Contracts for correctness

(today *and tomorrow*)

Thomas Gilray
Asst. Prof. @ UAB CS
thomas.gilray.org @tomgilray

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

UNIVERSITY OF MARYLAND

So, what is a ***contract*** anyway?

contract noun

con·tract | \ˈkän-ˌtrakt \

Definition of *contract* (Entry 1 of 3)

1 **a** **:** a binding agreement between two or more persons or parties *especially* **:** one legally enforceable

// If he breaks the *contract*, he'll be sued.

An agreement between multiple parties for mutual benefit.

contract noun

con·tract | \ˈkän-ˌtrakt \

Definition of *contract* (Entry 1 of 3)

1 a : a binding agreement between two or more persons or parties

especially : one legally enforceable

// If he breaks the *contract*, he'll be sued.

The agreement is enforced and violations are blamed on an offending party.

```
    }

    return e;
}
```

A reallocating array<T> class in C++

```
void insert(const T& ele, u64 index = 0)
{
    assert(length >= index);

    if (length+1 > buff_length)
    {
        // reallocate buffer
        T* oldbuff = buff;

        buff_length *= 2;
        buff = allocator.alloc(buff_length);

        // copy old data
        for (u64 i = 0; i < length; ++i)
        {
```

```
    }

    return e;
}
```

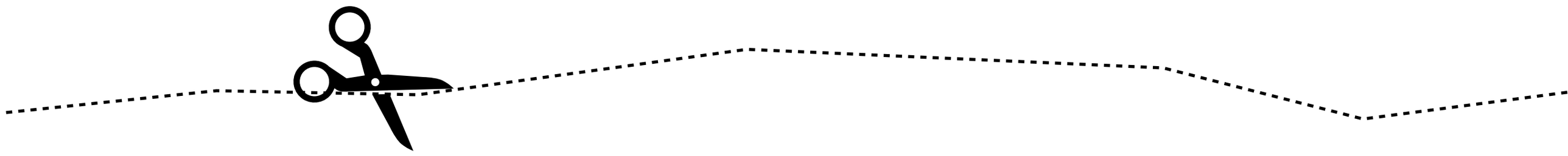
A reallocating array<T> class in C++

```
void insert(const T& ele, u64 index = 0)
{
    assert(length >= index);

    if (length+1 > buff_length)
    {
        // reallocate buffer
        T* oldbuff = buff;

        buff_length *= 2;
        buff = allocator.alloc(buff_length);

        // copy old data
        for (u64 i = 0; i < length; ++i)
        {
```

```
    }

    return e;
}
```

A reallocating array<T> class in C++

```
void insert(const T& ele, u64 index = 0)
{
    // Precondition:
    assert(length >= index);
    assert(length <= buff_length);

    // ... insert, possible reallocation ...
```



```
    // Postcondition:
    assert(length <= buff_length);
}
```

Meyer's "Design by Contract"

Implemented in Meyer's **Eiffel** programming language,
a typed, object-oriented language with contracts at the center.

"A contract carries mutual **obligations** and **benefits**."

"Design by contract". **Bertrand Meyer. 1986.**

Applying “Design by Contract”

```
put_child (new: NODE) is
  require
    new /= Void
  do

    -- Insertion algorithm

  ensure
    new.parent = Current
    child_count = old child_count + 1
  end
```

Precondition

Postcondition

“Applying design by contract”. **Bertrand Meyer. 1992.**

Preconditions

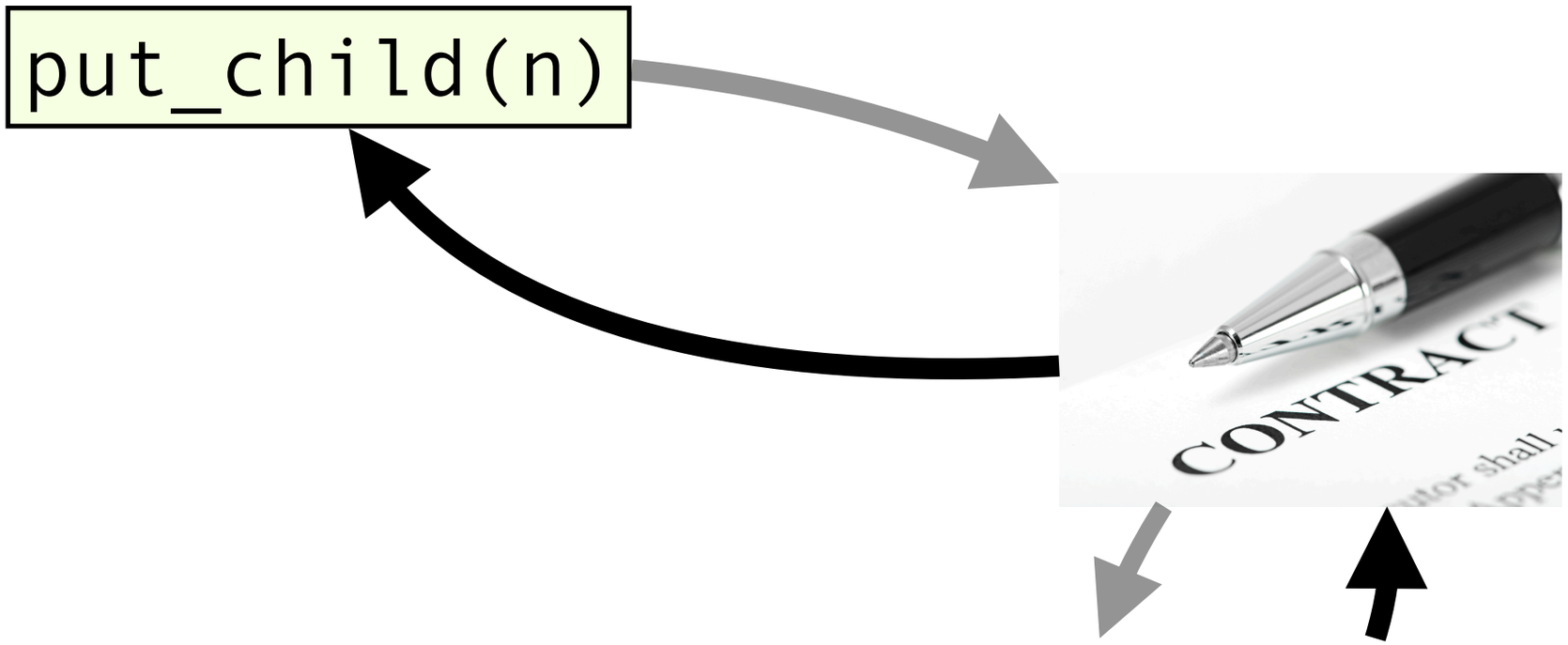
To call `put_child`, calling code must satisfy its **obligations**

`put_child(n)`


```
put_child (new: NODE) is
  require
    new /= Void
  do

    -- Insertion algorithm

  ensure
    new.parent = Current
    child_count = old child_count + 1
  end
```

Postconditions

To return `put_child`, must ensure it provides **benefits**




```
put_child (new: NODE) is
  require
    new /= Void
  do

    -- Insertion algorithm

  ensure
    new.parent = Current
    child_count = old child_count + 1
  end
```

If client **breaks** contract,

`put_child` is not obligated to provide benefits

`put_child(Void)`


```
put_child (new: NODE) is
  require
    new /= Void
  do

    -- Insertion algorithm

  ensure
    new.parent = Current
    child_count = old child_count + 1
  end
```

If client **breaks** contract,

`put_child` is not obligated to provide benefits

`put_child(Void)`

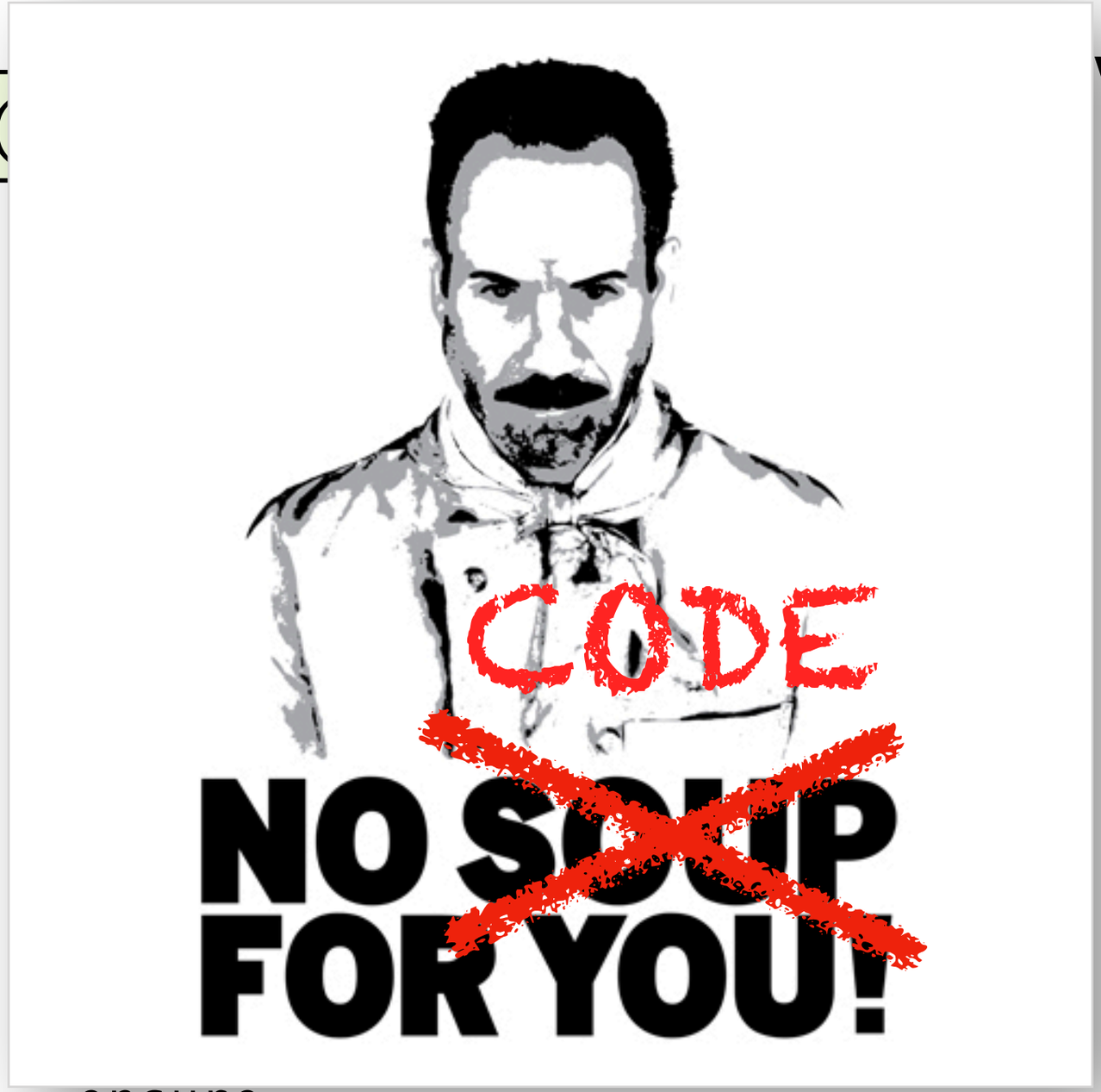
`not (Void /= Void)` 


```
put_child (new: NODE) is
  require
    new /= Void
  do

    -- Insertion algorithm

  ensure
    new.parent = Current
    child_count = old child_count + 1
  end
```

If client **breaks** contract,

`put_child` is not obligated to provide benefits

```
put_child(
```




```
Void)
ensure
  new.parent = Current
  child_count = old child_count + 1
end
```

Contracts are a ***linguistic mechanism***

implemented as a *built-in feature* of the language, using *source-to-source translation*, or *using a macro system*.

factorial in Java (e.g., using jContract)

```
/**
 *   @pre n >= 0
 *   @post return >= 1
 */
public static int fact(int n) {
    if (n <= 1) return 1;
    else return n * fact(n-1);
}
```

factorial in Java (e.g., using jContract)

```
public static int fact(int n) {
    assert n >= 0;
    if (n <= 1) {
        assert 1 >= 1;
        return 1;
    } else {
        int rv = n * fact(n-1);
        assert rv >= 1;
        return rv;
    }
}
```

The contract bakes dynamic checks into the source code, executed at every evaluation of fact(n)!

“Contracts for higher-order functions”. **Findler, Felleisen. 2002.**

Contracts.js

“Hygienic Macros for JavaScript”. **Tim Disney. 2015**

Contracts.js

Try it out online!

http://www.contractsjs.org/#/examples

"Hygienic Macros for JavaScript". **Tim Disney. 2015**

```
// [number] -> [string]
function array_numtostr(arr) {
    assert(arr instanceof Array);

    var str_arr = [];
    for (var i = 0; i < arr.length; ++i)
        str_arr.push(arr[i]+"");

    assert(str_arr instanceof Array);
    return str_arr
}
```

```
// [number] -> [string]
function array_numtostr(arr) {
    assert(arr instanceof Array
        && arr.reduce((a,n)=>(typeof n)
=="number" && a, true));

    var str_arr = [];
    for (var i = 0; i < arr.length; ++i)
        str_arr.push(arr[i]+"");

    assert(str_arr instanceof Array
        && str_arr.reduce((a,s)=>(typeof s)
=="string" && a, true));
    return str_arr
}
```

```
➡ @ ([...Num]) -> [...Str]
  function array_numtostr(arr) {

      var str_arr = [];
      for (var i = 0; i < arr.length; ++i)
          str_arr.push(arr[i]+"");

      return str_arr
  }
```

contracts.js's **@** macro allows the programmer to associate a function contract with `array_numtostr`

```
@ ([...Num], (Num)->Str) -> [...Str]
function array_numtostr(arr, format) {

    var str_arr = [];
    for (var i = 0; i < arr.length; ++i)
        str_arr.push(format(arr[i]));

    return str_arr
}

array_numtostr([14,18],
    (n) => “0x"+n.toString(16))

// => ["0xe","0x12"]
```

We can check that `arr` is an array, and that its elements are numbers…

```
@ ([...Num], (Num)->Str) -> [...Str]
function array_numtostr(arr, format) {
  assert(…??…);
```

…but how can we check that `format` satisfies `(Num)->Str` *before* array_numtostr is evaluated?

`format` **is a first-class function—a behavioral value!**

Contracts on behavioral values are ***delayed.***

The contract `array_numtostr` requires on its argument `format` is enforced in the same way as the contract on `array_numtostr`!

```
@ ([...Num], (Num)->Str) -> [...Str]
function array_numtostr(arr, format) {

    var str_arr = [];
    for (var i = 0; i < arr.length; ++i)
        str_arr.push(format(arr[i]));

    return str_arr
}

array_numtostr([14,18], (n) => n)

// => ["0xe","0x12"]
```

```
Error: array_numtostr: contract violation
expected: Str
given: 14
in: the return of
    the 2nd argument of
    ([....Num], (Num) -> Str) -> [....Str]
function array_numtostr guarded at line: 4
blaming: (calling context for array_numtostr)
```

Higher-order contract systems track program labels alongside contracts to ***properly assign blame*** when failure occurs.

“Correct blame for contracts”. **Dimoulas. 2011.**

```
@ (a: [...Num], f: (Num)->Str)
    -> r: [...Str] | a.length == r.length
function array_numtostr(arr, format) {

    var str_arr = [];
    for (var i = 0; i < arr.length; ++i)
        str_arr.push(format(arr[i]));

    return str_arr
}

array_numtostr([14,18],
    (n) => "0x"+n.toString(16))

// => ["0xe","0x12"]
```

Behavioral contracts

- **Same expressivity** as the host programming language
- **Reports actual/observed errors** & witnesses to errors
- Can add **significant run-time overhead**, breaks tail calls
- Error discovery is **delayed until runtime**

Static typechecking

- **Separate type language** with a static semantics
- Reports **potential errors** in abstract terms
- Produces fast code **without run-time monitoring** of types
- Potential failures are **discovered ahead of time**

The future of dynamically enforced contracts is ***static verification***!


Formal verification
of program properties
Dynamic enforcement of program
properties with behavioral contracts

```
f((n)=>n+1, 0)
f((n)=>n*n, 2)
f(h, 0)
```

```
@ ((Num)->Num, Num) -> Num
function f(g, x) {
    // y = ...
    return g(y)
}
```

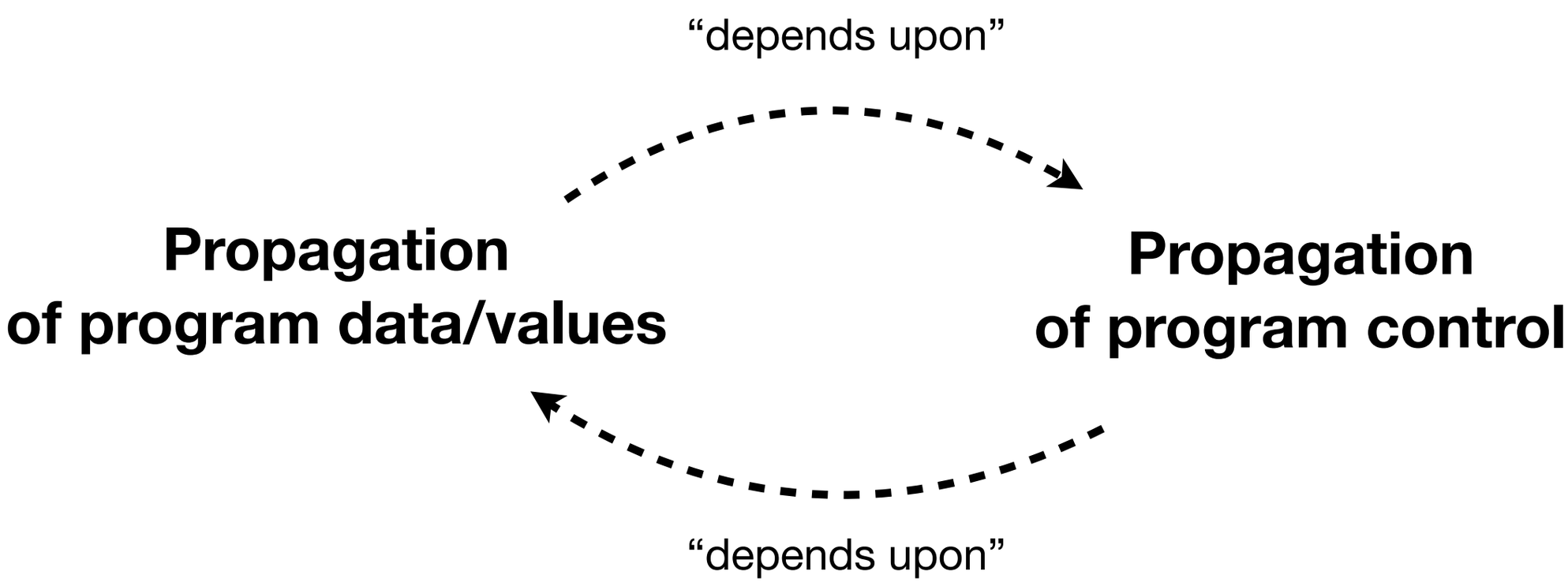
...and on the callers of `f`.

...depends on data-flow to `g`...

Control-flow at `g(y)` here...

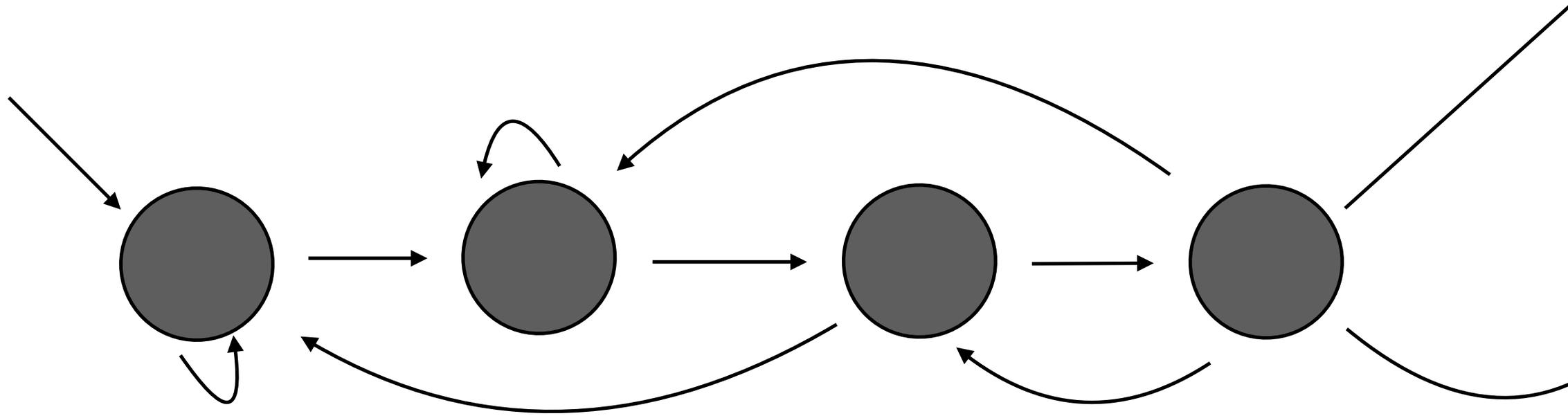
This is called the

higher-order control-flow problem



and to tackle this problem, we use

AI



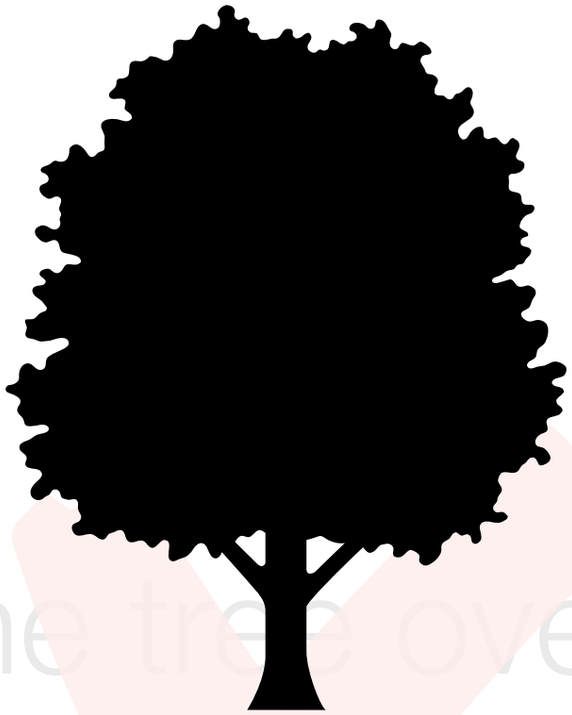
Abstract Interpretation

“Do you know the way?”

“Sure, you take a left just past that ***<u>tree</u>*** there.”

“*The little poplar just there?*”

Ah no, the tree over *there*,
the ***one with 51 branches***
& 8206 leaves.

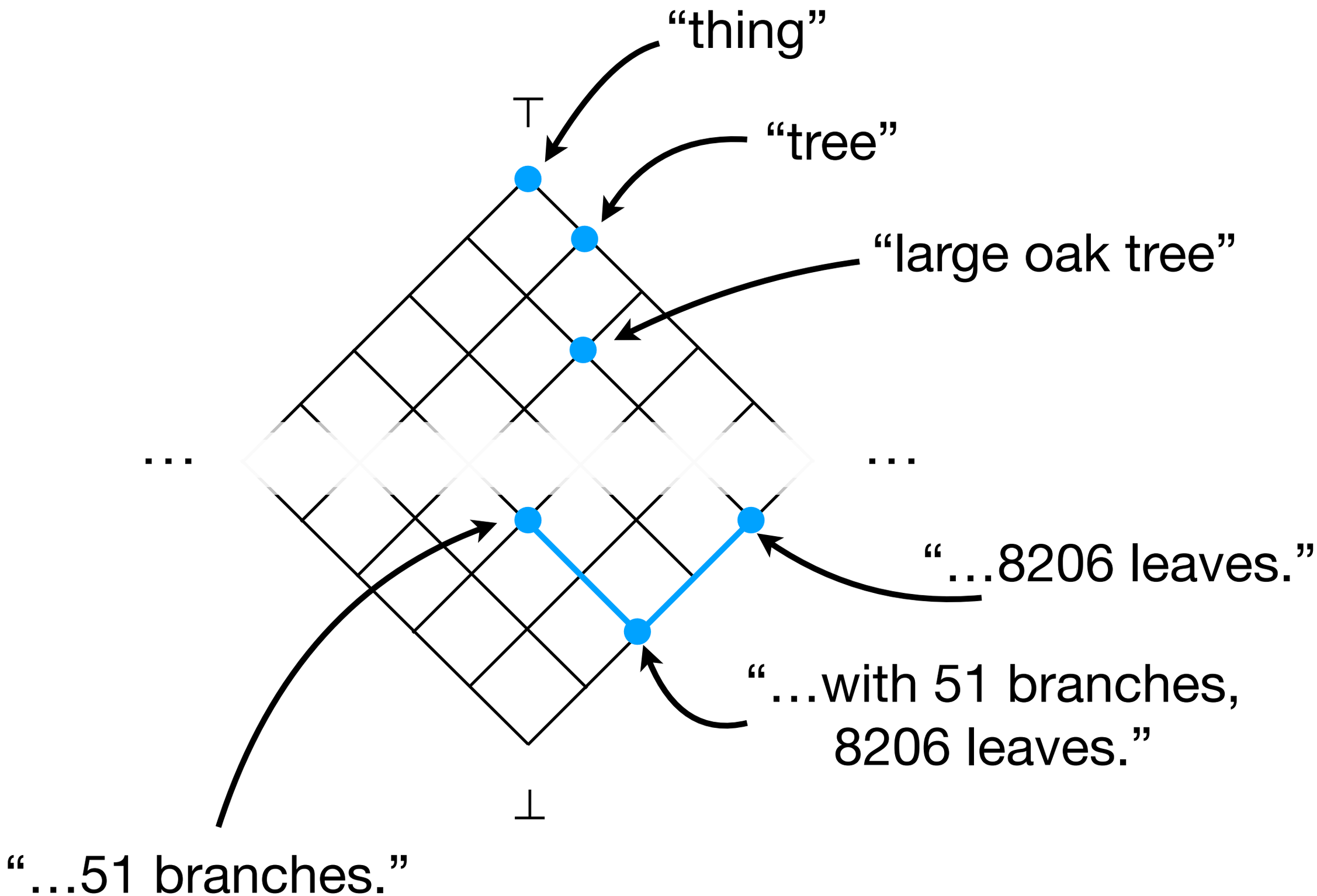


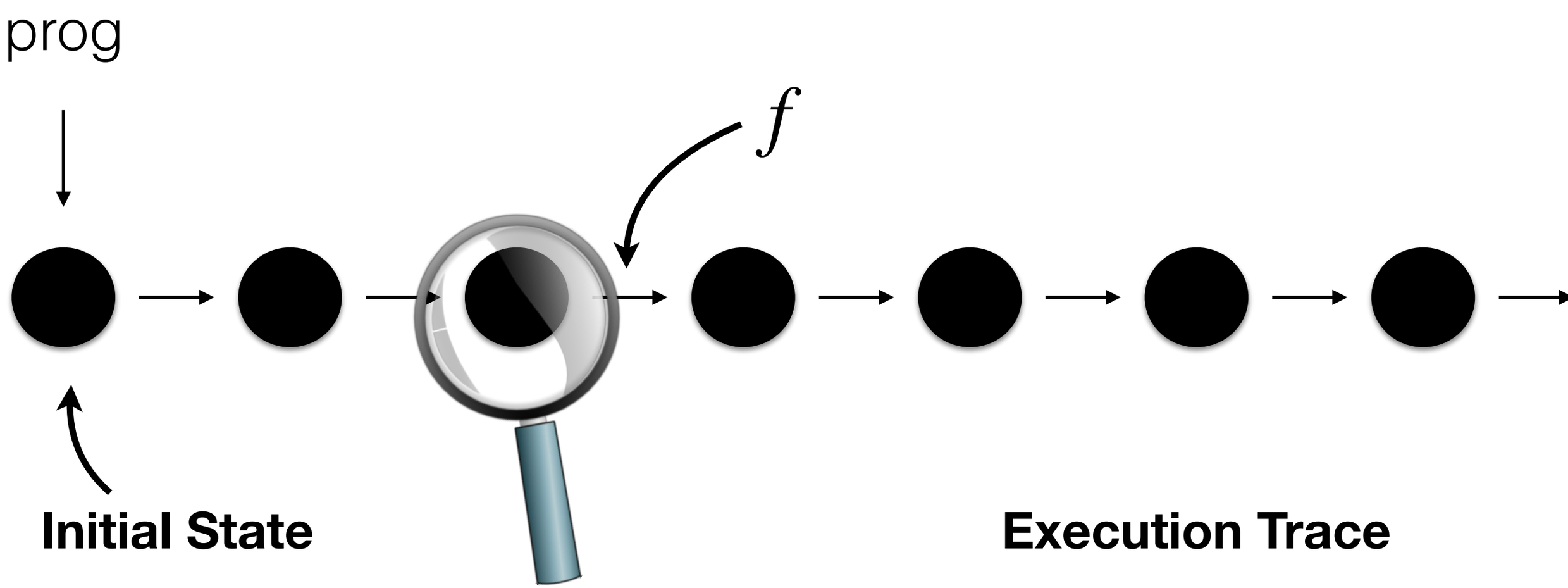
Ah no, the tree over *there*,

the ***one with 51 branches***

& 8200 leaves.

Ah no, the ***large oak tree*** there.


“thing”
⊤
“tree”
“large oak tree”
…
…
“…8206 leaves.”
“…with 51 branches,
8206 leaves.”
⊥
“…51 branches.”



States may contain:

- the program counter,
- a binding environment,
- a model of the heap,
- a model of the stack,

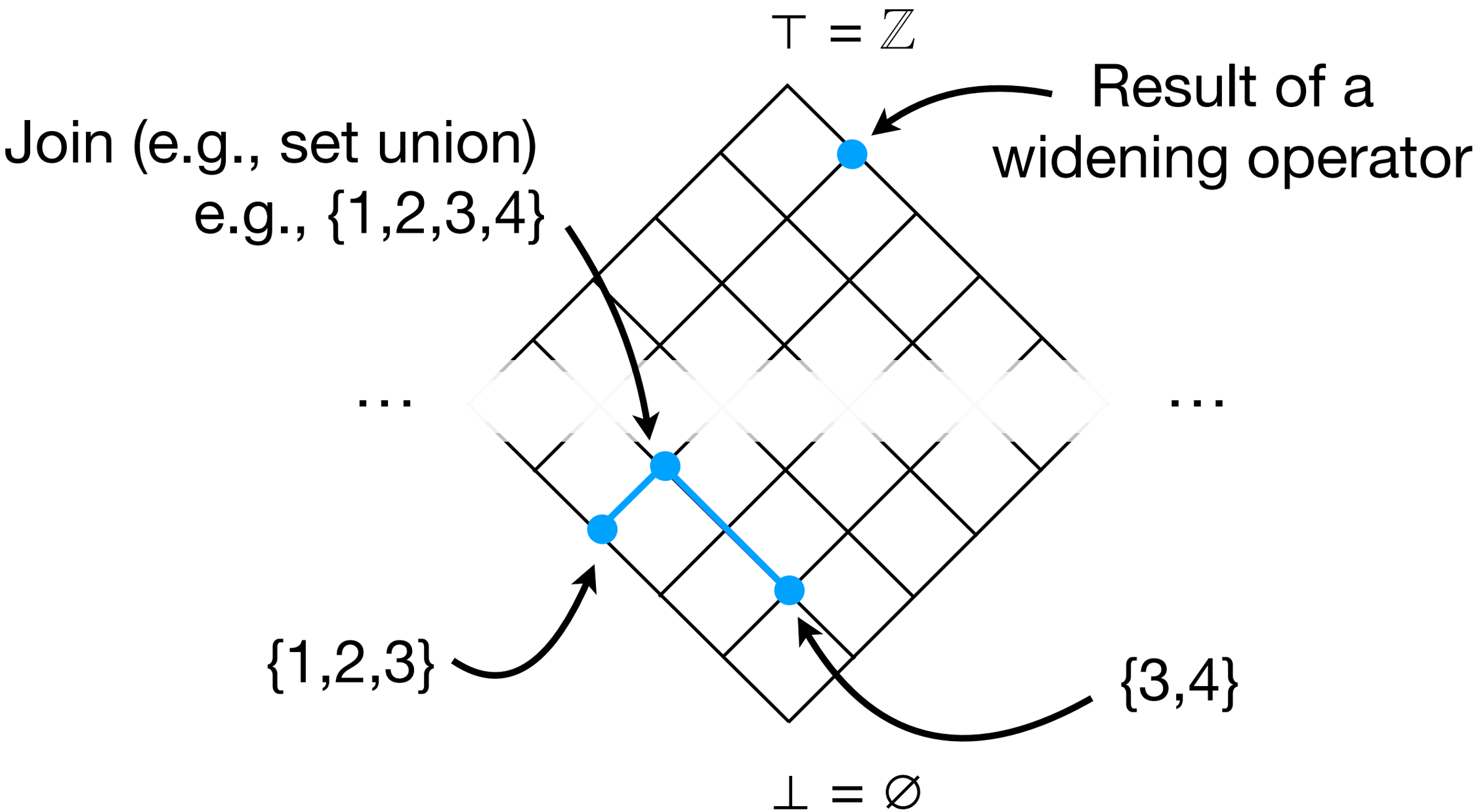
etc…

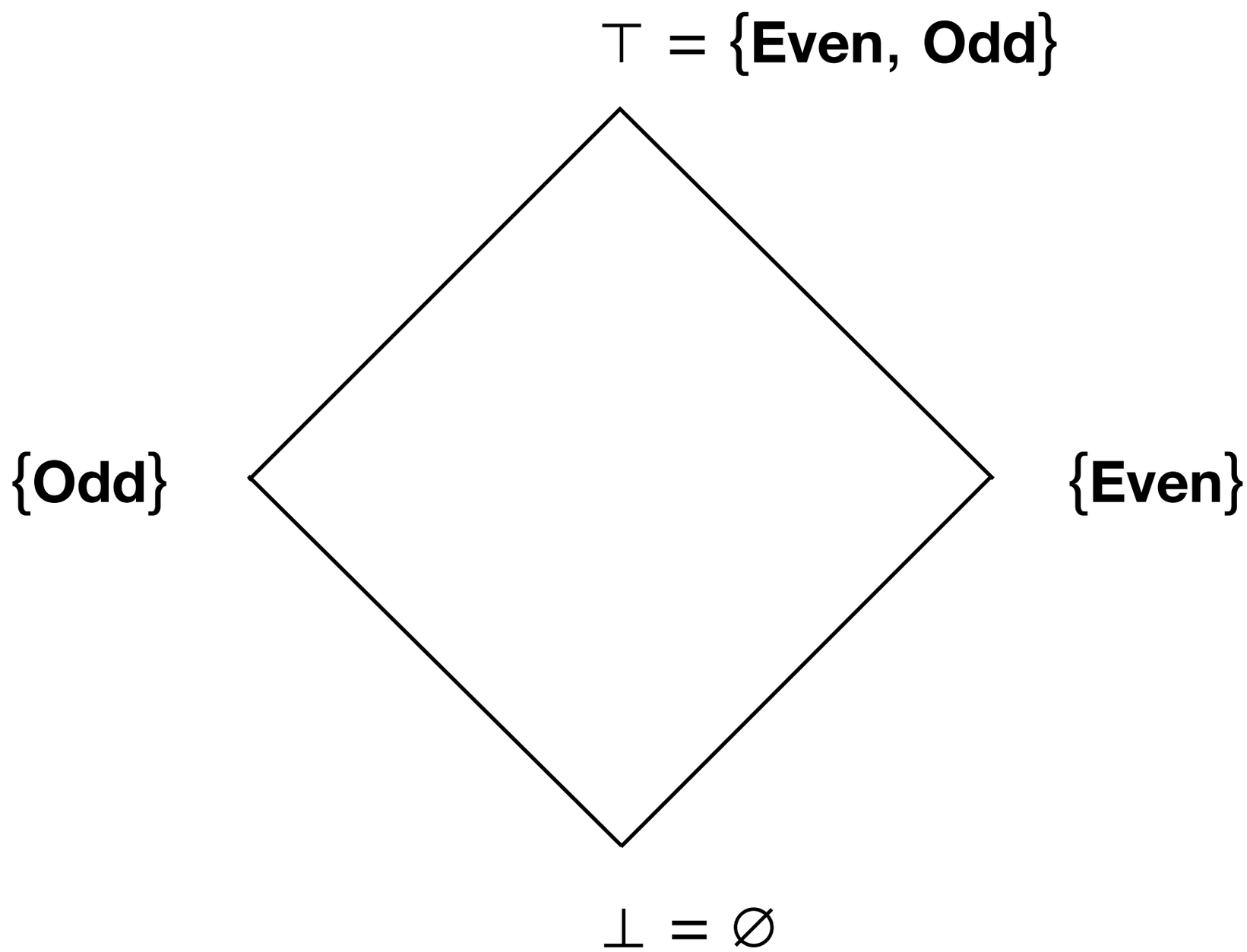
Plotkin (1981), Tarski (1955)

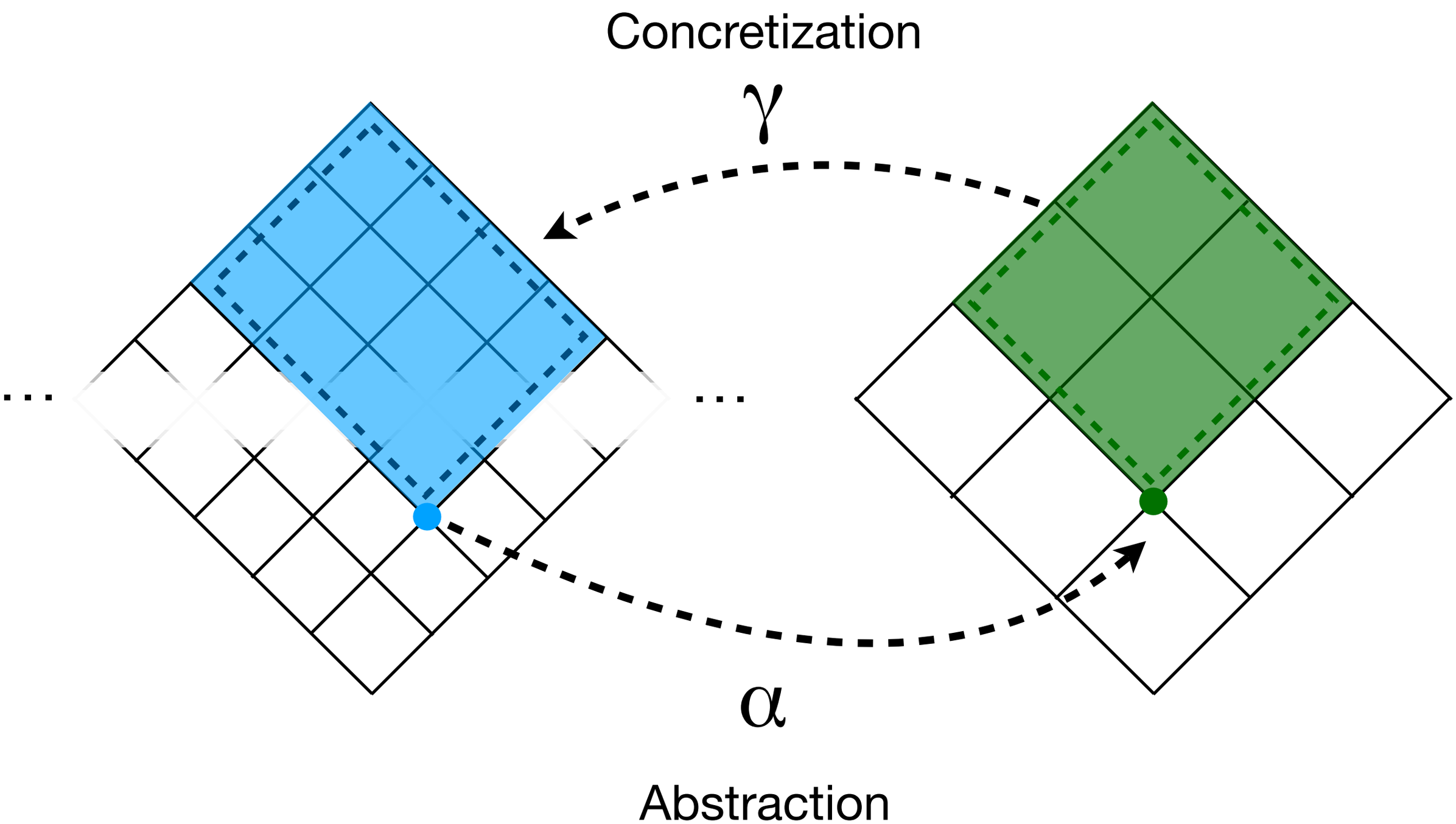
{…,-3,-2,-1,0,1,2,3,…}

```
// @post !isEven(return)
public static int nextOdd(int x) {
    if (isEven(x))
        return x+1;
    else return x+2;
}
```

{…,5,−3,~~1~~**1**,1,3,5,…}


$\top = \mathbb{Z}$
Result of a widening operator
Join (e.g., set union)
e.g., {1,2,3,4}
...
...
{1,2,3}
{3,4}
$\bot = \varnothing$

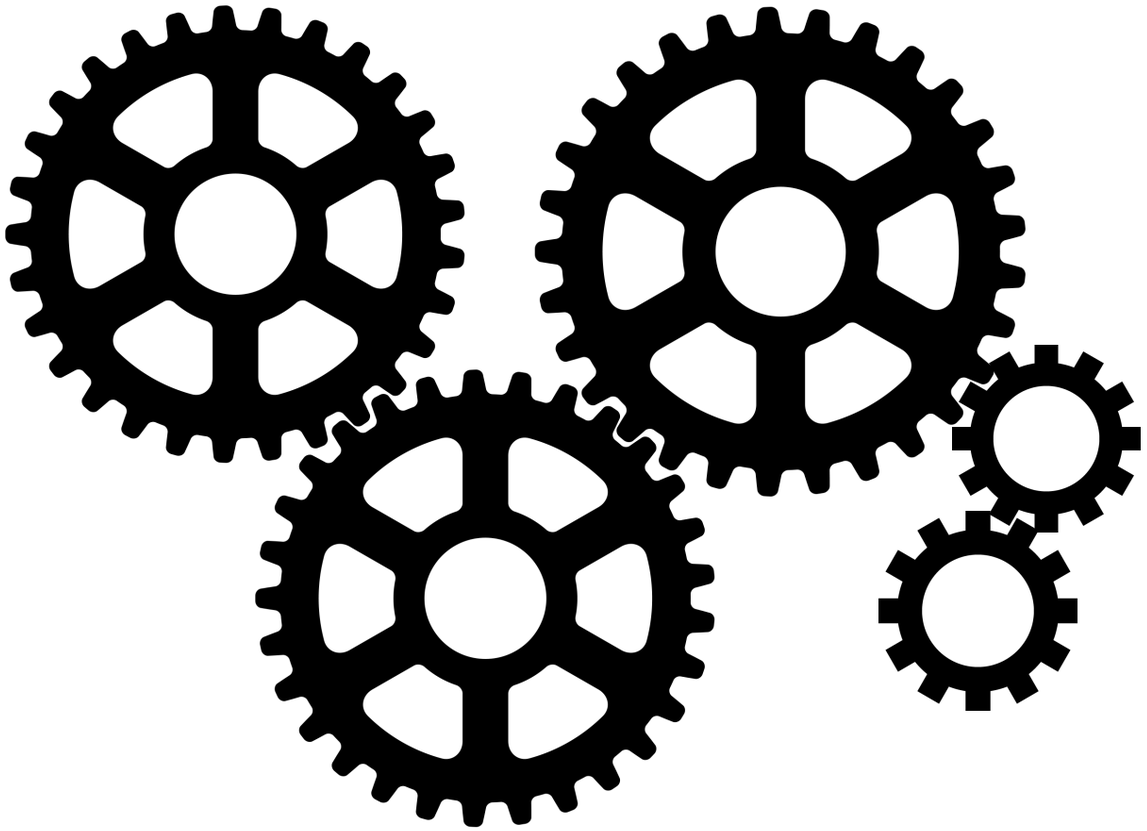

$\top$ = {Even, Odd}
{Odd}
{Even}
$\bot$ = $\varnothing$



Galois connection

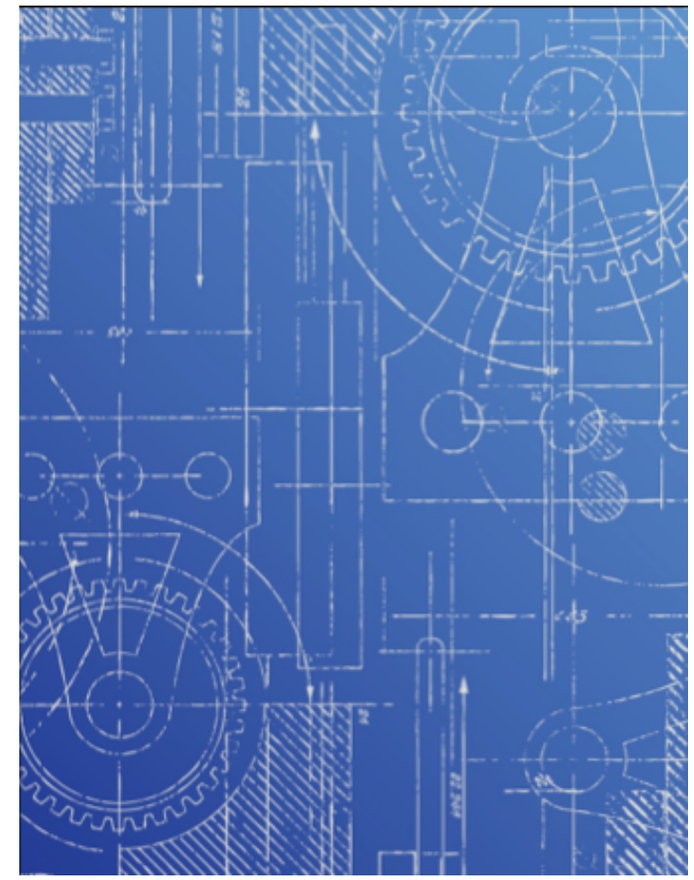
$$\alpha(c) \sqsubseteq a \iff c \sqsubseteq \gamma(a)$$

Concrete Interpreter

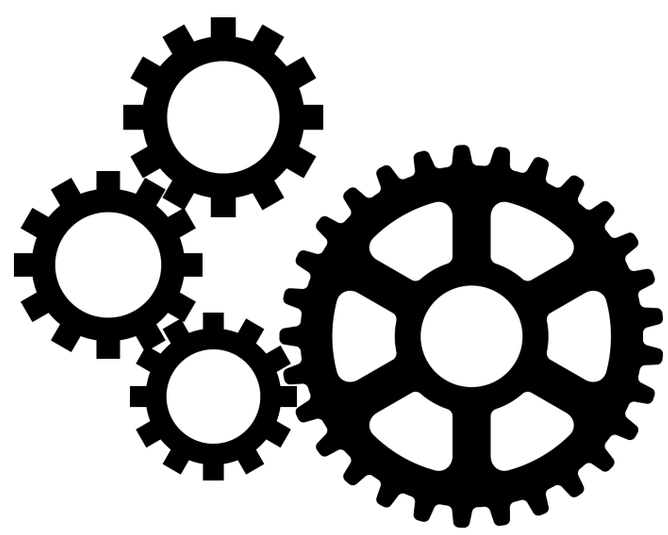


+

Abstraction Specification

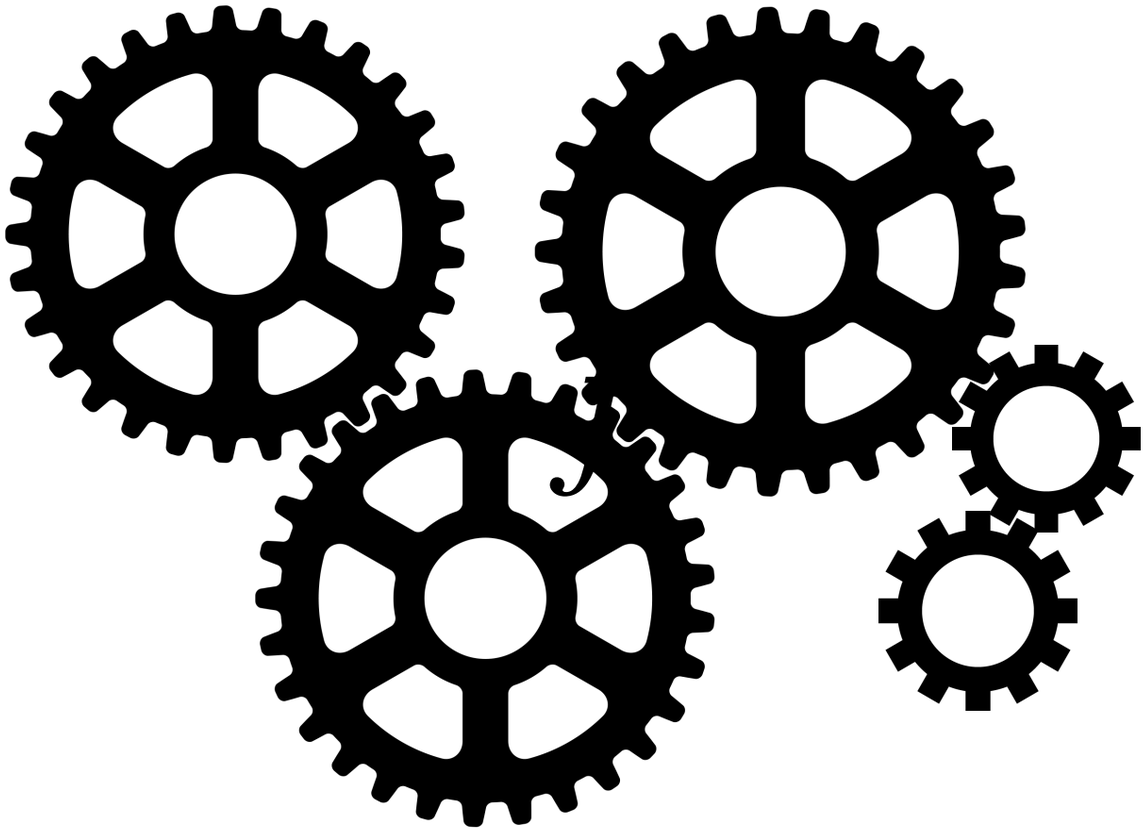


=



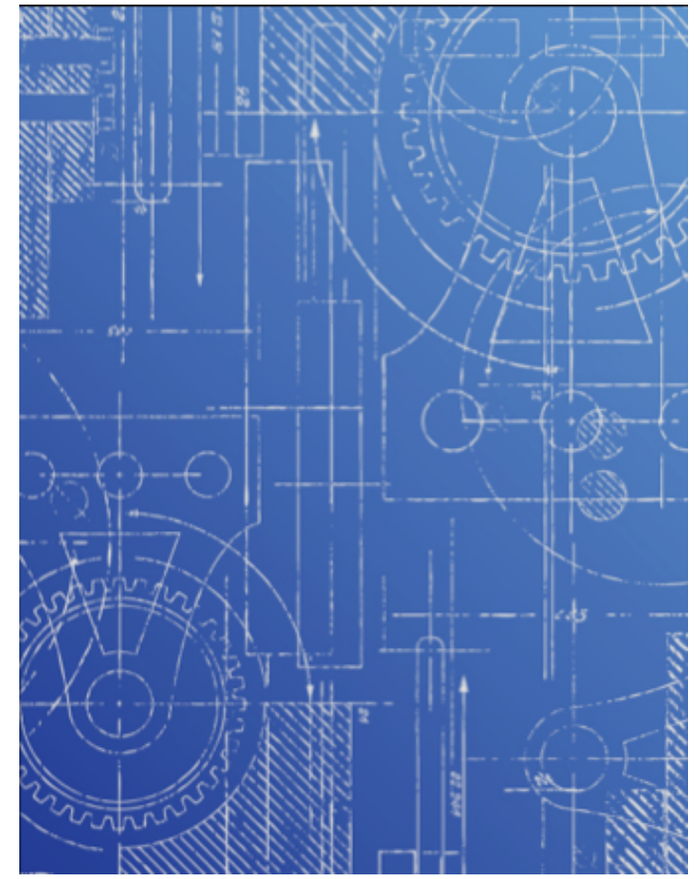
Abstract Interpreter

Concrete Interpreter

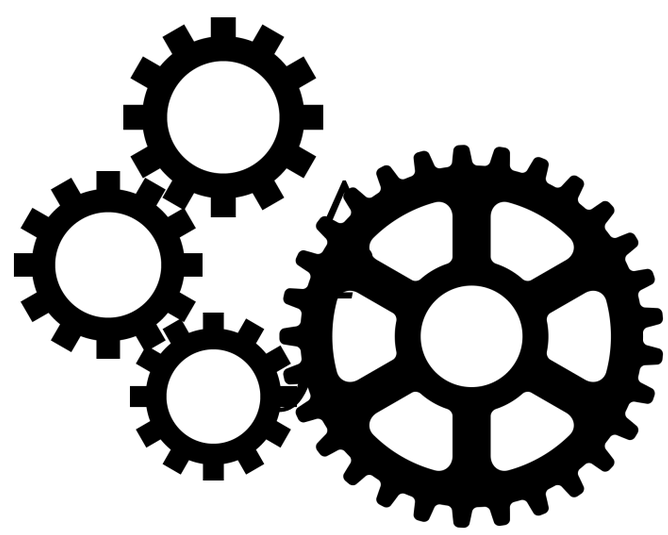


+

Abstraction Specification



=



Abstract Interpreter

The calculational approach to Abstract Interpretation

$$\hat{f} = \alpha \circ f \circ \gamma$$

Cousot, Cousot (1976,1977,1979)

```
public static int nextOdd(int x) {
    if (isEven(x))
        return x+1;
    else return x+2;
}
```

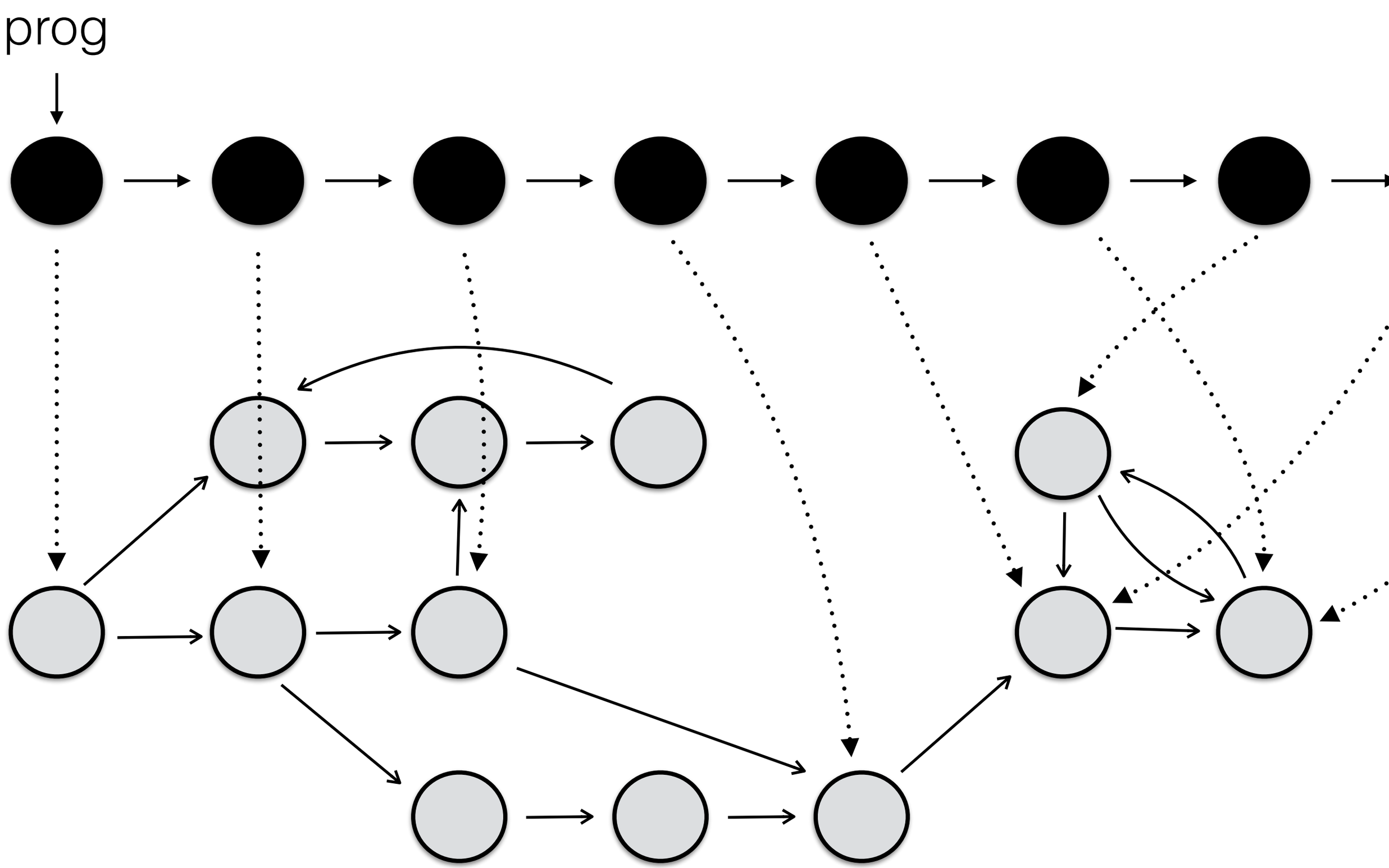
$$(+) = \alpha \circ (\hat{+}) \circ \gamma$$

Calculated abstract implementation

$$\{\mathbf{Odd}\} \;\hat{+}\; \{\mathbf{Odd}\} = \{\mathbf{Even}\}$$

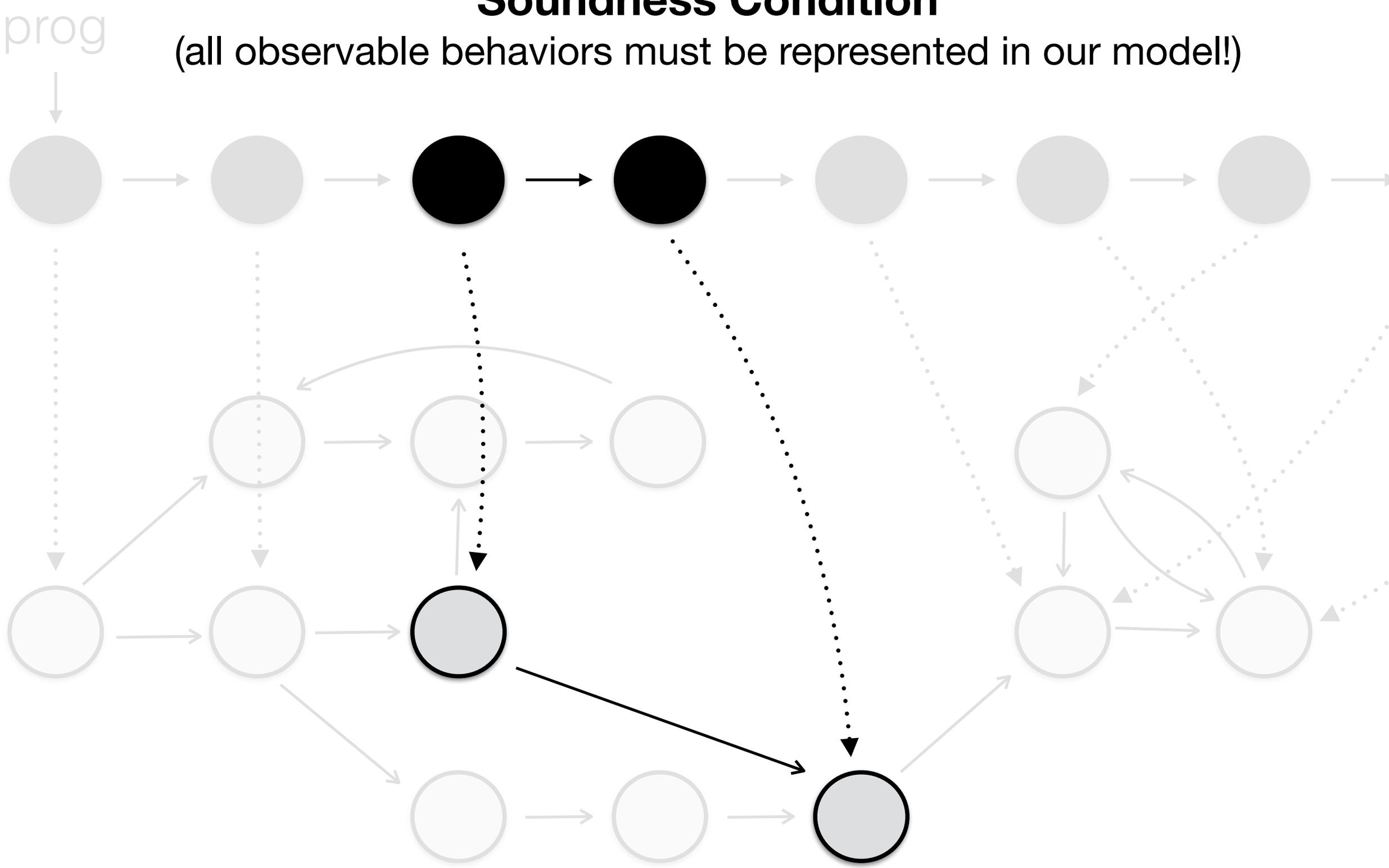
$$\gamma \qquad\qquad\qquad\qquad \alpha$$

$$\{\ldots,\mathbf{-3,-1,1,3},\ldots\} + \{\ldots,\mathbf{-3,-1,1,3},\ldots\} = \{\ldots,\mathbf{-6,-4,-2,0,2,4,6},\ldots\}$$


prog

Soundness Condition

(all observable behaviors must be represented in our model!)



Abstract Interpretation

+

Symbolic Execution

```
    }

    return e;
}


void insert(const T& ele, s64 index = 0)
{
    // Precondition:
    assert(length >= index);

    // Possible reallocation, shift-back

    // Placement-new a T at index
    new (&buff[index]) T(ele);

    // Postcondition:
    assert(length <= buff_length);
}
```

```
    }

    return e;
}


void insert(const T& ele, s64 index = 0)
{
    // Precondition:
    if (!(length >= index))
        err("Assert failed.");

    // Possible reallocation, shift-back

    // Placement-new a T at index
    new (&buff[index]) T(ele);

    // Postcondition:
    if (!(length <= buff_length))
        err("Assert failed.");
}
```

```
    }

    return e;
}
```

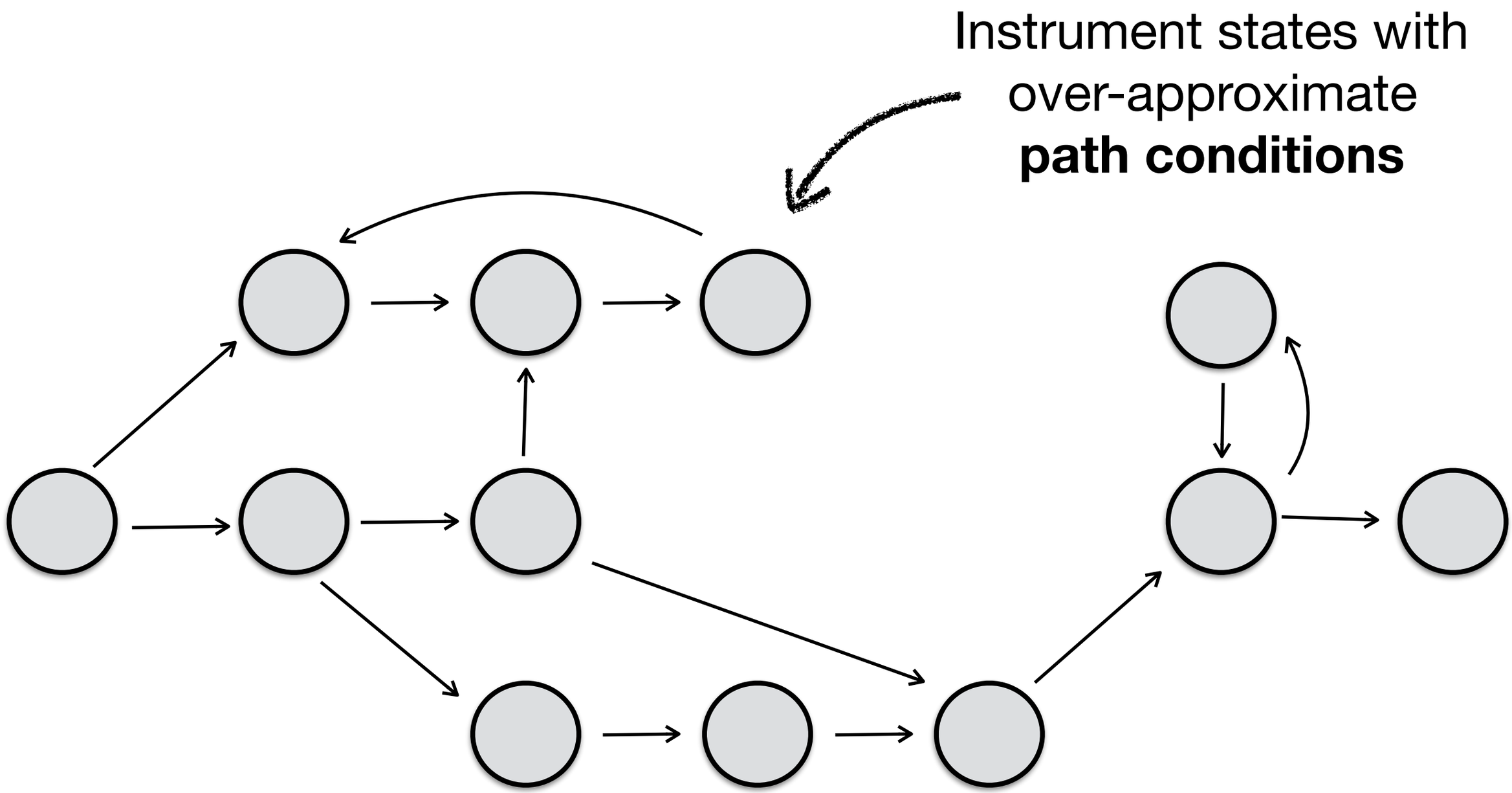
this $= \alpha$ **ele** $= \beta$ **index** $= \gamma$

```
void insert(const T& ele, s64 index = 0)
{
    // Precondition:
    if (!(length >= index))
        err("Assert failed.");

    // Possible reallocation, shift-back

    // Placement-new a T at index
    new (&buff[index]) T(ele);

    // Postcondition:
    if (!(length <= buff_length))
        err("Assert failed.");
}
```

α.**length** $< \gamma$

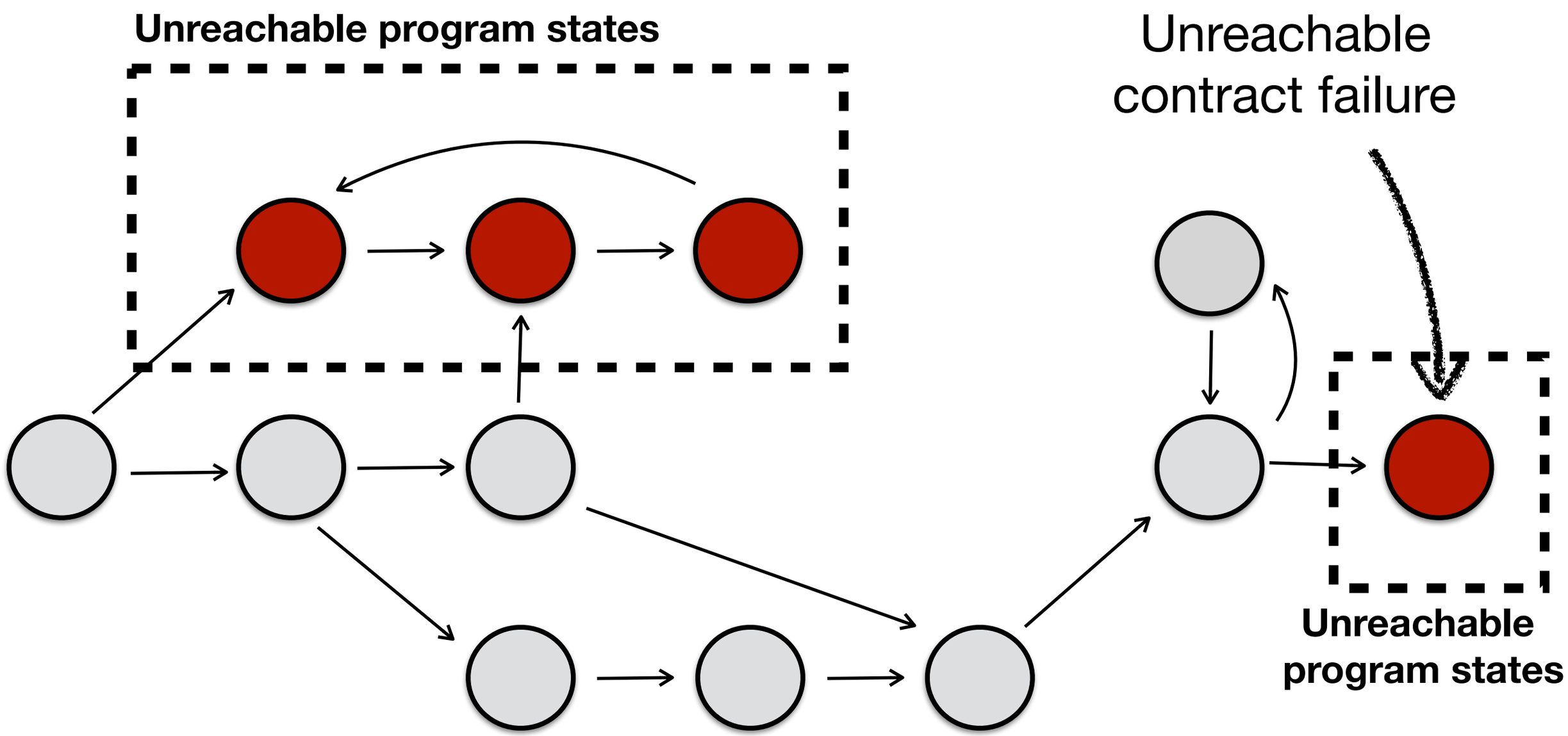
α.**length** $> = \gamma$

Abstract Symbolic Execution



"Soft Contract Verification for Higher-order Stateful Programs".
Nguyễn, Gilray, Tobin-Hochstadt, Van Horn. 2018.

Abstract Symbolic Execution



"Soft Contract Verification for Higher-order Stateful Programs".
Nguyễn, Gilray, Tobin-Hochstadt, Van Horn. 2018.

```
// [number] -> [string]
function array_numtostr(arr) {
    assert(arr instanceof Array
        && arr.reduce((a,n)=>(typeof n)
=="number" && a, true));

    var str_arr = [];
    for (var i = 0; i < arr.length; ++i)
        str_arr.push(arr[i]+"");

    assert(str_arr instanceof Array
        && str_arr.reduce((a,s)=>(typeof s)
=="string" && a, true));
    return str_arr
}
```

```
// [number] -> [string]
function array_numtostr(arr) {
    assert(arr instanceof Array
        && arr.reduce((a,n)=>(typeof n)
=="number" && a, true));

    var str_arr = [];
    for (var i = 0; i < arr.length; ++i)
        str_arr.push(arr[i]+"");

    assert(str_arr instanceof Array
        && str_arr.reduce((a,s)=>(typeof s)
=="string" && a, true));
    return str_arr
}
```

Thanks

- Contracts are a ***linguistic mechanism*** for embedding ***dynamic monitors to enforce program correctness***.
- This gives a precise run-time bound, ***defining correct behavior***.
- Unfortunately, this adds ***significant run-time overhead*** and ***delays error discovery*** until it may be too late to fix.
- ***Abstract symbolic execution*** (AI+SE) gives us a way to verify contracts ***on a best-effort basis*** where a failure to verify a contract ***degrades gracefully*** to run-time monitoring.