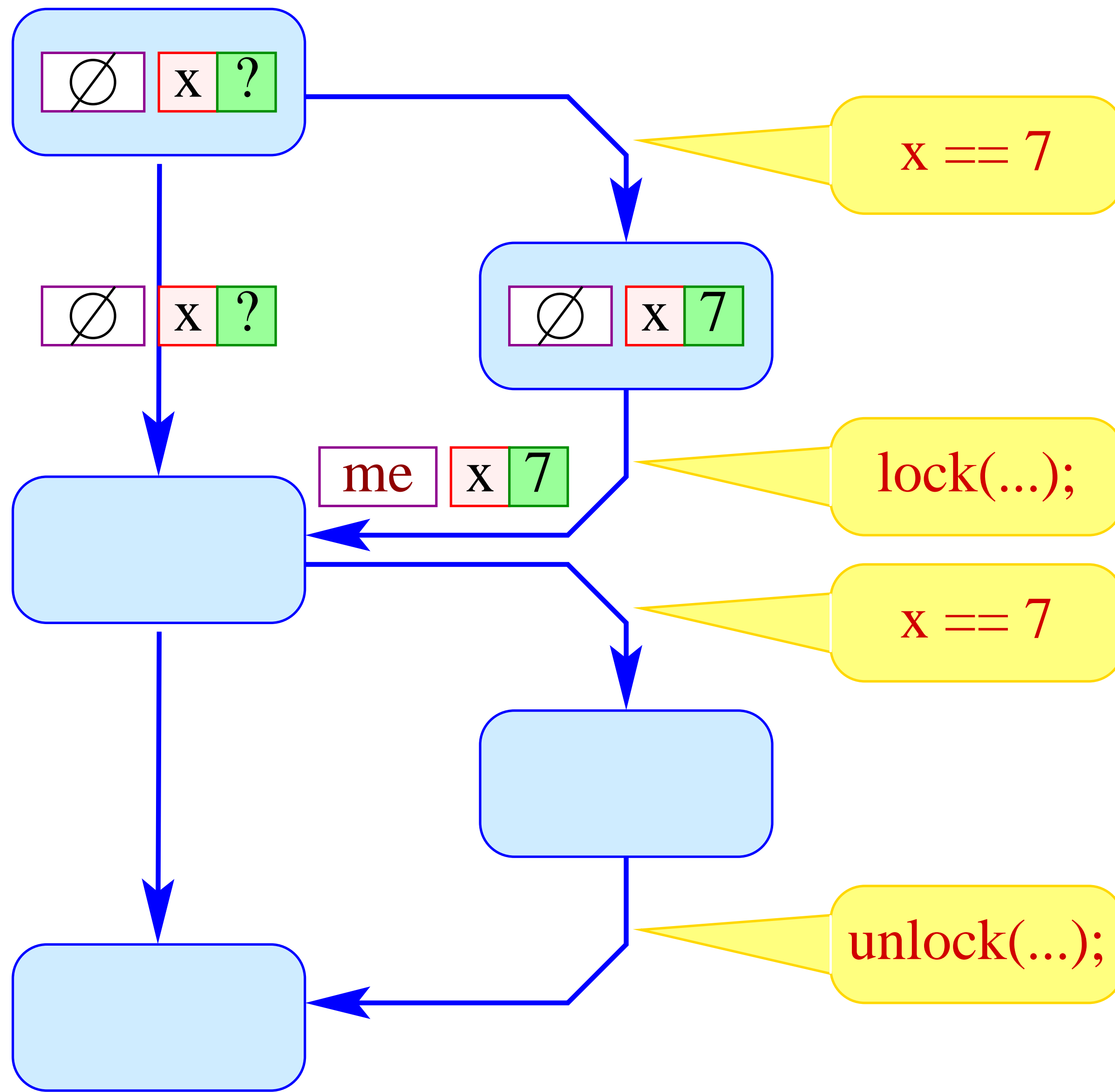


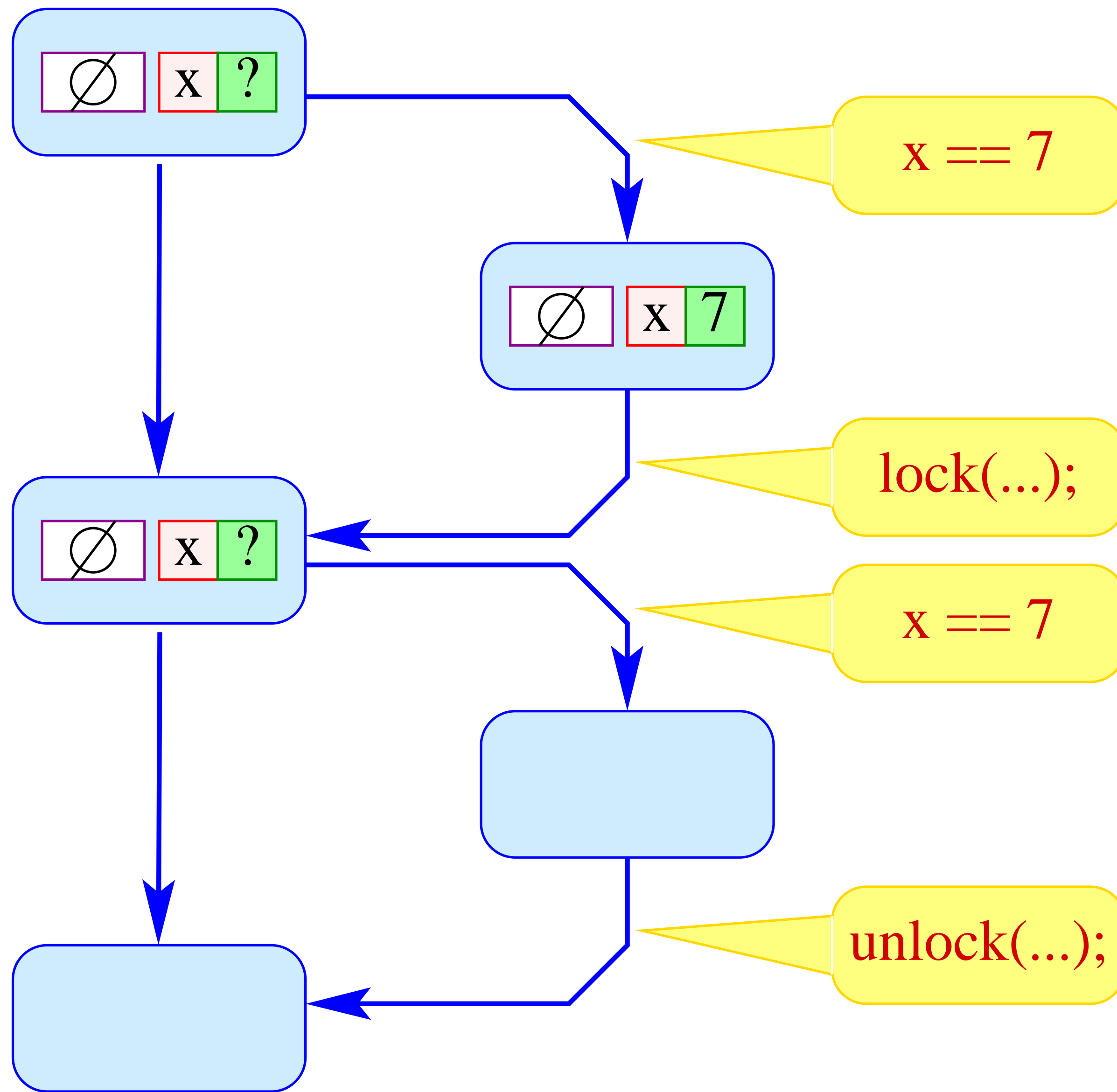
... does not track negative information :-(

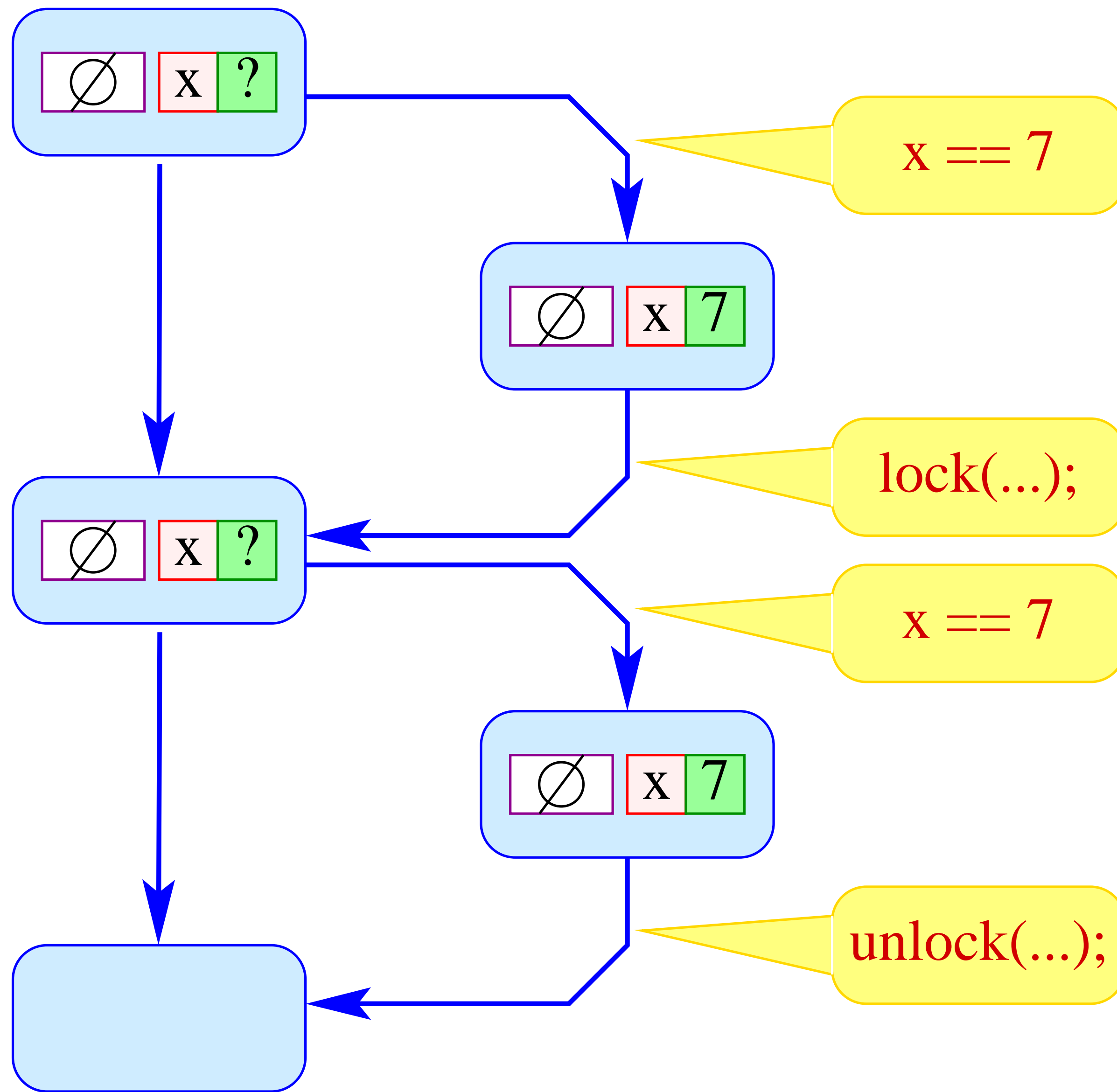

∅
x
?
x == 7
lock(...);
x == 7
unlock(...);

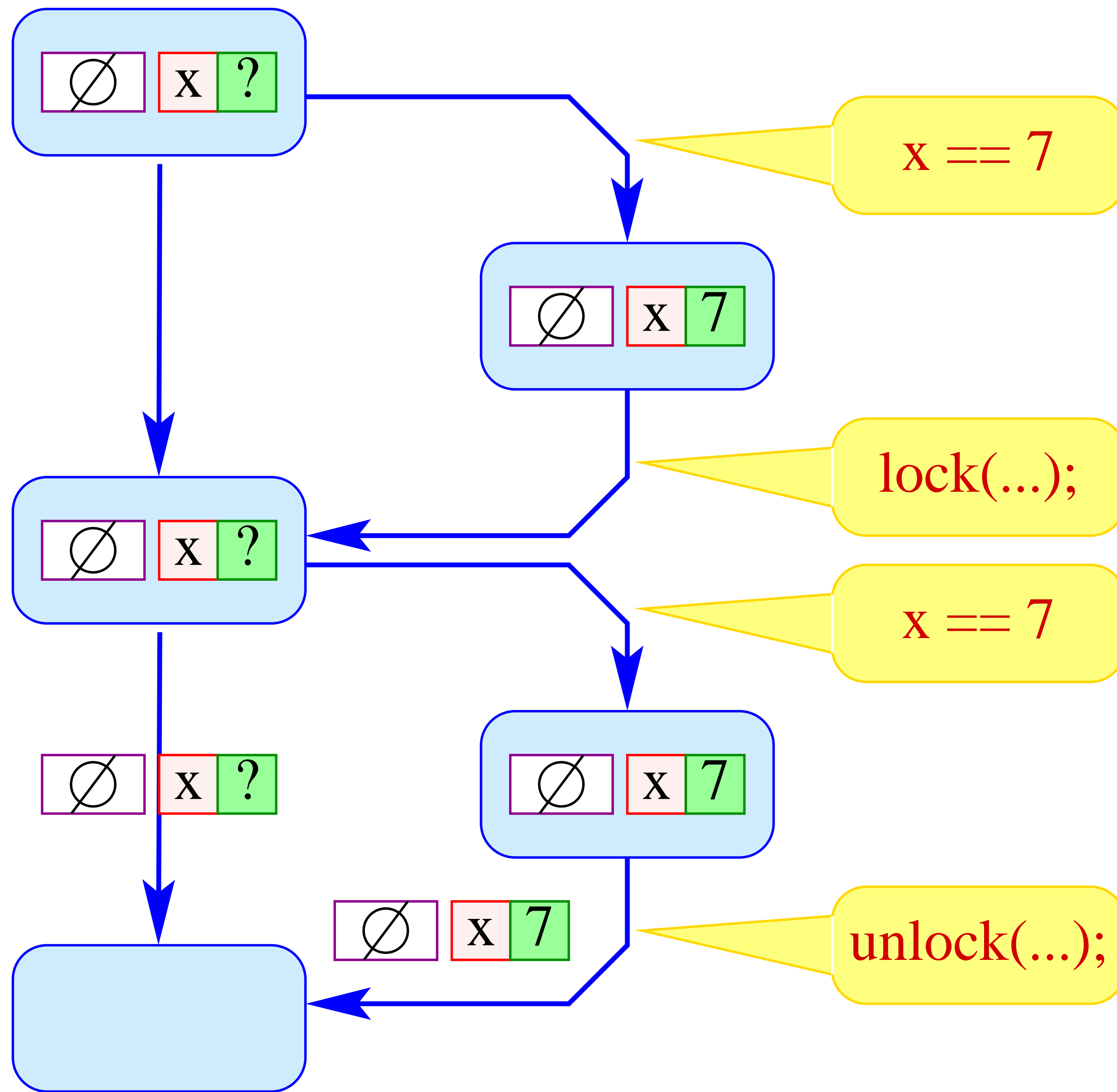


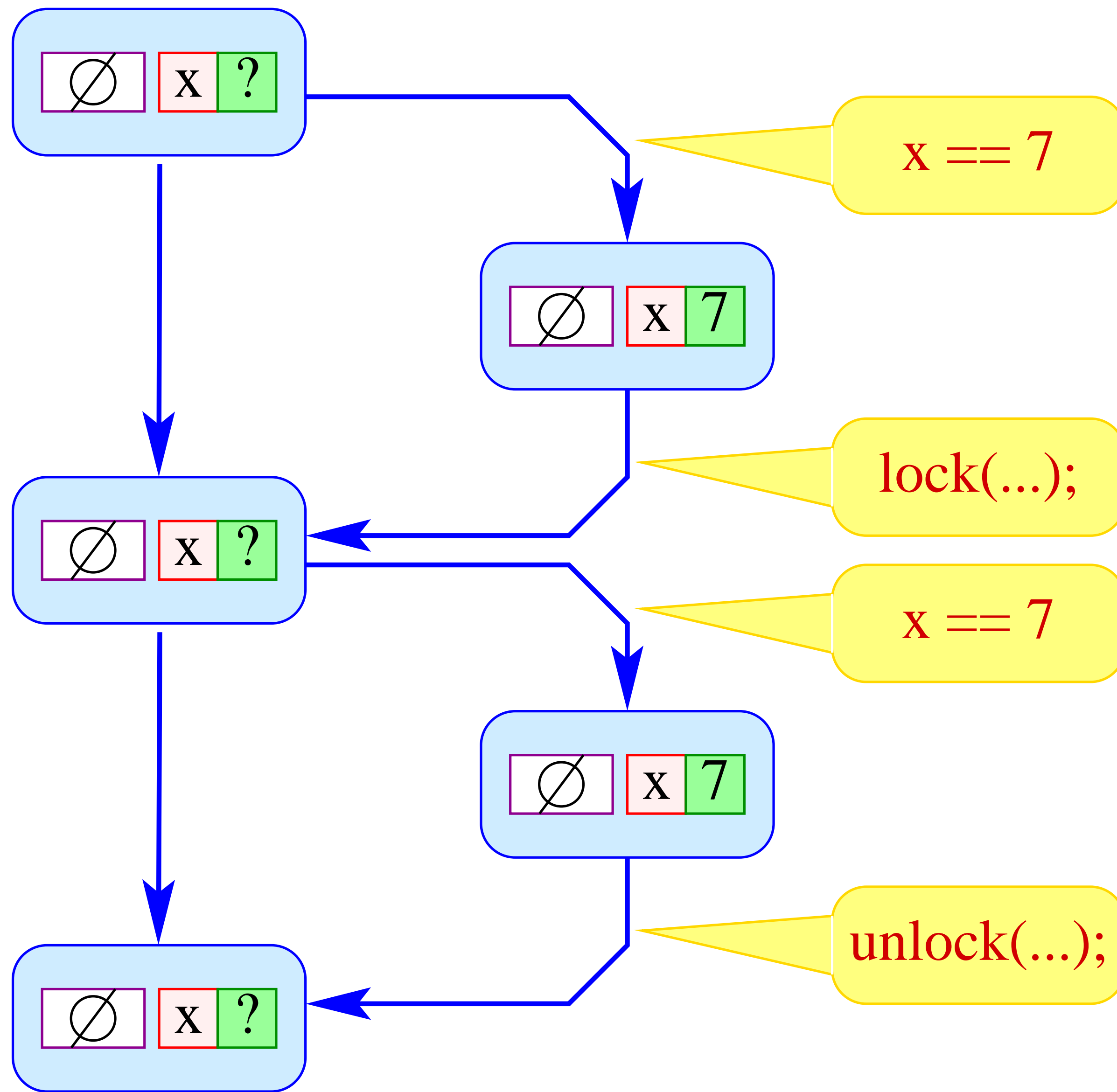
∅
x
?
x == 7
∅
x
7
lock(...);
x == 7
unlock(...);


∅
x
?
∅
x
?
∅
x
7
me
x
7
x == 7
lock(...);
x == 7
unlock(...);

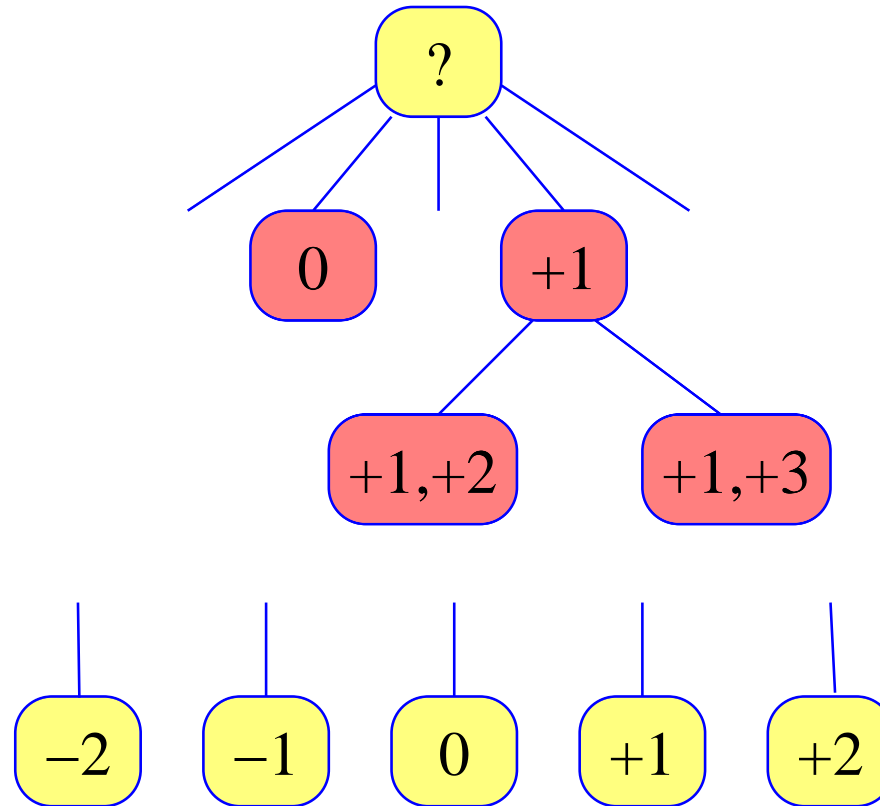

∅
x
?
x == 7
∅
x
7
lock(...);
∅
x
?
x == 7
unlock(...);

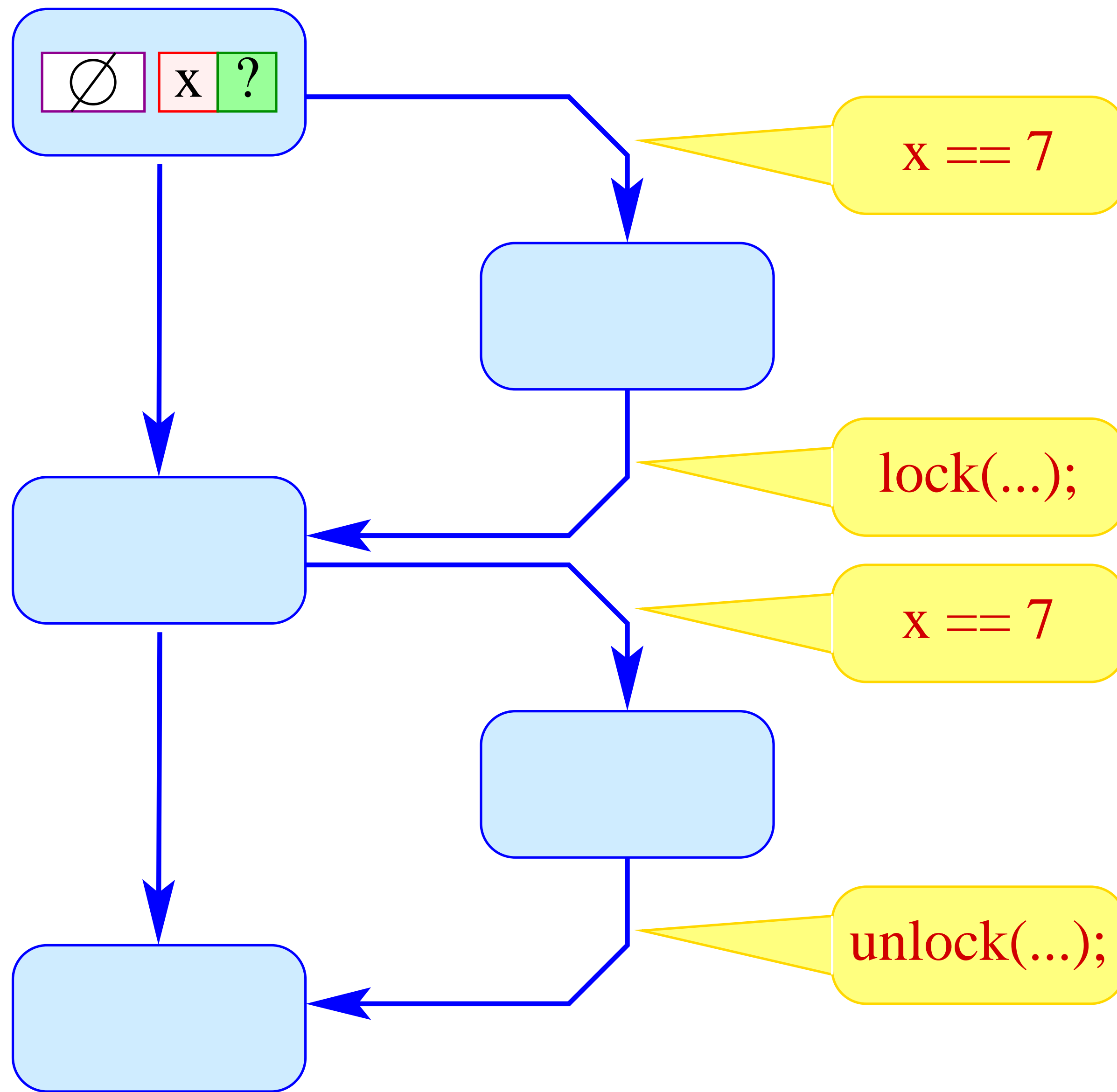

∅
x
?
x == 7
∅
x
7
lock(...);
∅
x
?
x == 7
∅
x
7
unlock(...);

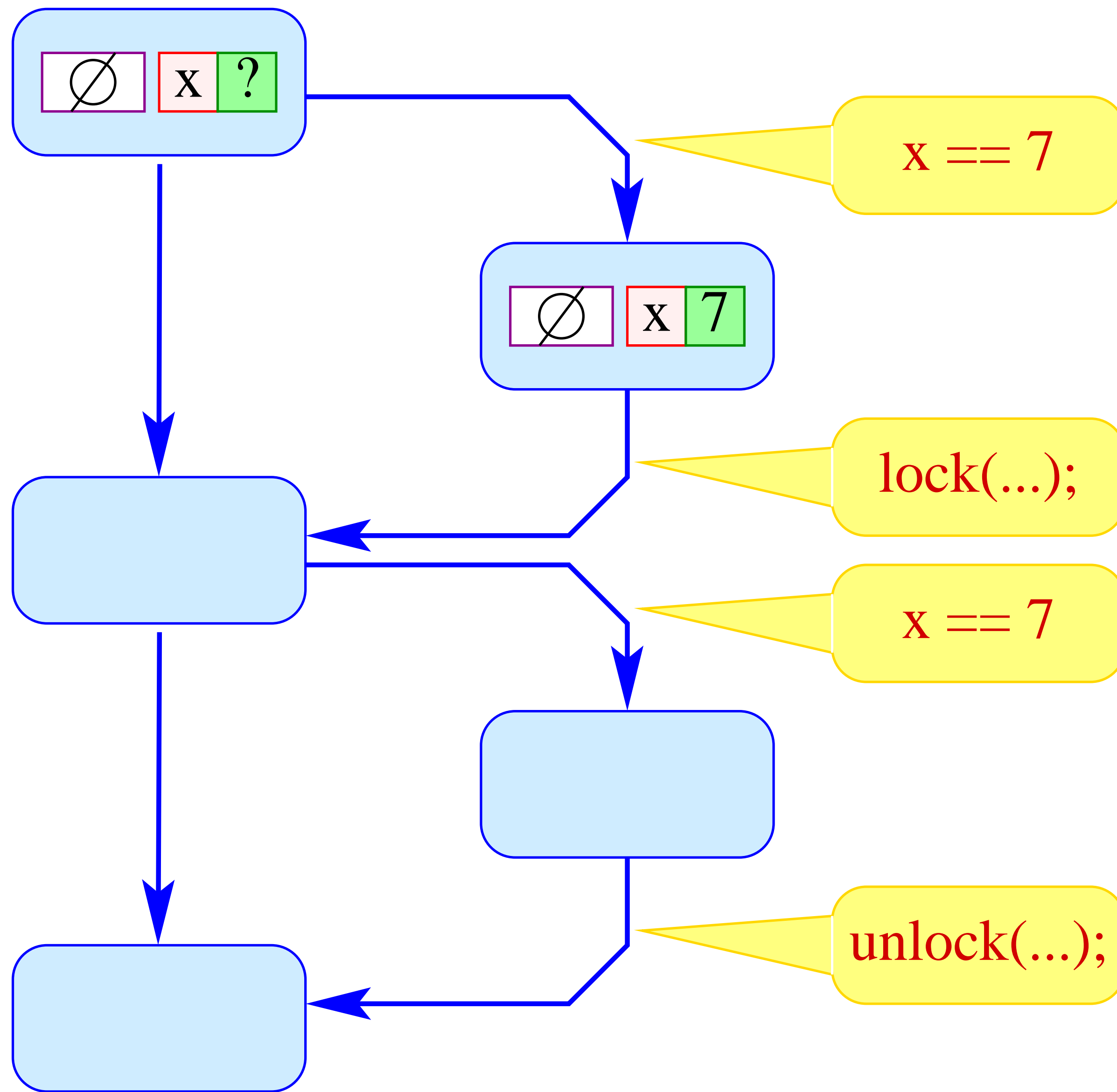

∅
x
?
x == 7
∅
x
7
∅
x
?
lock(...);
x == 7
∅
x
?
∅
x
7
∅
x
7
unlock(...);

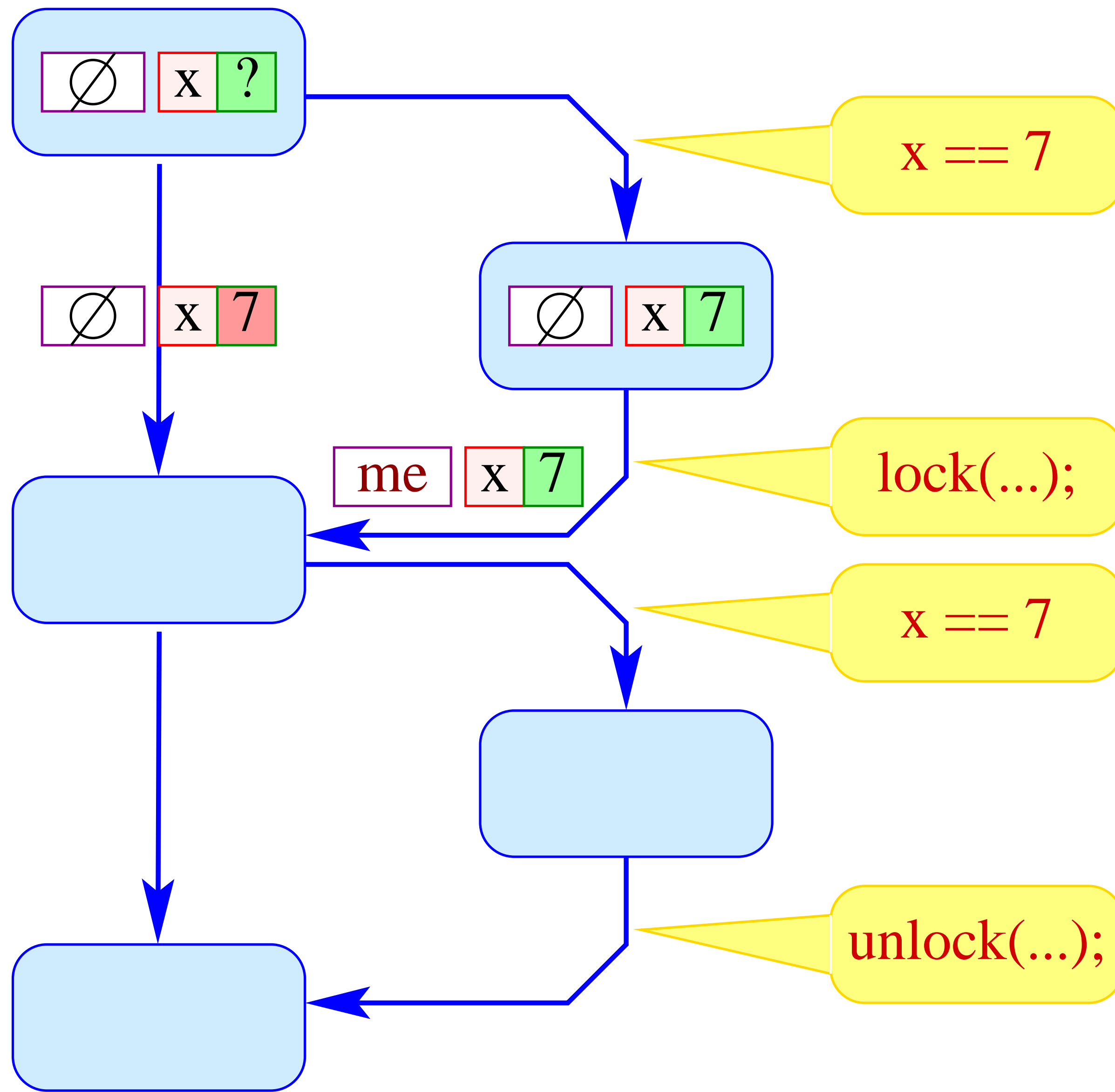

∅
x
?
x == 7
∅
x
7
lock(...);
∅
x
?
x == 7
∅
x
7
unlock(...);
∅
x
?

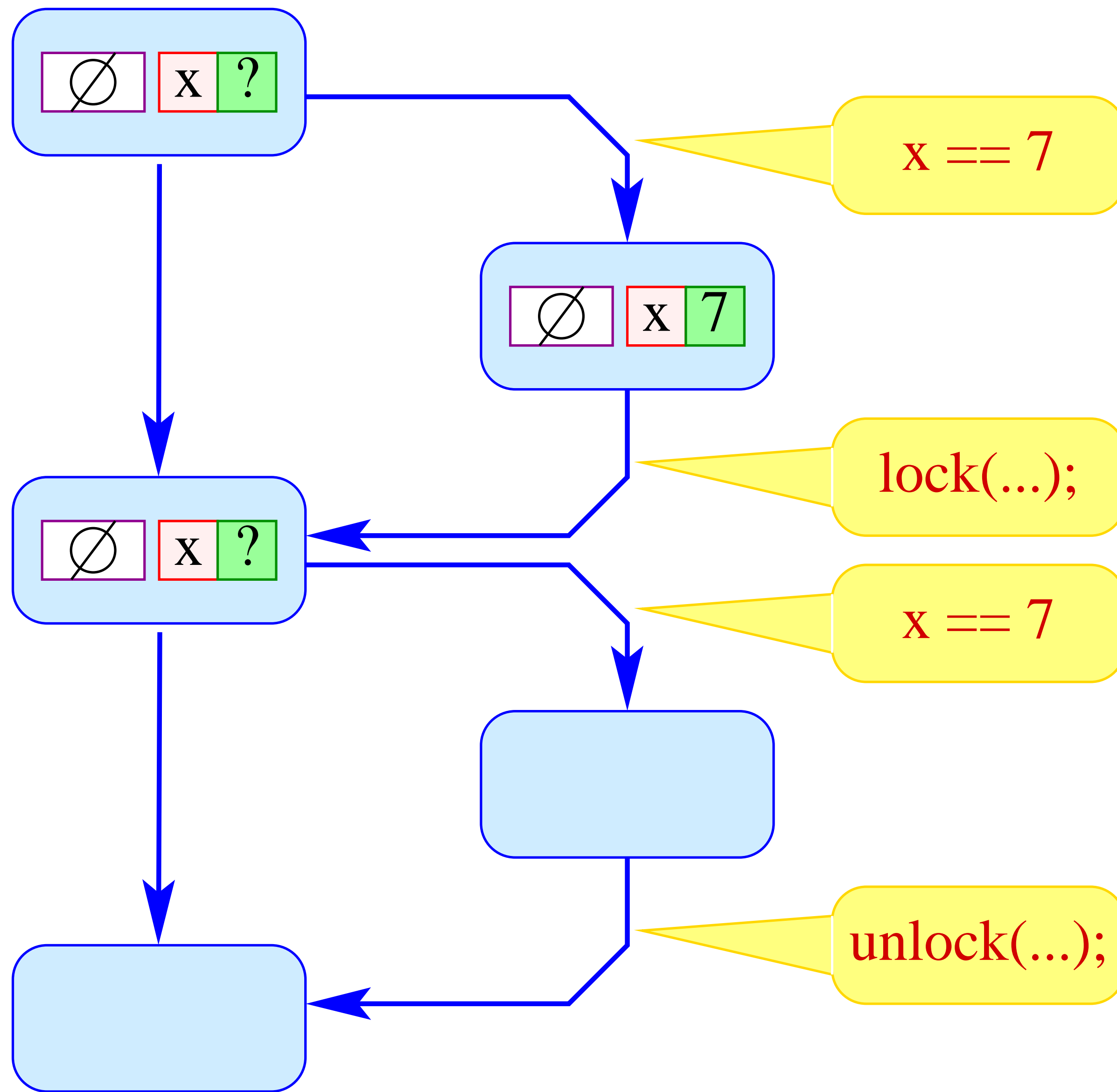
Idea 1:



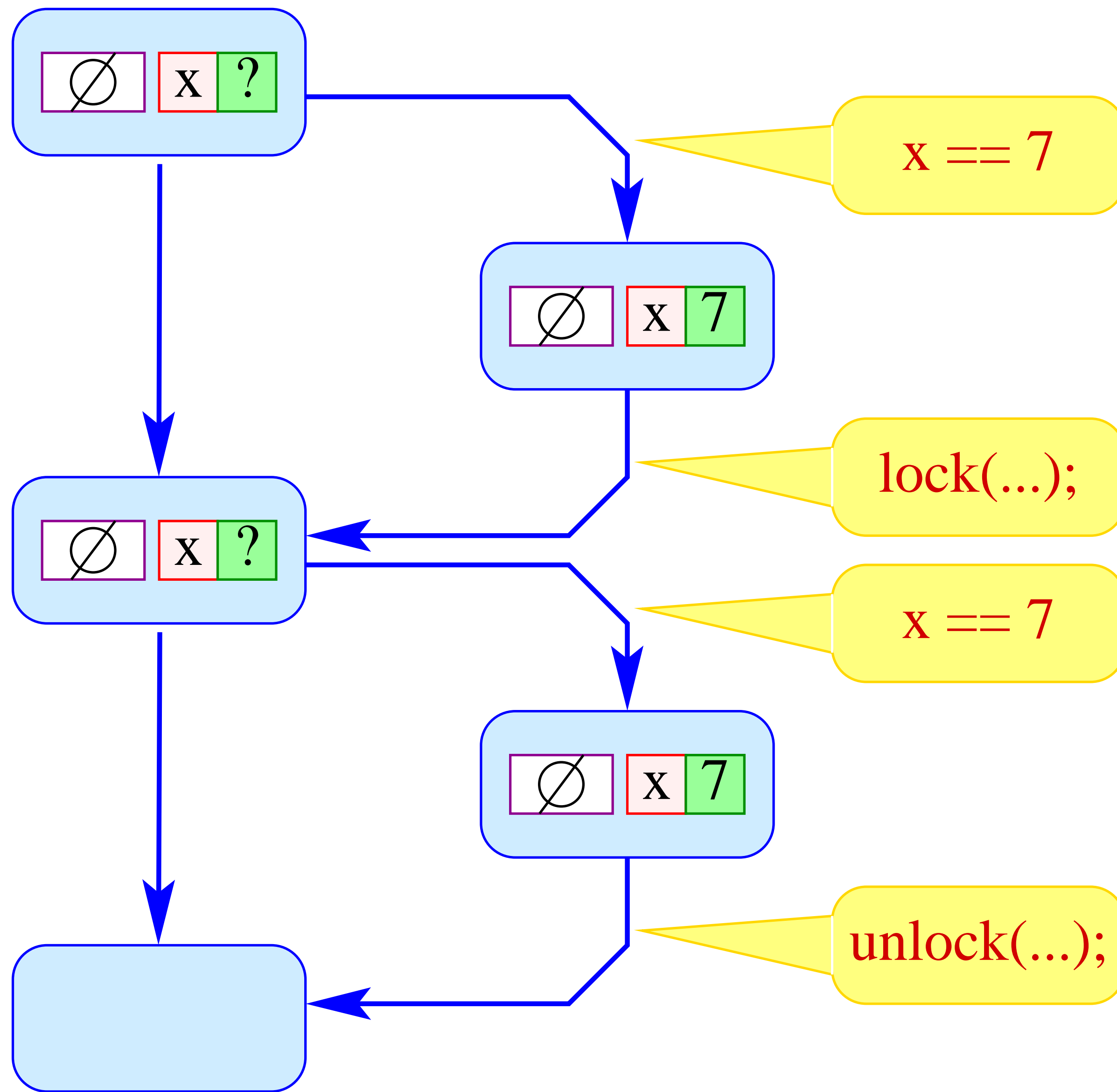

∅
x
?
x == 7
lock(...);
x == 7
unlock(...);

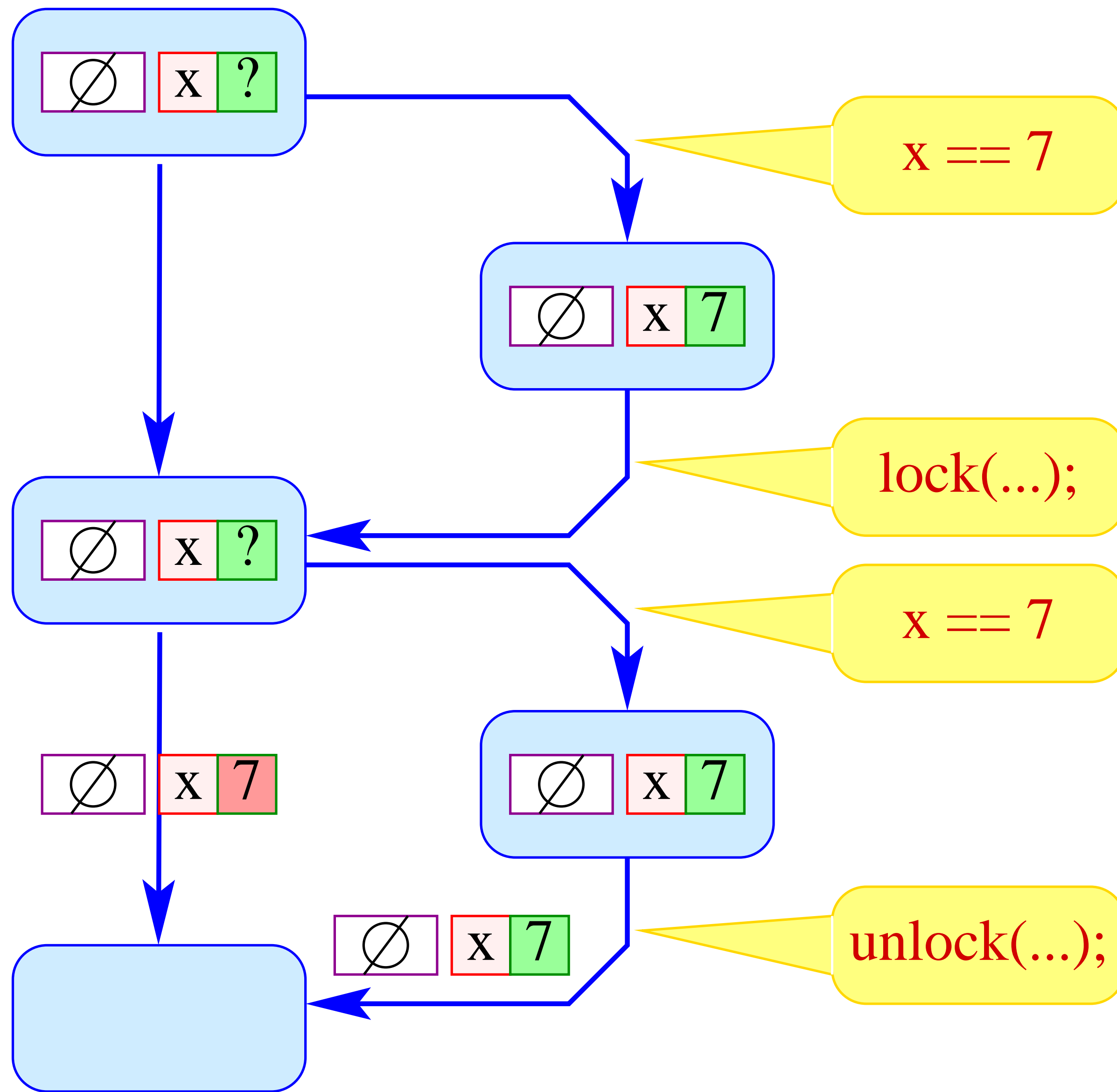

∅
x
?
x == 7
∅
x
7
lock(...);
x == 7
unlock(...);

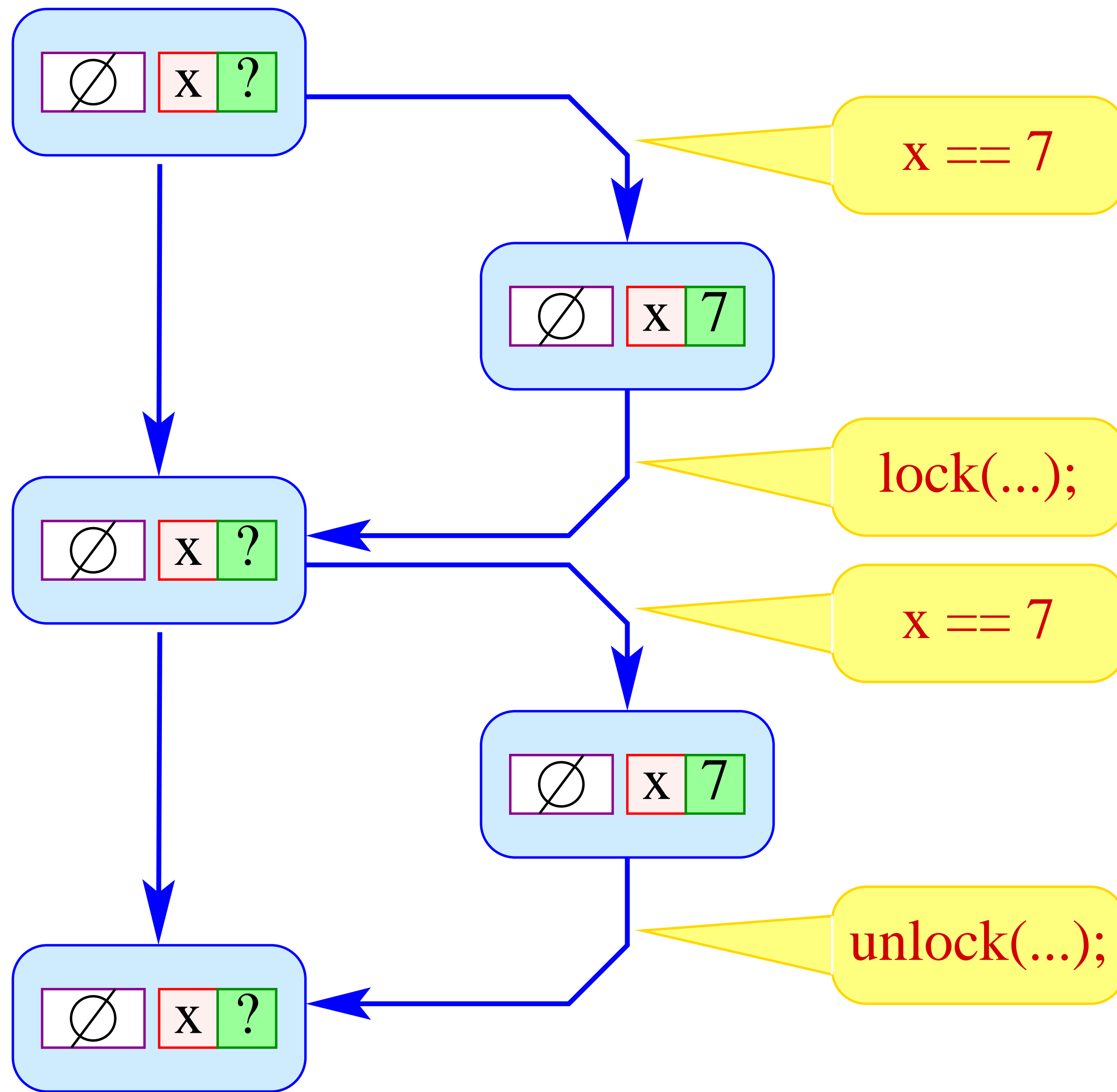

∅
x
?
x == 7
∅
x
7
∅
x
7
me
x
7
lock(...);
x == 7
unlock(...);



∅
x
?
x == 7
∅
x
7
lock(...);
∅
x
?
x == 7
unlock(...);


∅
x
?
x == 7
∅
x
7
lock(...);
∅
x
?
x == 7
∅
x
7
unlock(...);

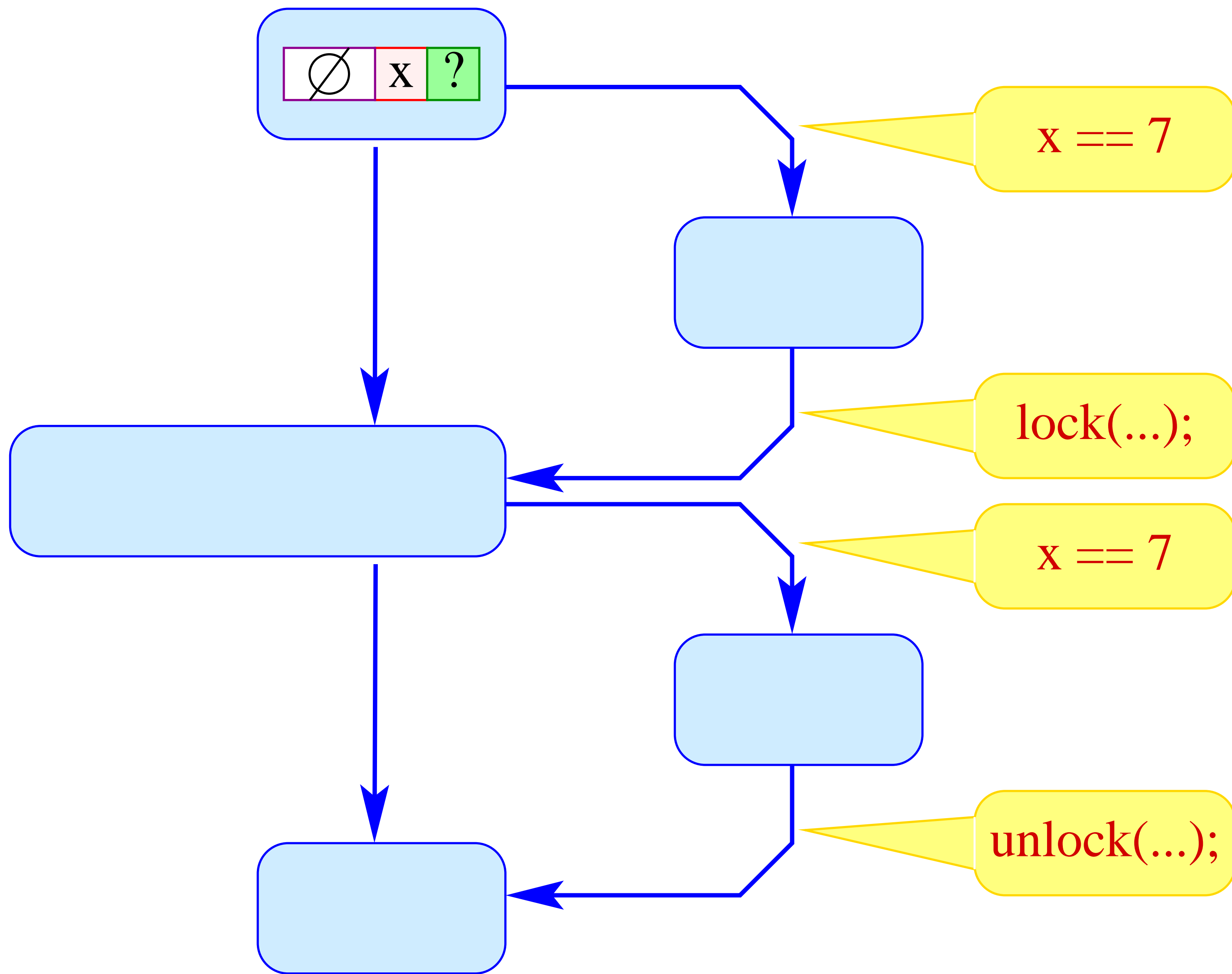

x ?
x == 7
x 7
lock(...);
x ?
x == 7
x 7
x 7
x 7
unlock(...);

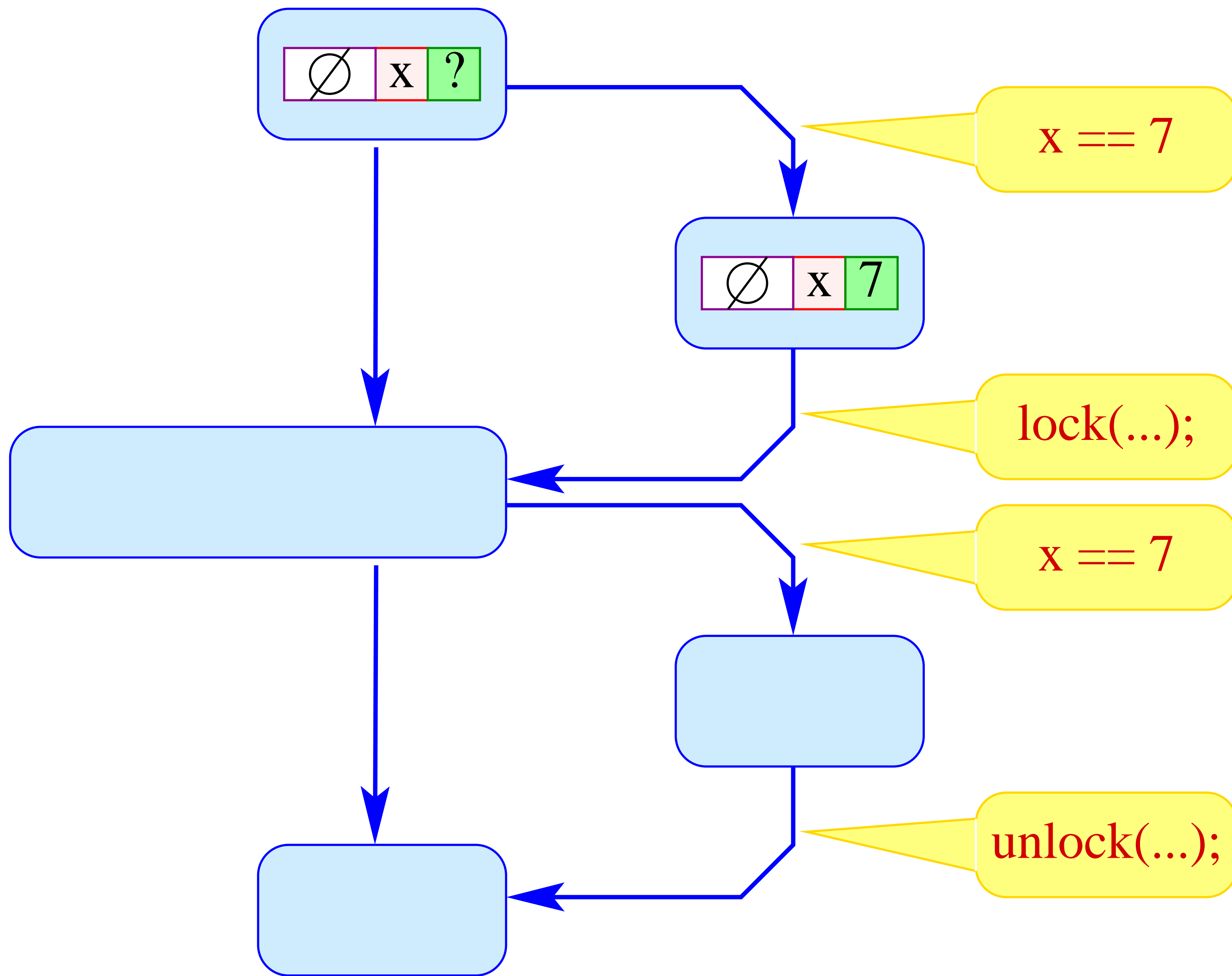

∅
x
?
x == 7
∅
x
7
lock(...);
∅
x
?
x == 7
∅
x
7
unlock(...);
∅
x
?

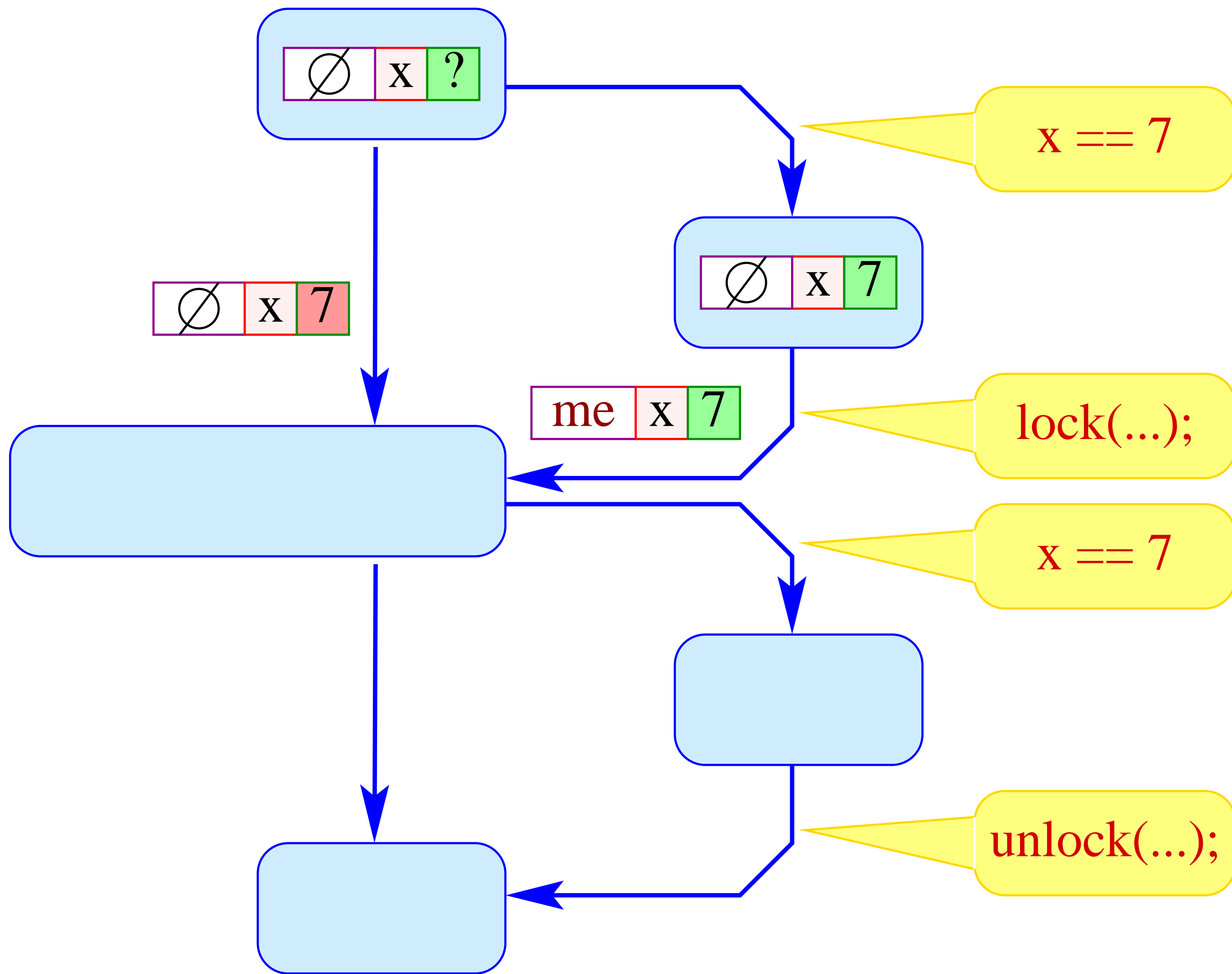
Anti-constants are not quite sufficient :-(

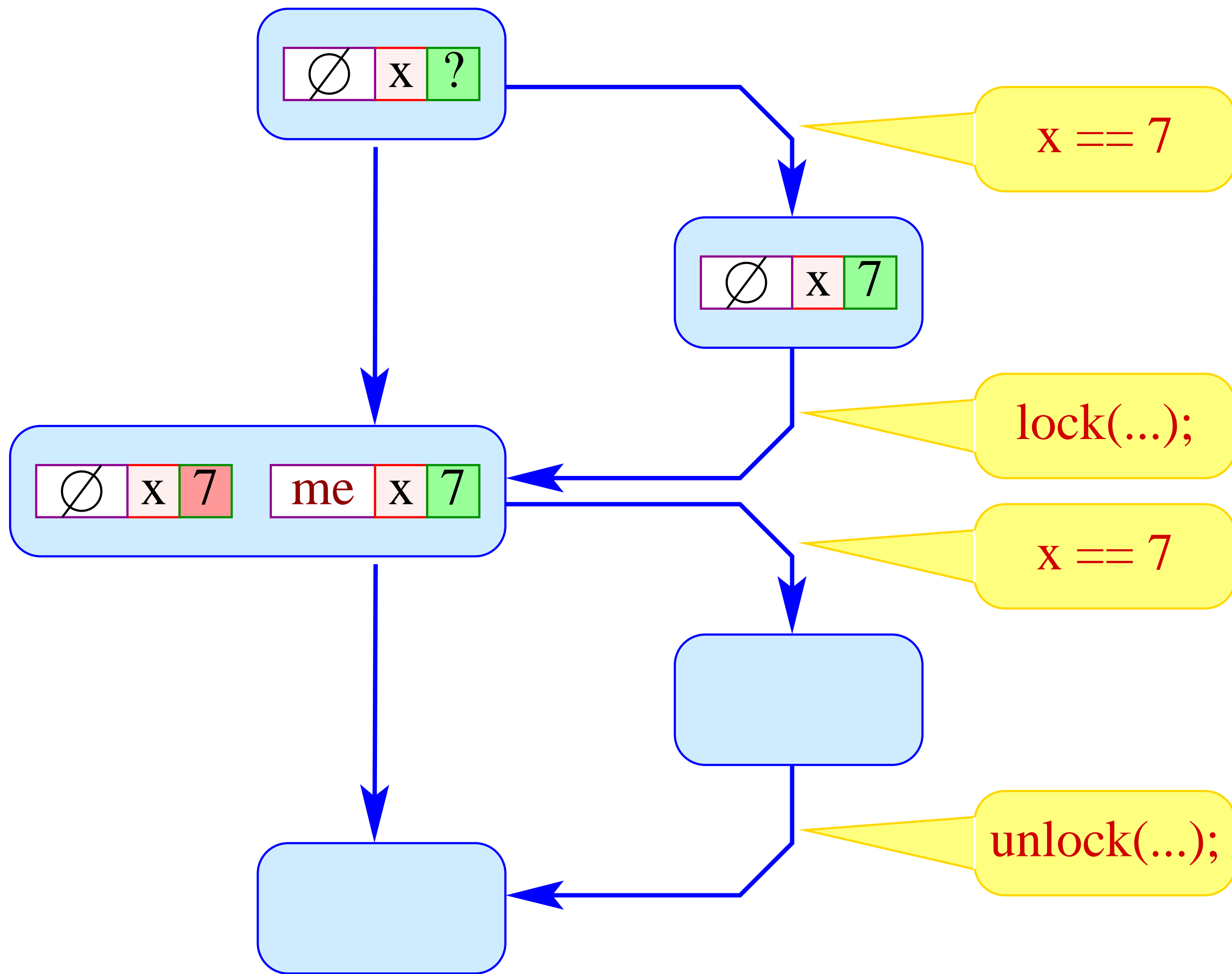
Idea 2:

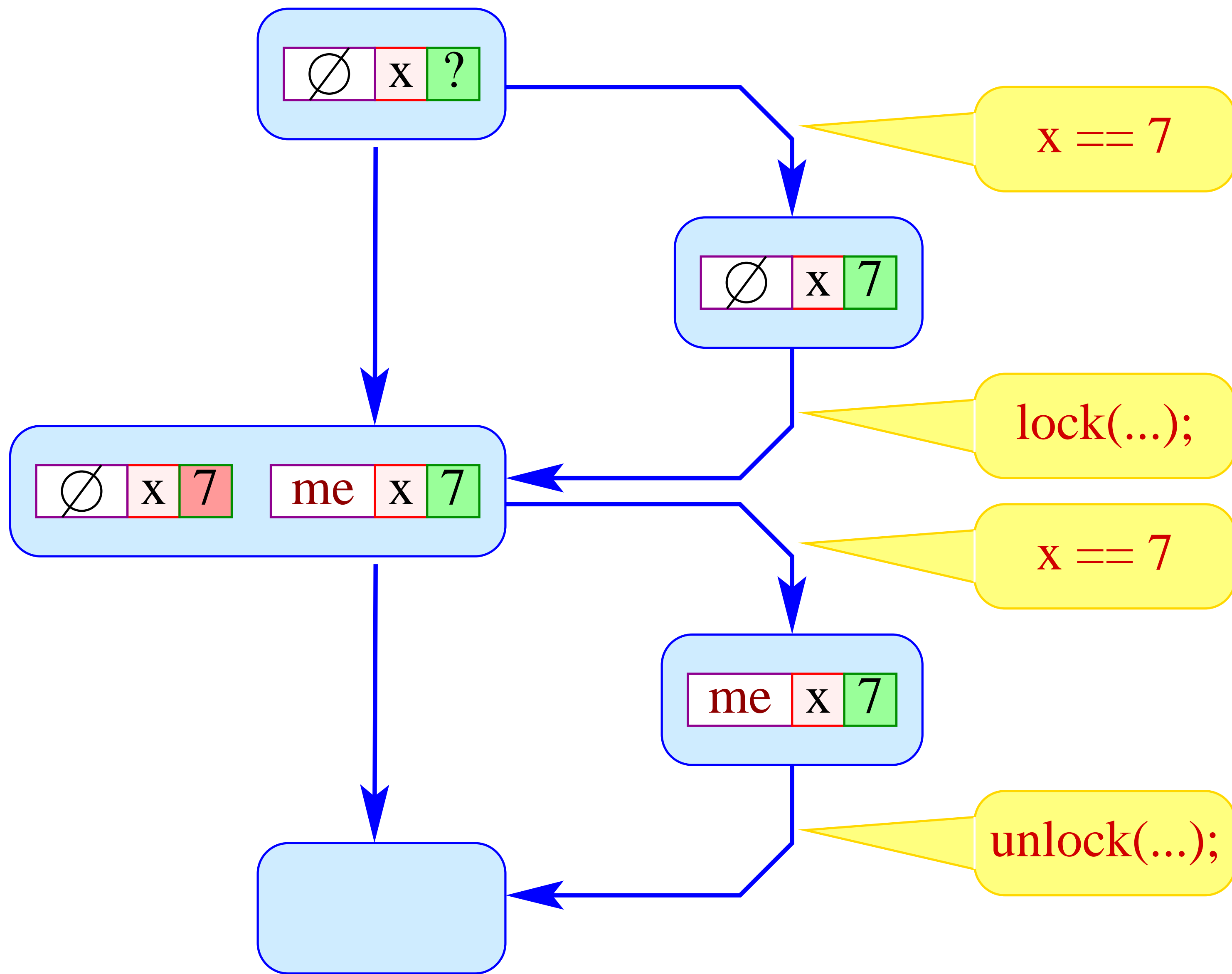
- Track each possibly hold set of locks separately;
- Join variable assignments only relative to a possible set of locks :-)

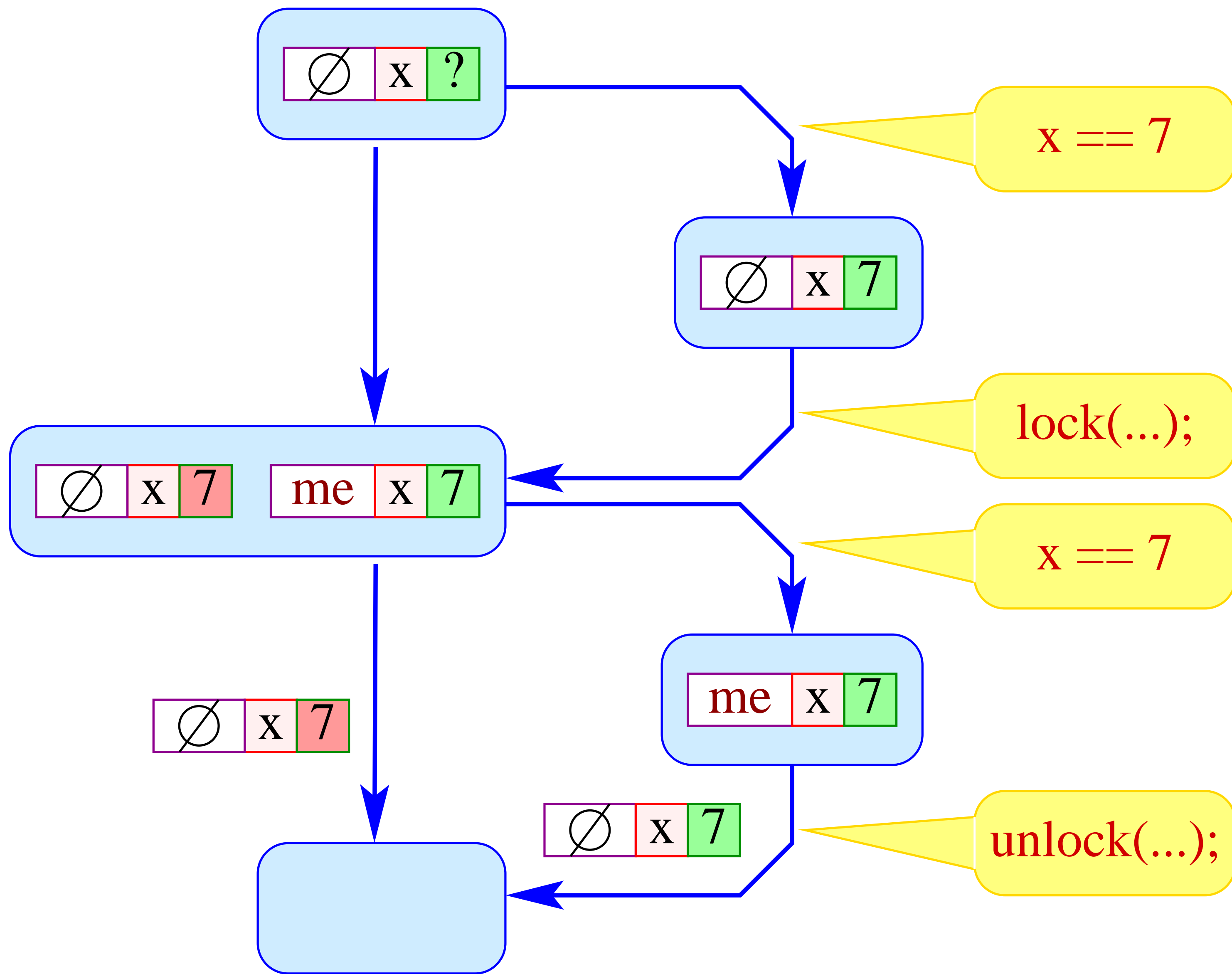

∅
x
?
x == 7
lock(...);
x == 7
unlock(...);

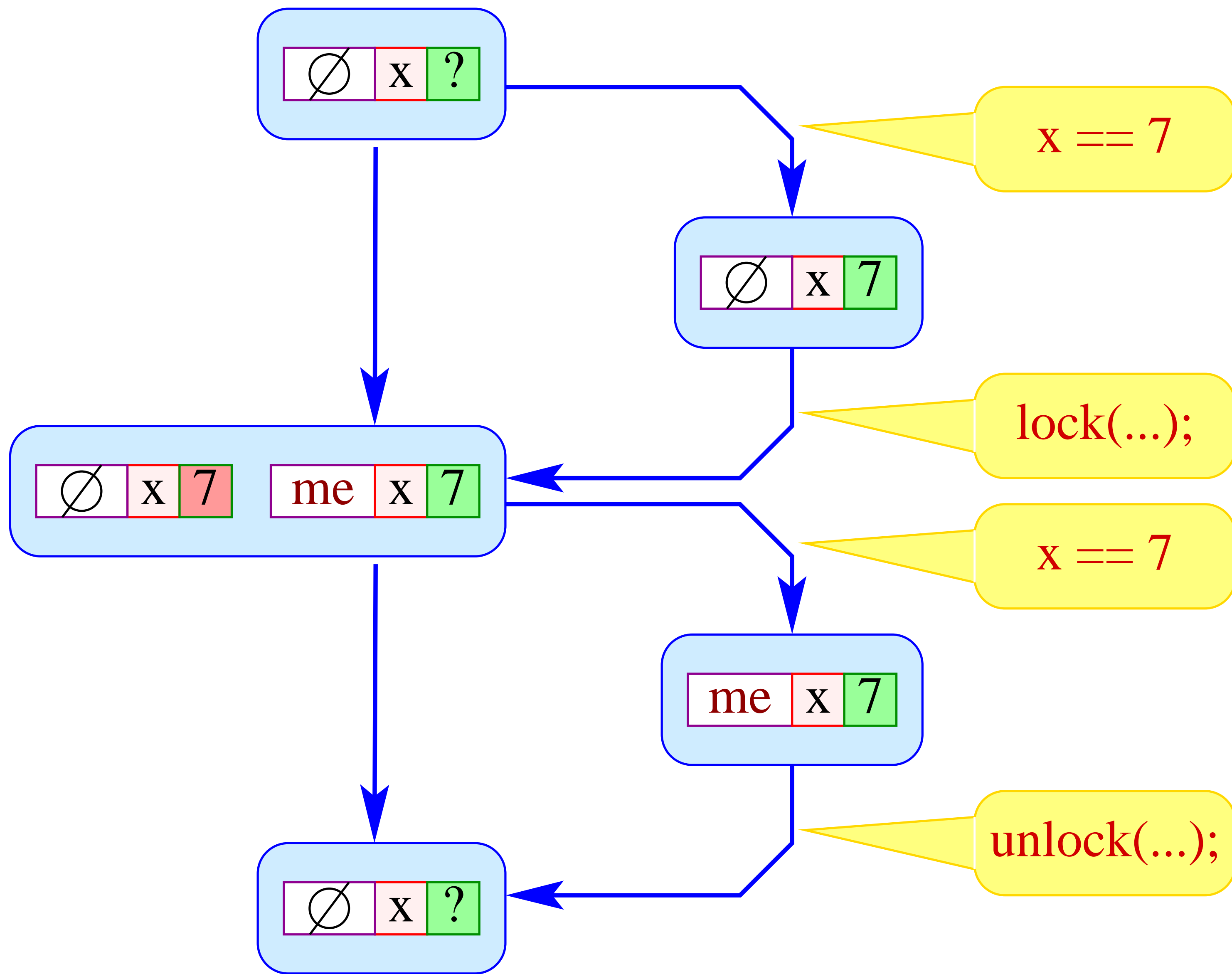

∅
x
?
x == 7
∅
x
7
lock(...);
x == 7
unlock(...);


∅ x ?
x == 7
∅ x 7
∅ x 7
me x 7
lock(...);
x == 7
unlock(...);


∅ x ?
x == 7
∅ x 7
lock(...);
∅ x 7
me x 7
x == 7
unlock(...);


∅ x ?
x == 7
∅ x 7
lock(...);
∅ x 7
me x 7
x == 7
me x 7
unlock(...);


∅ x ?
x == 7
∅ x 7
lock(...);
∅ x 7
me x 7
x == 7
me x 7
∅ x 7
∅ x 7
unlock(...);


∅ x ?
x == 7
∅ x 7
lock(...);
∅ x 7
me x 7
x == 7
me x 7
unlock(...);
∅ x ?

2.2. Tracking Mutex Locks

Sadly enough,

- ... pthread mutex locking may fail;
- ... some code checks whether locking has succeeded.

$\Longrightarrow$ We cannot assume that a lock has been acquired before the check point;

$\Longrightarrow$ We must track variables holding return values of `pthread_mutex_lock()`.

Example:

```
ret = pthread_mutex_lock (Mutex);
while (ret == 4)
    ret = pthread_mutex_lock (Mutex);
if (ret != 0) RetCond = FALSE;
else while (RetCond) {...}
```

3. Results

We obtained ...

- a reasonably efficient analyzer which is able to deal with almost all benchmark applications provided to us by Airbus (varying in size between 10,000 and 60,000 LOC) ...
- whch is reasonably precise to flag few spurious warnings.

Some numbers:

Benchmark	Threads	Base Analysis	
1	3	.30	73831
2	4	1.00	133098
3	0	.28	107066
4	3	.15	52419
5	3	1.33	157010
6	7	18.15	248761
7	25	6.21	447505

Benchmark	Threads	Checked Locks		
1	3	.36	77001	8
2	4	1.30	186806	37
3	0	.24	88172	0
4	3	.20	61302	6
5	3	25.30	682210	36
6	7	2.56	291278	13
7	25	4.20	?	?

Benchmark	Threads	Unchecked Locks		
1	3	.32	73116	7
2	4	1.05	150104	29
3	0	.19	78002	0
4	3	.20	60783	6
5	3	3.48	276745	31
6	7	15.49	716220	13
7	25	?	?	?

The Universal Tiny Problem Solver

Simple Idea:

→ Compute set of reachable configurations exactly!

→ Represent this set as BDD :-)

→ int value == 32 bits :-))
thread pc == int value
configuration == vector of int's :-))

→ Implement Posix thread function calls directly as BDD operations :-)

→ Provide whatever can be compiled into this structure ...

Consequences:

Abstraction from scheduling policies

No Support for:

- recursion;
- dynamic addresses;
- iterated thread creation;
- general multiplication :-(

. . . but:

Instead, we provide:

- non-recursive procedures with reference parameters;
- local variables;
- indexed jumps;
- nested arrays, structs;
- array accesses indexed with iteration variables;
- unknown values :-)

Results:

- We have implemented with various (semi-naive) fixpoint iteration strategies;
- We have experimented with various benchmark programs:

→ a correct use of Hoare monitors (from AIRBUS);

→ a flawed use of Hoare monitors (from AIRBUS);

→ dining philosophers of various sizes;

→ bounded buffers with semaphores;

→ reader–writer locking;

→ Peterson's algo;

→ Bakery algo . . .

- We found the usual results:

  → Flawed programs are often easier to analyze :-)

  → There was no unique best iteration strategy :-(

- The tool, though, was efficient enough to analyze the given benchmark programs — at least for small problem instances :-}

- Abstract interpretation based extra tool is needed to extract the concurrent control protocol out of realistic programs ...