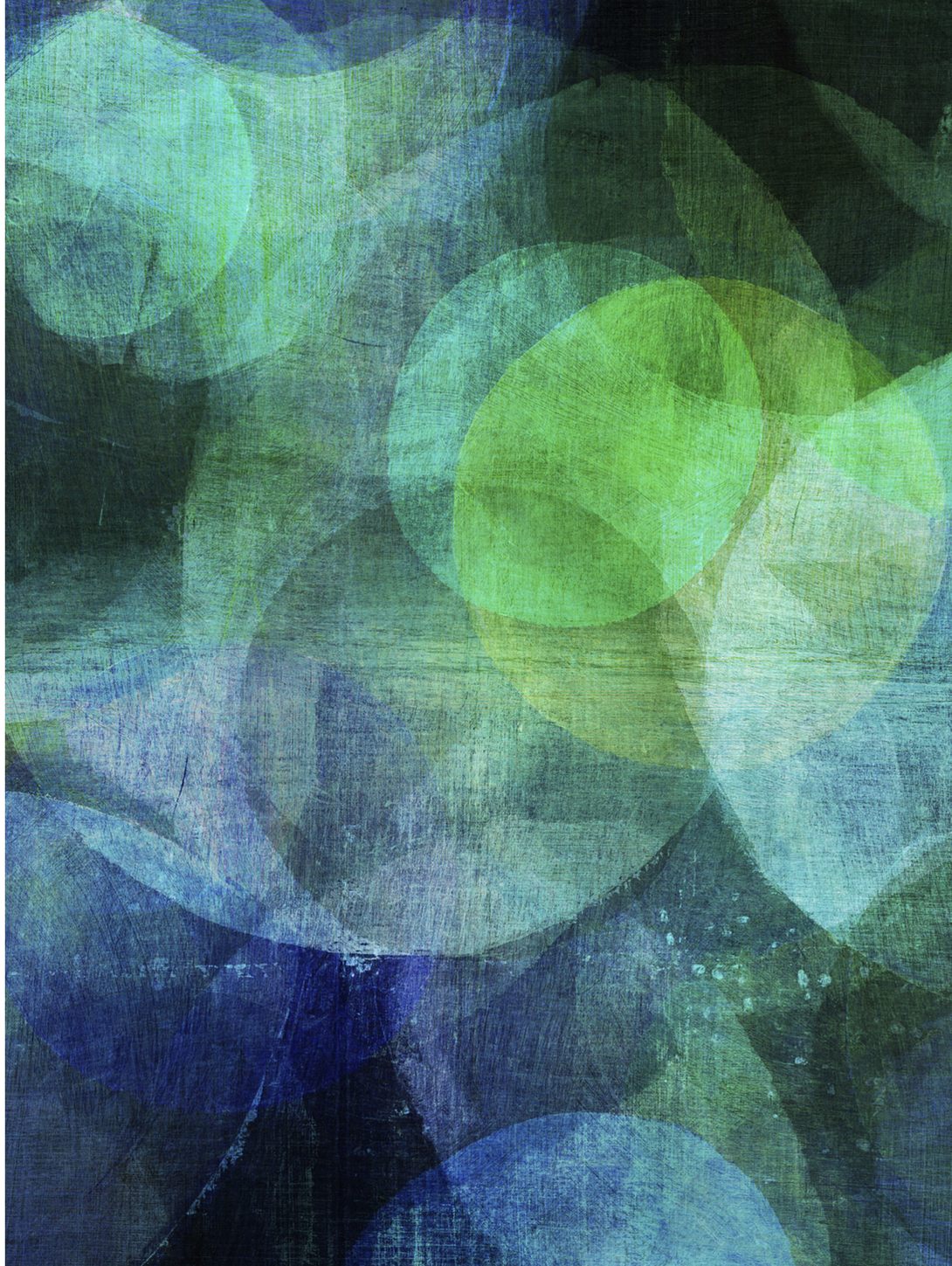
IMPROVING DYNAMIC CODE ANALYSIS BY CODE ABSTRACTION

I. Mastroeni and V. Arceri
Univ. of Verona and Univ. of Venezia

MOTIVATION

Dynamic languages do have string-to-code statements



JAVASCRIPT HAS EVAL ALLOWING EVIL THINGS

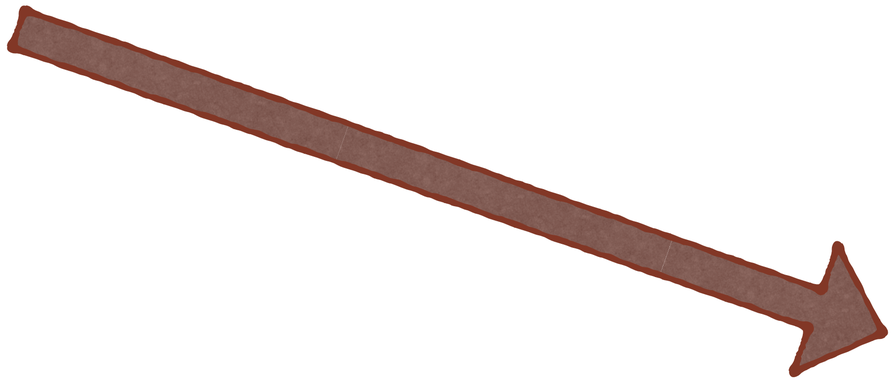
`eval(”a = a + 1”)`

String

Parsing of ”a = a + 1”

`a=a+1`

Code!



```
eval('alert("Your query was ' + document.location.search + '");');
```

XSS, SSJS, CSRF, Code injection...

HOW MUCH IS EVAL POPULAR?

Best practice:

"eval is the most misused feature of JavaScript. Avoid it."

Benevolent code

G. Richards *et al.* *"The eval that men do"*

Malicious code

We have analyzed a data-set of JavaScript malware samples:

57% have at least one eval call

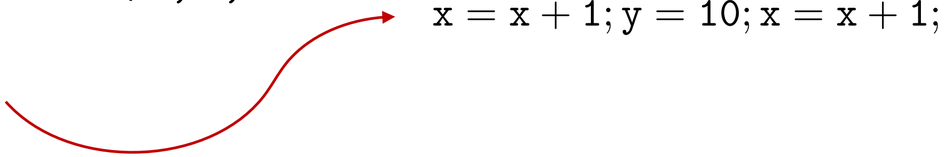
10% have nested eval calls

THE STATIC ANALYSIS PROBLEM

- Static analysis assumption: The program code is static
- Eval brakes this assumption!

```
x = 0;
while (x >= 0) {
  if (x % 2 == 0)
    x = x + 1;
  else
    x = x + 2;
}
```

```
while (x++ < 3)
  os = os + "x=x+1;y=10;x=x+1;";

if (x > 10)
  os = "while(x>5){x=x+1;y=x;}";

if (x == 5)
  os = "hello";

if (x == 8)
  os = "while(x;";

eval(os);
```



Derived at run-time

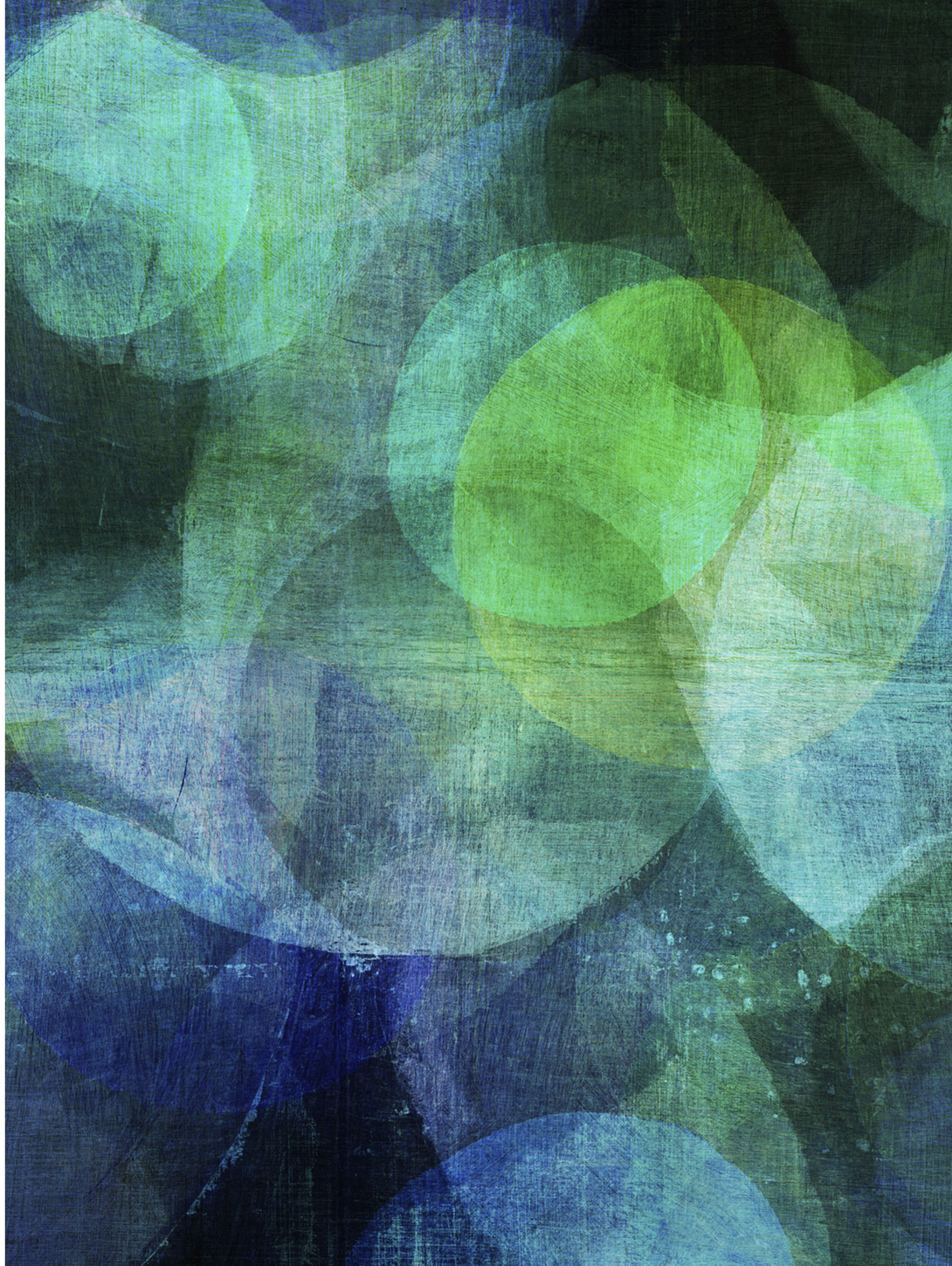
May contain other eval calls

STATE OF THE ART IN ANALYZING DYNAMIC CODE

- Before introducing the sound static analyzer:
  - Several static analyzers have been proposed for dynamic languages (PHP, Python, Ruby…)
  - String analyses fails to precisely analyze string manipulation programs
  - Almost none of them faced the problem of analyzing string-to-code statements
  - They ignore the eval effects (inducing unsoundness) or forbid eval usage
- A sound static analyzer for dynamic code (TOPS'21): Ingredients
  - A string abstract domain keeping enough information for extracting code
  - Design and implementation of an algorithm for extracting executable code from string abstractions
  - IDEA: Analyze code, when an eval is met, extract executable code and recursively call the static analyzer on this code!

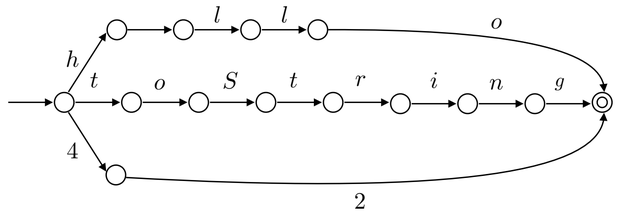
THE EXISTING SOLUTION

A sound abstract interpreter for dynamic code



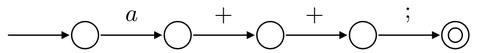
THE STRING ABSTRACTION: FINITE STATE AUTOMATA

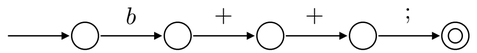
```
{ "42",
"toString",
"hello"}
```



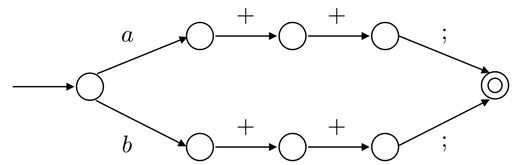
The guard value is statically unknown

```
if (...)
  x = "a++;"
else
  x = "b++;"

eval(x);
```





Which is the value of `x` before the eval execution?



THE STRING ABSTRACTION: FINITE STATE AUTOMATA

The guard value is statically unknown

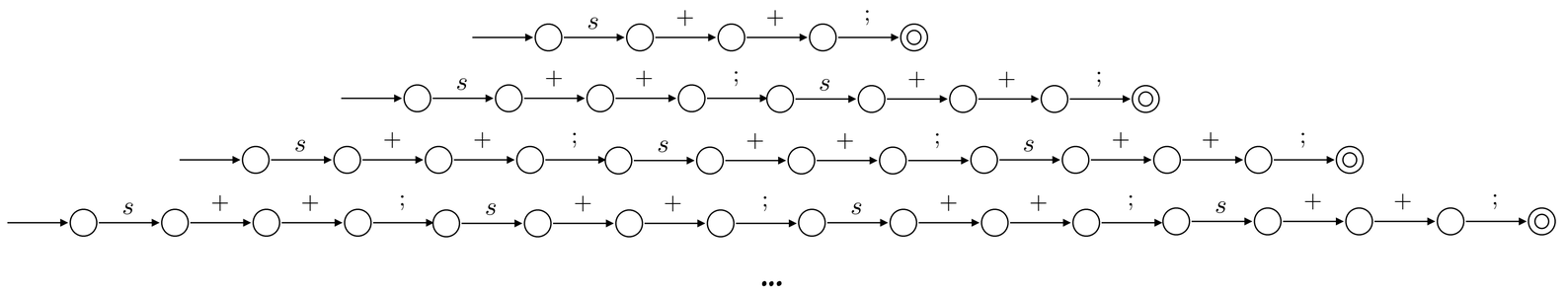
```
while (...)
  x = x + "s++;"
```

→ `"s++;s++;s++;s++;..."`

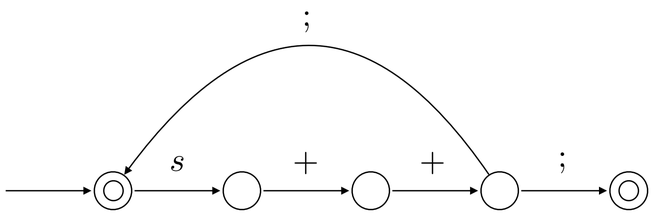
`eval(x)`

Which is the value of `x` before the eval execution?

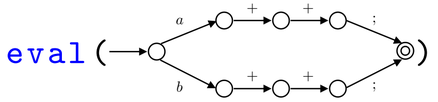
We want to ensure:
- Soundness
- Convergence



Widening



STATIC ANALYSIS OF EVAL



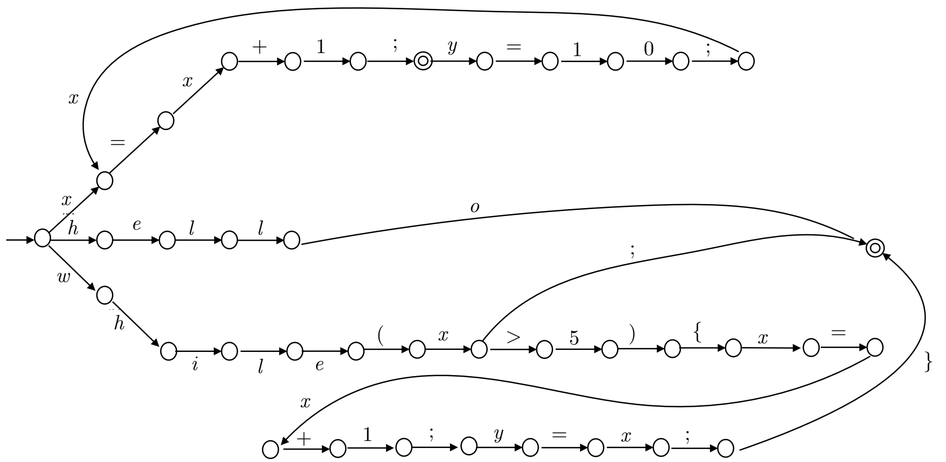
?

➤ How do we extract executable code from automata? 1) We abstract the argument!

```
while (x++ < 3)
  os = os + "x=x+1;y=10;x=x+1;";

if (x > 10)
  os = "while(x>5){x=x+1;y=x;}";

if (x == 5)
  os = "hello";

if (x == 8)
  os = "while(x;";

eval(os);
```

Not executable

os



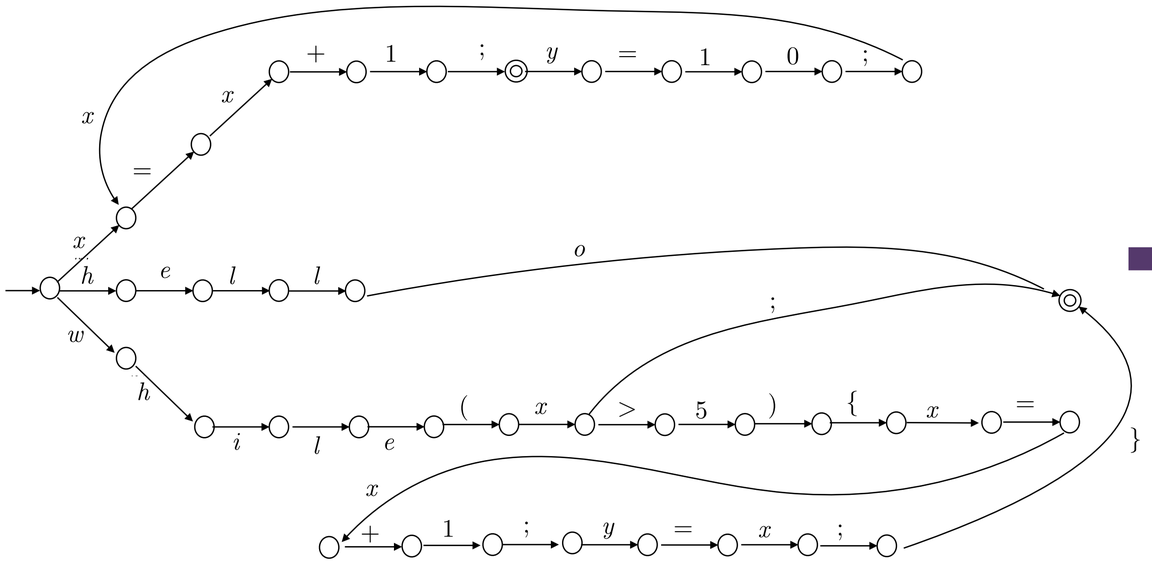
The value of x is statically unknown!

FROM STRINGS TO SEQUENCES OF PARTIAL STATEMENTS

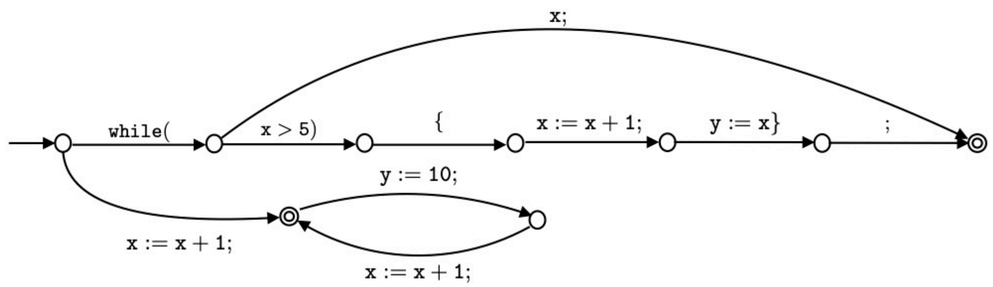
- 2) Automaton of characters ➠ Automaton of partial statements

A partial statement is a prefix statement ending with a punctation symbol or a punctuation symbol

`x=x+1;` `while(` `x>5)` `{` `}`

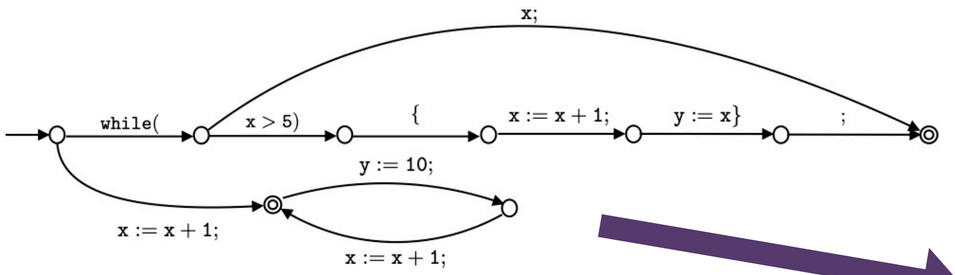


Partial statements



FROM PARTIAL STATEMENTS TO CODE

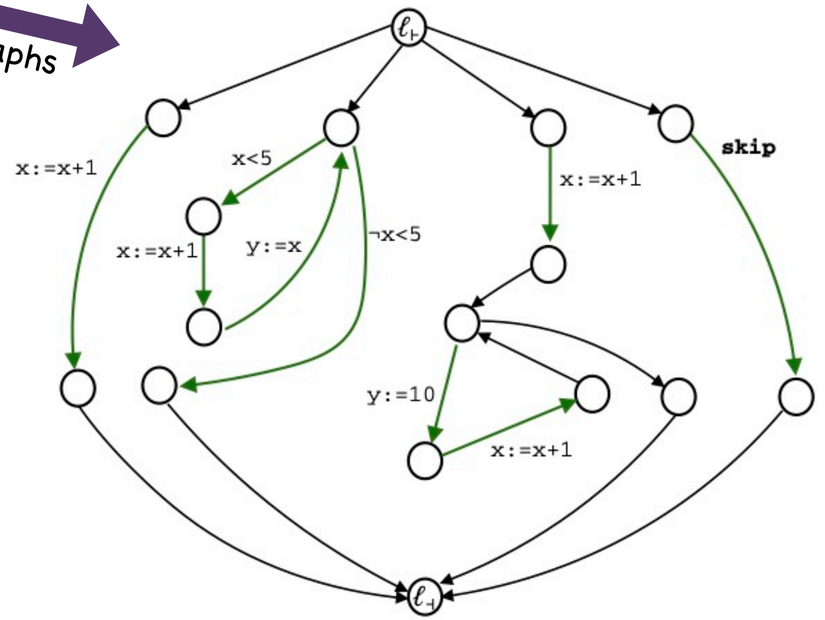
- 3) Automaton of partial statements ➠ Regular Expressions ➠ CFG



Regular expressions

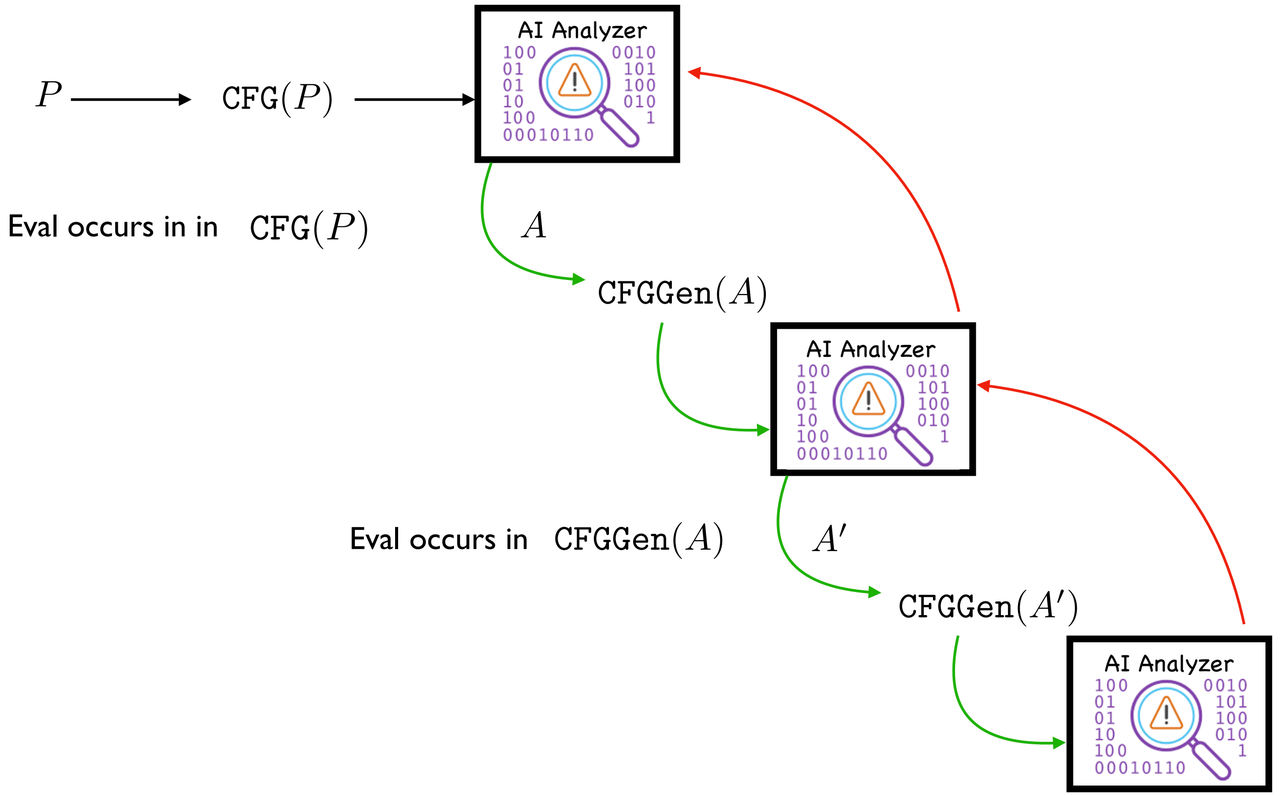
$x:=x+1; \parallel \textbf{while}(x>5)\{x:=x+1;y:=x\}; \parallel x:=x+1;(y:=10;x:=x+1;)^{*} \parallel \textbf{while}(x;$

Control flow graphs



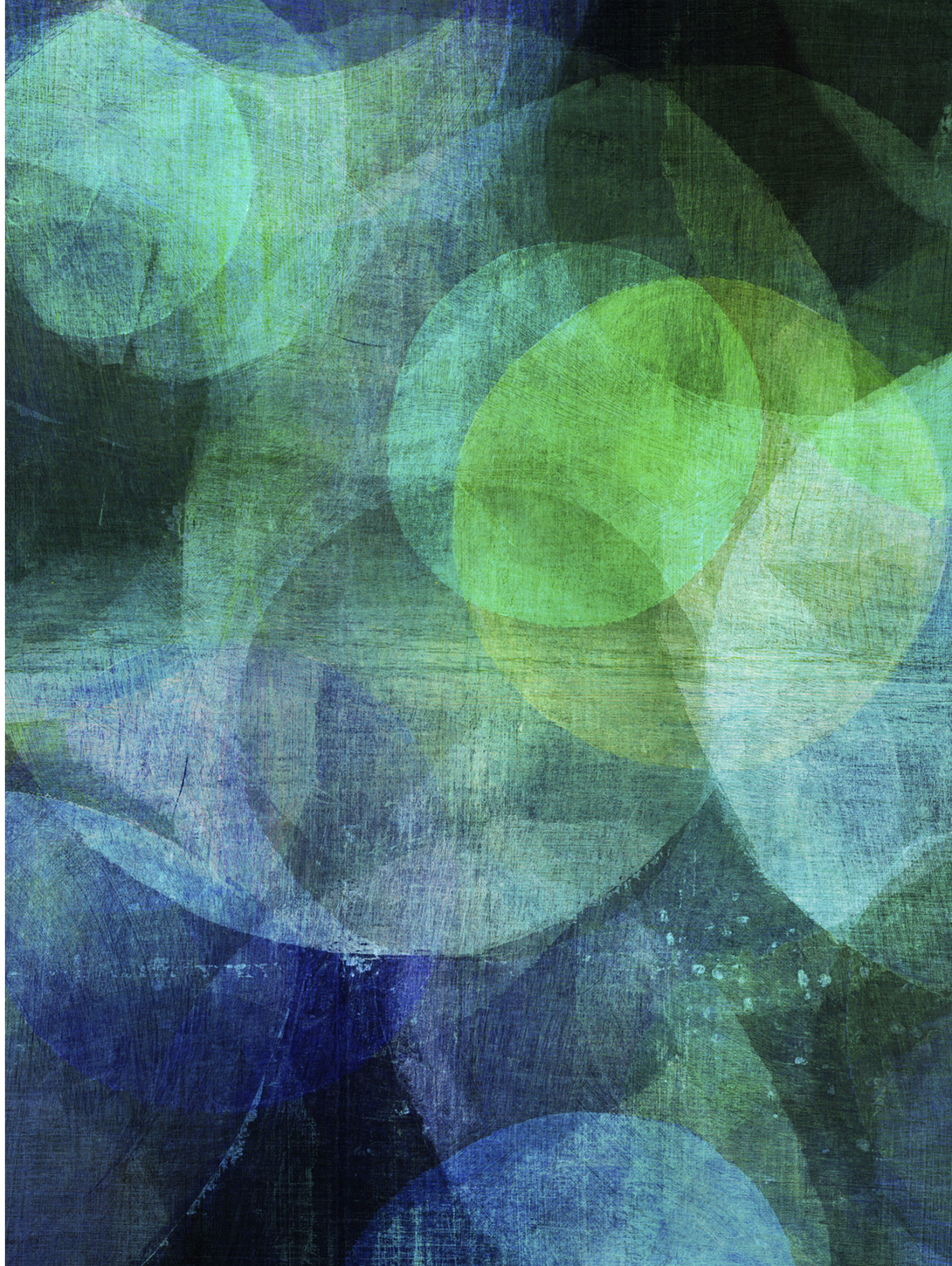
ANALYZING DYNAMICALLY GENERATED CODE

- 4) Recursively call the analyzer on the generated CFG!



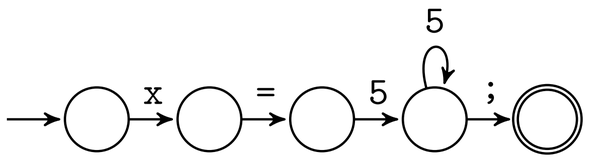
THE ANALYSIS LIMITS

Soundness vs Imprecision



LIMITATIONS... A SOLUTION

- In order to guarantee soundness we lose precision!
- When the executable code contained in the automaton is infinite we have a problem!



$\mathscr{L}(A) = \{x = 5^n; \mid n > 0\}$

```
str = "x=5";
while (i < 3) {
 str = str + "5";
 i = i + 1;
}
str = str + ";"; eval(str);
```

x=5;,x=55;,x=555;...

Infinite set of executable programs

Return TOP to guarantee soundness (TOPS'21)

Tame infinite sets of programs by abstracting code

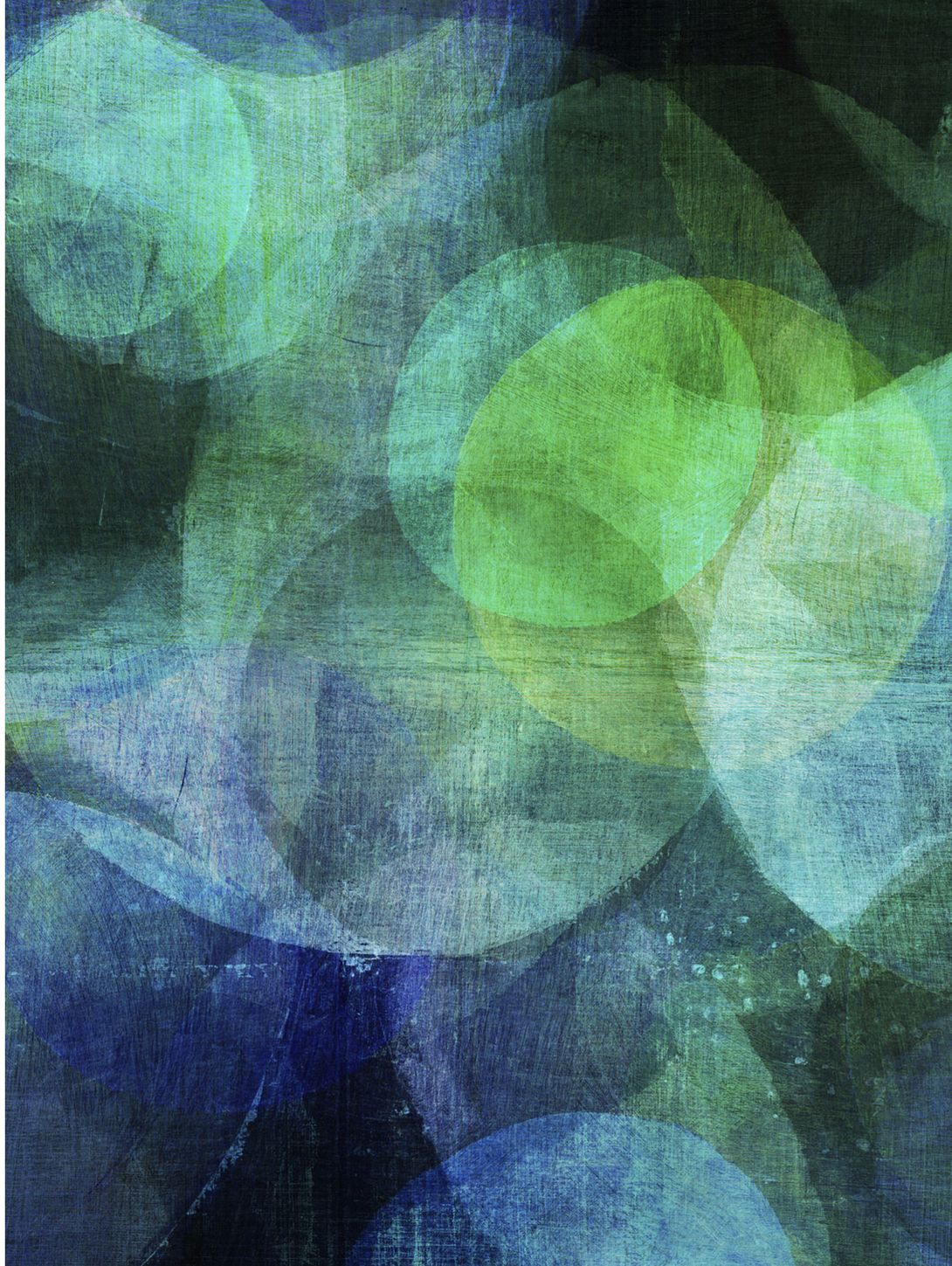
$x := 5(5)^*;$

HOW CAN WE IMPROVE PRECISION?

By performing the analyzer on abstract code!

SEMANTIC-DRIVEN CODE ABSTRACTION

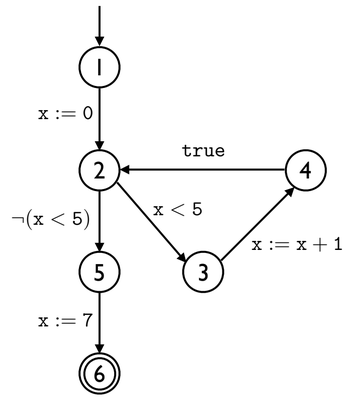
Taming "infinite" programs

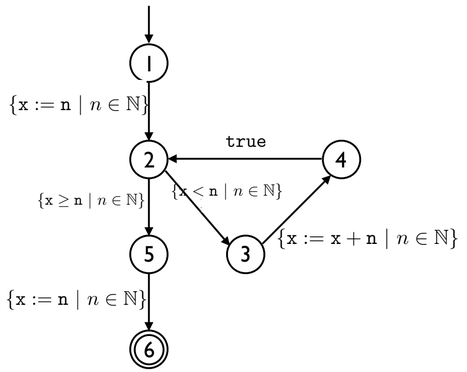


CODE ABSTRACTION

- We model code as control flow graphs (CFG)
- We leave unchanged the control structure and we abstract the effects on the state: labels as sets of statements

```
ℓ1 x := 0;
ℓ2 while (x<5)
   {ℓ3 x := x + 1ℓ4};
ℓ5 x:=7ℓ6
```



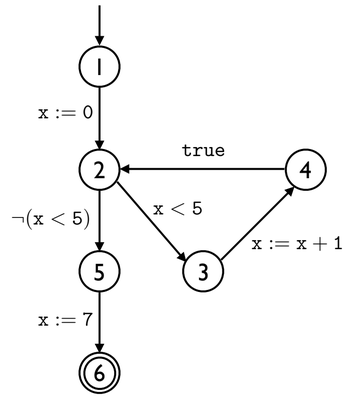


It represents infinite programs

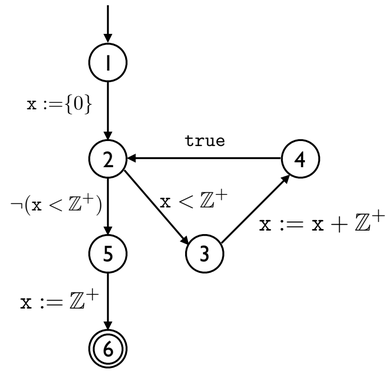
CODE ABSTRACTION VS SEMANTIC ABSTRACTION

- Semantics naturally induces a code abstraction: By abstractly observing code we cannot distinguish between statements with involving the same abstract values

```
ℓ1 x := 0;
ℓ2 while (x<5)
   {ℓ3 x := x + 1ℓ4};
ℓ5 x:=7ℓ6
```



Sign abstraction



$\rho \in uco(\mathbb{V})$

Semantic abstraction

$$\overline{\Upsilon}[\rho](\mathtt{x:=e}) \stackrel{\text{def}}{=} x := \widehat{\rho}(\mathtt{e}) \stackrel{\text{def}}{=} \{\, x := \mathtt{e}' \mid \mathtt{e}' \in \widehat{\rho}(\mathtt{e}) \,\}$$
$$\overline{\Upsilon}[\rho](\mathtt{b}) \stackrel{\text{def}}{=} \widehat{\rho}(\mathtt{b}) \qquad \overline{\Upsilon}[\rho](\mathtt{eval(s)}) \stackrel{\text{def}}{=} \mathtt{eval}(\widehat{\rho}(\mathtt{s}))$$

Code abstraction

$$\widehat{\rho}(\mathtt{a}) : \begin{cases} \widehat{\rho}(\mathtt{a_1 op a_2}) \stackrel{\text{def}}{=} \{\, \mathtt{a'op a''} \mid \mathtt{a'} \in \widehat{\rho}(\mathtt{a_1}), \mathtt{a''} \in \widehat{\rho}(\mathtt{a_2}) \,\} \stackrel{\text{def}}{=} \widehat{\rho}(\mathtt{a_1}) \mathtt{op} \widehat{\rho}(\mathtt{a_2}) \\ \widehat{\rho}(\mathtt{x}) \stackrel{\text{def}}{=} \mathtt{x}, \qquad \widehat{\rho}(\mathtt{n}) \stackrel{\text{def}}{=} \{\, \mathtt{m} \mid m \in \rho(\{n\}) \,\} \end{cases}$$
$$\widehat{\rho}(\mathtt{b}) : \begin{cases} \widehat{\rho}(\mathtt{b_1 bop b_2}) \stackrel{\text{def}}{=} \widehat{\rho}(\mathtt{b_1}) \mathtt{bop} \widehat{\rho}(\mathtt{b_2}), \qquad \widehat{\rho}(\neg \mathtt{b}) \stackrel{\text{def}}{=} \neg \widehat{\rho}(\mathtt{b}) \\ \widehat{\rho}(\mathtt{x}) \stackrel{\text{def}}{=} \mathtt{x}, \qquad \widehat{\rho}(\mathtt{true}) \stackrel{\text{def}}{=} \{\, \mathtt{t} \mid t \in \rho(\mathtt{true}) \,\}, \qquad \widehat{\rho}(\mathtt{false}) \stackrel{\text{def}}{=} \{\, \mathtt{t} \mid t \in \rho(\mathtt{false}) \,\} \end{cases}$$
$$\widehat{\rho}(\mathtt{s}) : \begin{cases} \widehat{\rho}(\mathtt{concat(s_1, s_2)}) \stackrel{\text{def}}{=} \mathtt{concat}(\widehat{\rho}(\mathtt{s_1}), \widehat{\rho}(\mathtt{s_2})), \\ \widehat{\rho}(\mathtt{substr(s, a_1, a_2)}) \stackrel{\text{def}}{=} \mathtt{substr}(\widehat{\rho}(\mathtt{s}), \widehat{\rho}(\mathtt{a_1}), \widehat{\rho}(\mathtt{a_2})) \\ \widehat{\rho}(\mathtt{x}) \stackrel{\text{def}}{=} \mathtt{x}, \qquad \widehat{\rho}(\text{"}\sigma\text{"}) \stackrel{\text{def}}{=} \{\, \text{"}\delta\text{"} \mid \delta \in \rho(\sigma) \,\} \end{cases}$$

Induced code abstraction $\overline{\Upsilon}[\rho]$ ➠ it can be embedded in the original analysis for ρ

WHICH CODE ABSTRACTIONS CAN BE ANALYZED?

- Given a static analysis observing the semantic abstraction ρ, which code abstractions η can be still be analyzed without changing the analysis?
- Let φ denotes CFG labels

$$[\![\eta(\varphi)]\!] = [\![\eta(\varphi)]\!]^{\rho}$$

COMPLETENESS

WHICH CODE ABSTRACTIONS CAN BE ANALYZED?

- Given a static analysis observing the semantic abstraction ρ, which code abstractions η can be still be analyzed without changing the analysis?
- Let φ denotes CFG labels

$$[\![\eta(\varphi)]\!] = [\![\eta(\varphi)]\!]^{\rho}$$

COMPLETENESS

- Example: ρ=Sign, η(x:=5)={x:=n | 0 < n < 6} ✘
- Example: ρ=Sign, η(x:=5)={x:=n | n ≥ 0} ✓

WHICH CODE ABSTRACTIONS CAN BE ANALYZED?

- Given a static analysis observing the semantic abstraction ρ, which code abstractions η can be still be analyzed without changing the analysis?
- Let φ denotes CFG labels

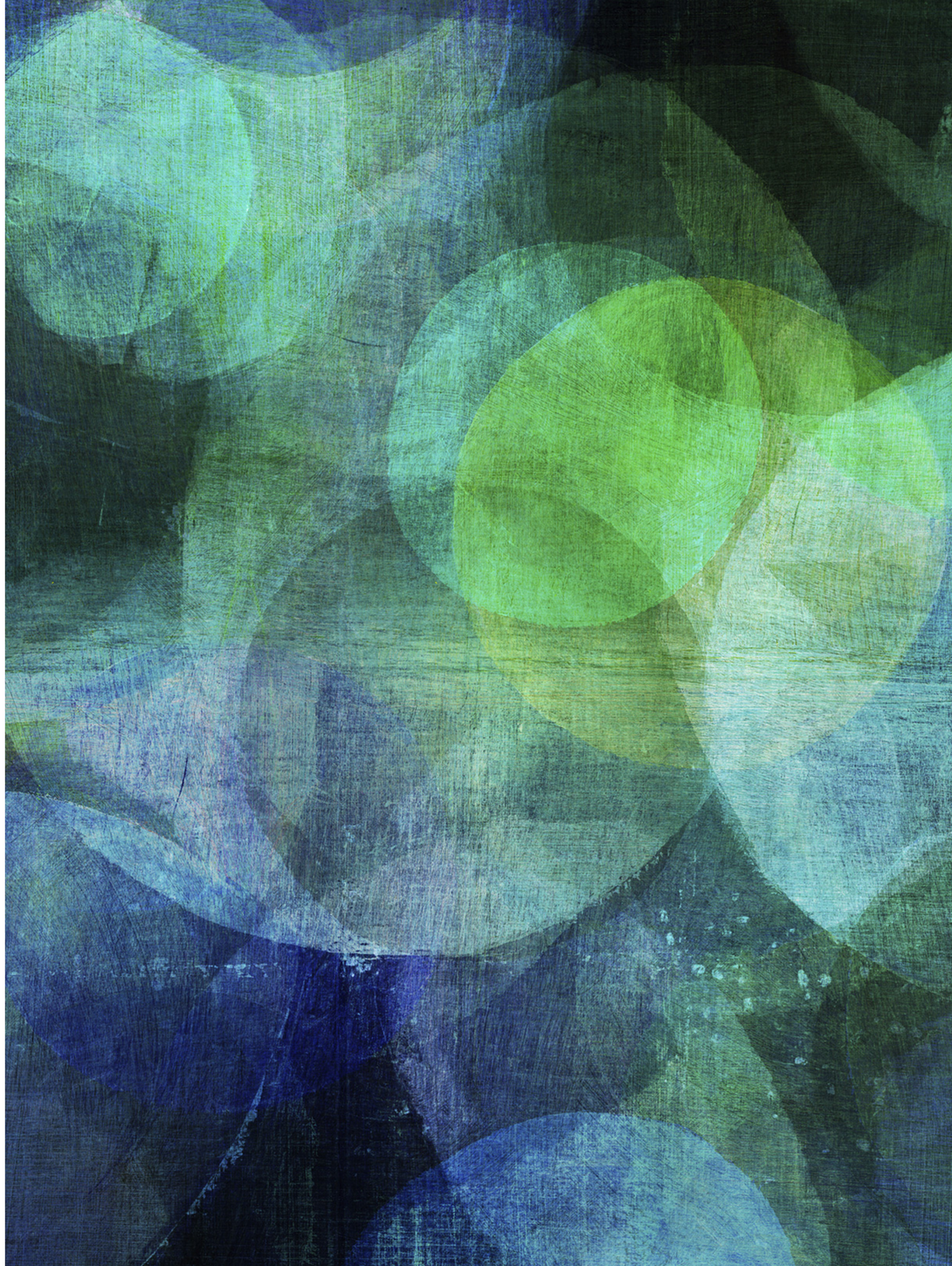
$$[\![\eta(\varphi)]\!] = [\![\eta(\varphi)]\!]^{\rho}$$

COMPLETENESS

- The code abstraction $\overline{\eta}_{\uparrow} \stackrel{def}{=} \overline{\Upsilon}[\rho] \circ \eta$ always satisfies this equation, for any initial code abstraction η!
- We can make any abstract code analyzable by a given analysis!

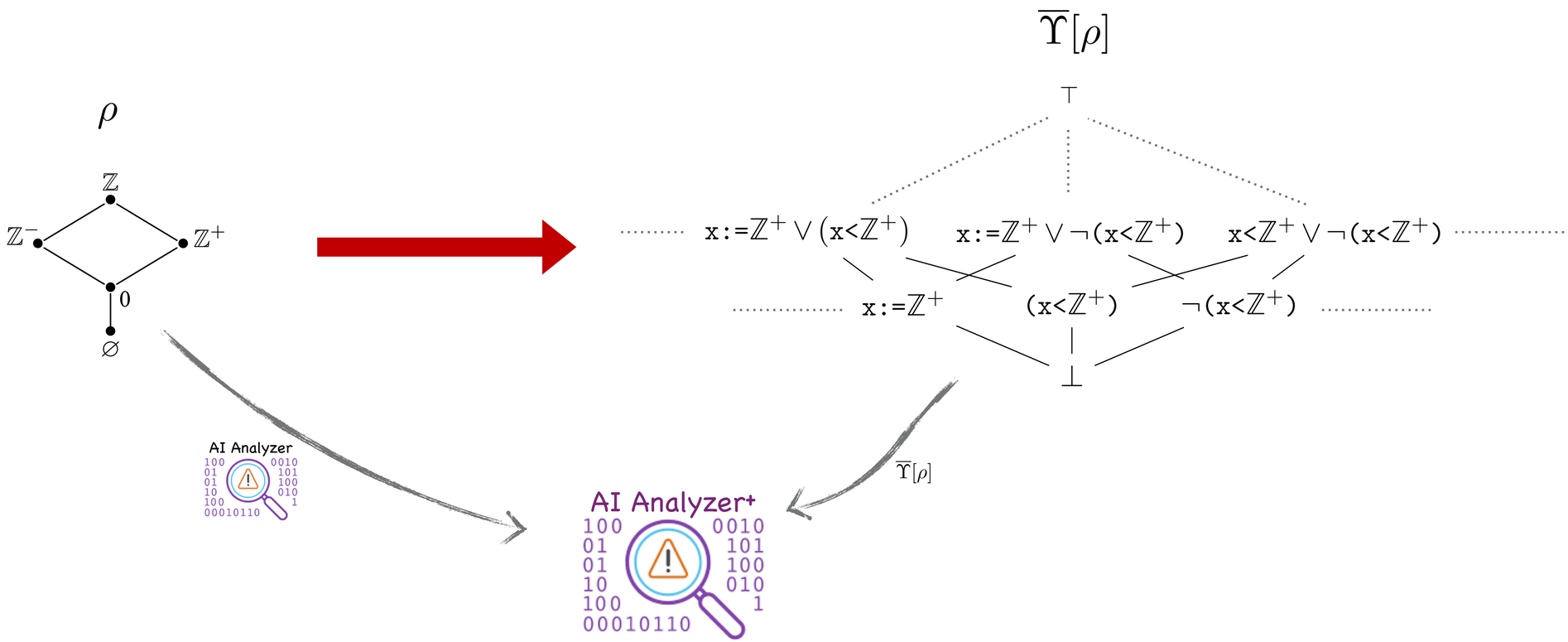
AN IMPROVED DYNAMIC CODE ANALYSIS

Analyzing abstract code



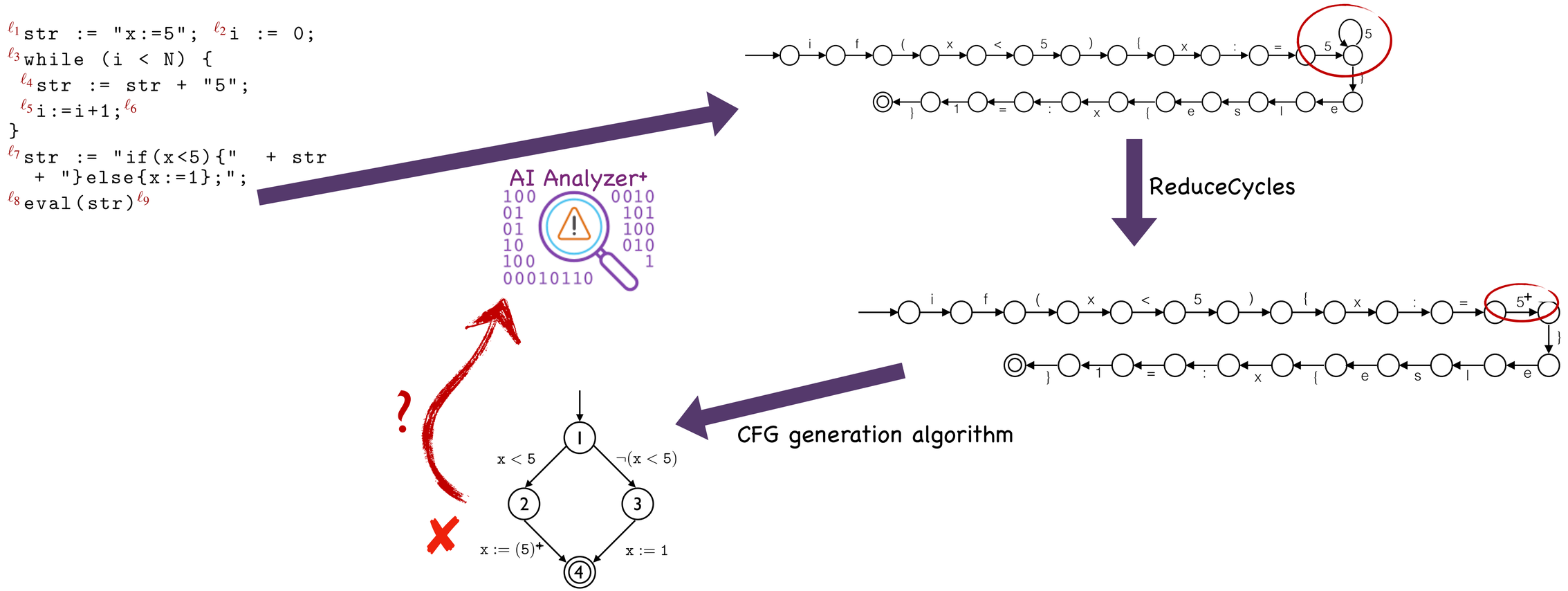
AN IMPROVED DYNAMIC CODE ANALYSIS

- Consider a static analysis observing the semantic abstraction ρ, interpreting also code abstracted by $\Upsilon[\rho]$



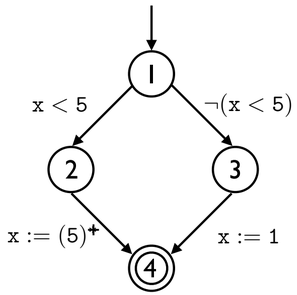
FROM AUTOMATA TO (ABSTRACT) CFG

- Analyze the program and when eval is met extract the automaton recognizing the language of its argument

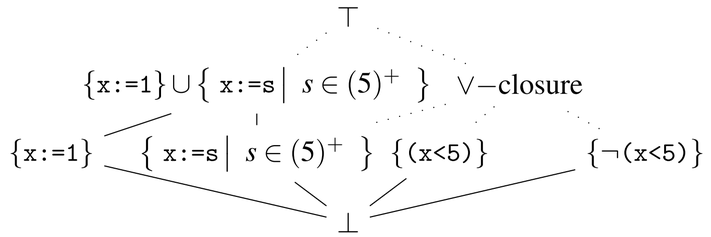


MAKING (ABSTRACT) CODE ANALYZABLE

- The abstract d=code generated induces a code abstraction η

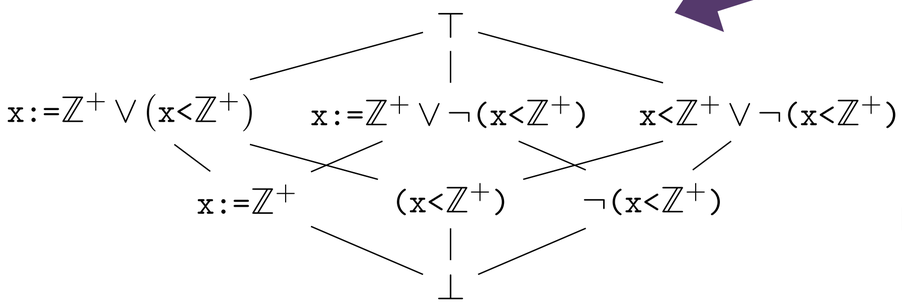


Induced abstraction



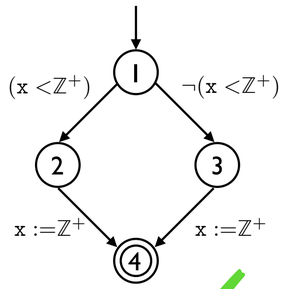
η

Completeness



$\eta \circ \overline{\Upsilon}[\rho]$

Abstract CFG



CONCLUDING

Where we are and where we can go

"YOU CAN ANALYZE THE CODE YOU DON'T SEE!"