Mixing Type Checking and Symbolic Execution

Khoo Yit Phang (UMD College Park)

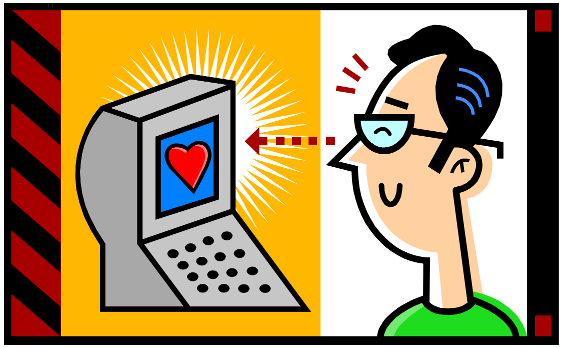
Bor-Yuh Evan Chang (CU Boulder)

Jeffrey S. Foster (UMD College Park)

FRACTAL - December 5, 2009

An all too common scenario …

Oh Verifier, help me prove my program has no bugs.



On line 142, there may be a bug.

Isn’t it obvious this can’t happen!?!?

Static verifiers must over-approximate and thus raise false alarms.

False alarm example:
The need for path sensitivity

🔓 `unlock();` ✗

🔓

`if (multithreaded) fork();`

… *statements*$_1$ … 🔓

`if (multithreaded) lock();` ?

… *statements*$_2$ … 🔒

`if (multithreaded) unlock();` ✗

mislabels good code as buggy

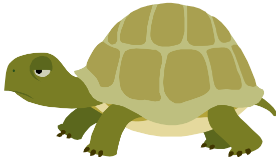
This abstraction is too coarse. Standard practice is to re-design it to be precise enough for this example.

Re-design with path sensitivity

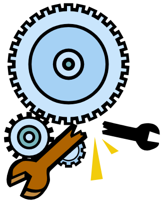
🔓 unlock(); ✗

```
if (multithreaded) fork();
… statements1 …
if (multithreaded) lock();
… statements2 …
if (multithreaded) unlock();
```

multithreaded ∧ 🔒
∨
¬multithreaded ∧ 🔓

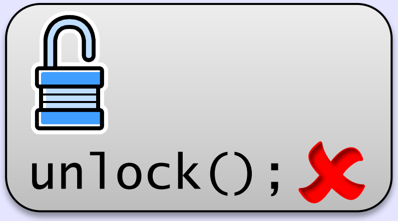


<u>Bad</u>: Too much precision leads to slow, inefficient analysis

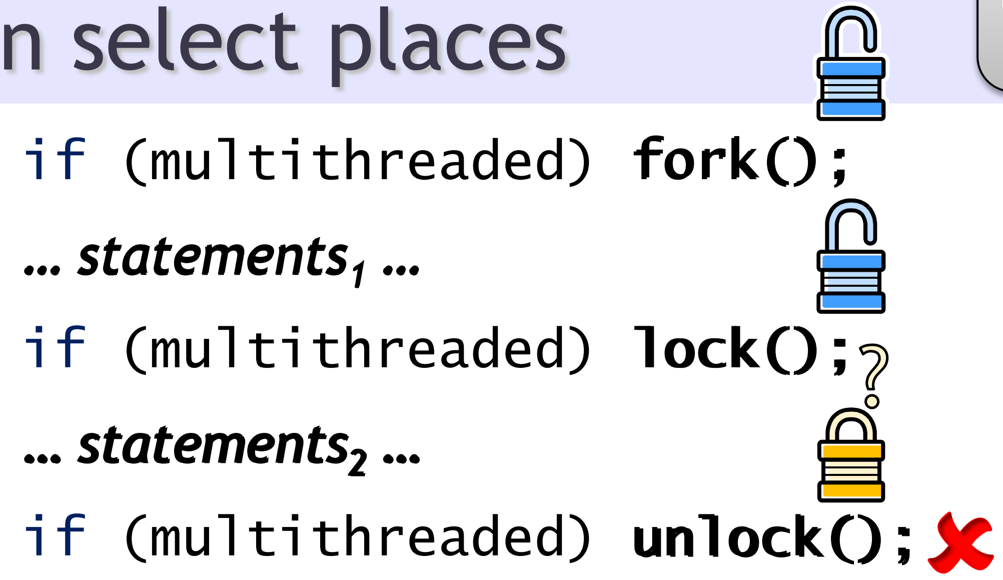


<u>Bad</u>: Ad-hoc addition of precision leads to brittle analyzers

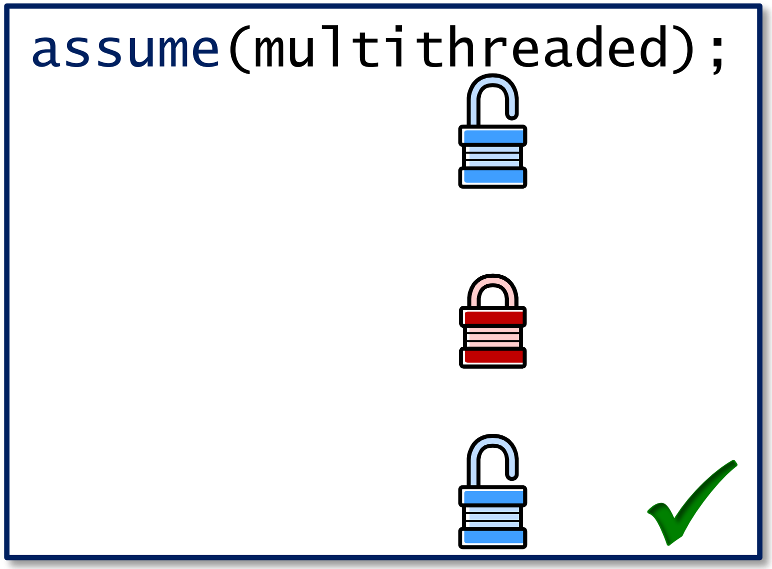
Observation: Just need precision in select places



unlock(); ✗

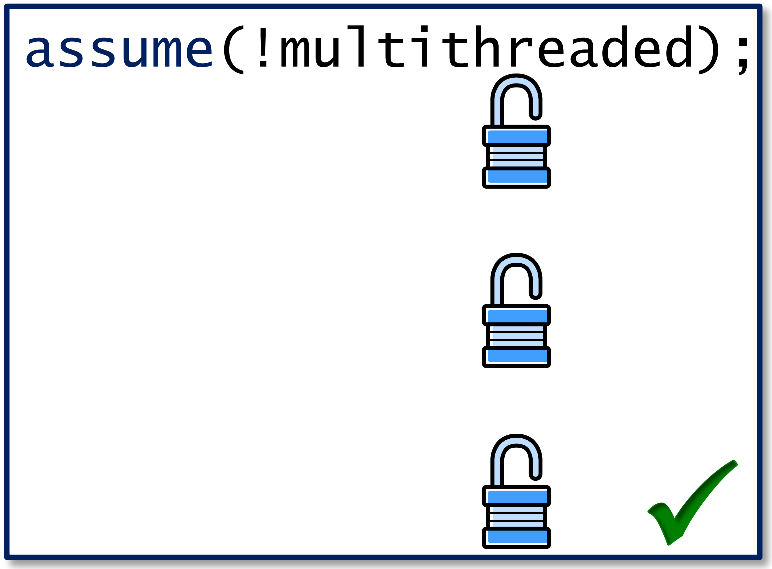


```
if (multithreaded) fork();
… statements₁ …
if (multithreaded) lock(); ?
… statements₂ …
if (multithreaded) unlock(); ✗
```



assume(multithreaded);

✓

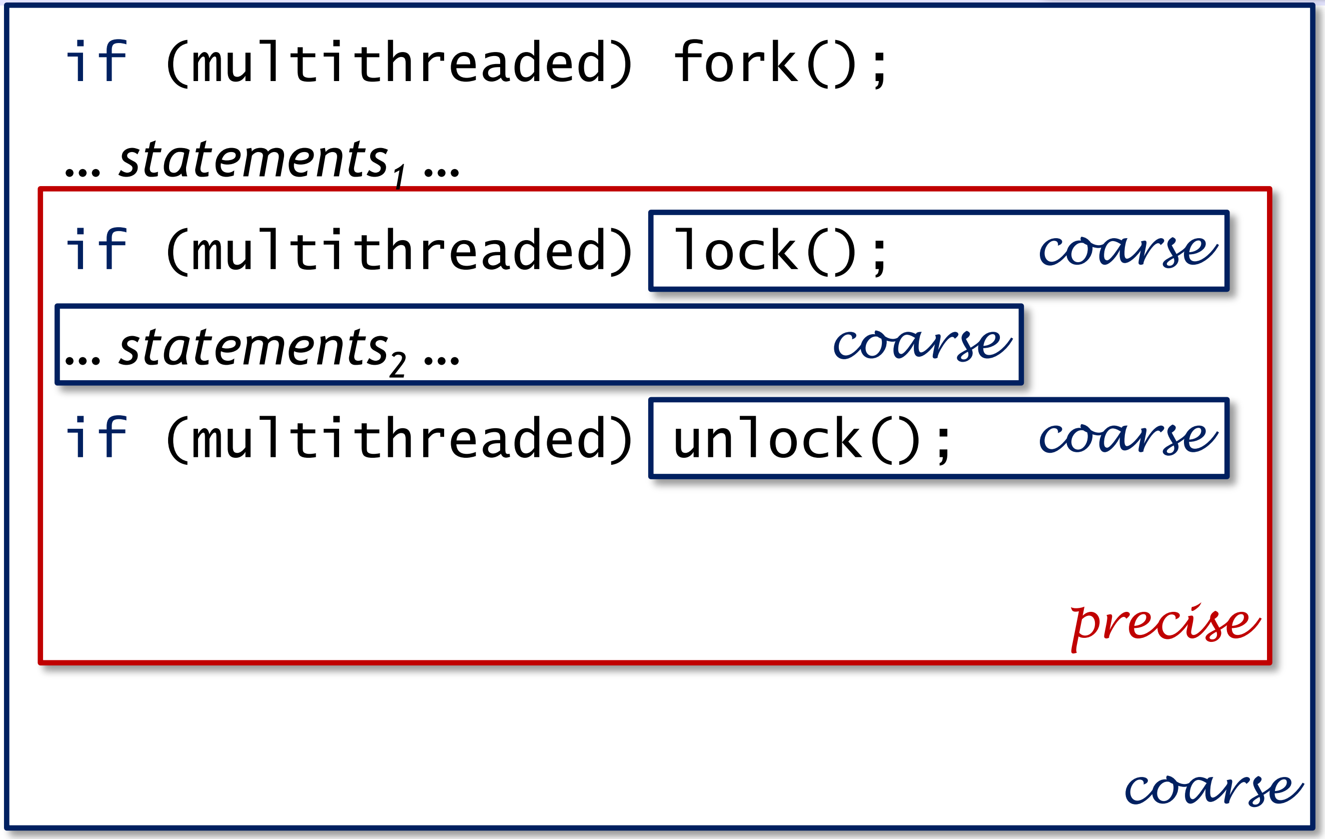


assume(!multithreaded);

✓

Approach: Split the program between analyses

unlock(); ✗



`if (multithreaded) fork();`

... *statements$_1$* ...

`if (multithreaded) lock();`

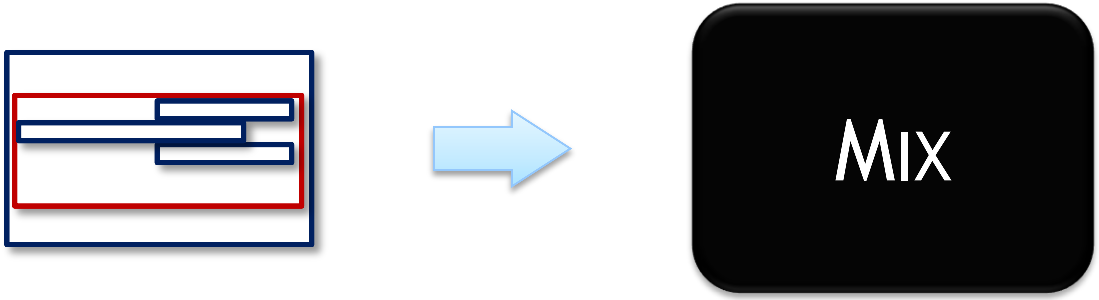
... *statements$_2$* ...

`if (multithreaded) unlock();`

Switch to precise analysis only where needed

MIX is …

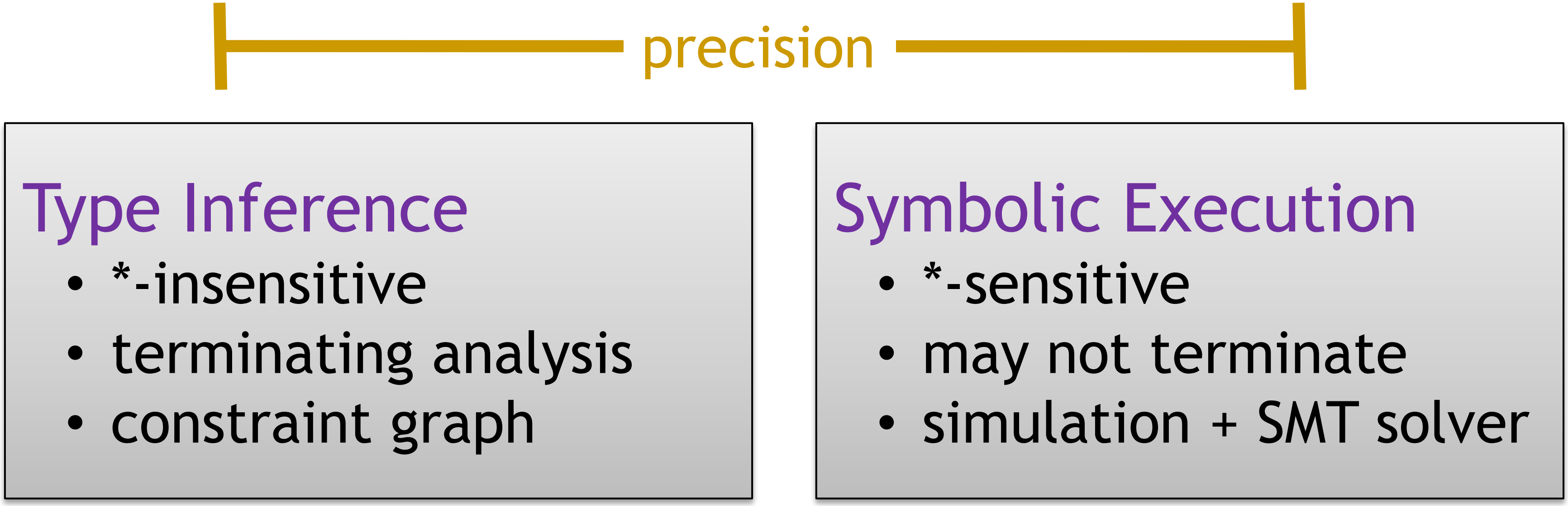
A tunable program analysis that alternates between type inference and symbolic execution



- Standard, off-the-shelf type inference
- Standard, off-the-shelf symbolic execution
- Mixing rules to translate information at block boundaries

Why type inference and symbolic execution?

- Case study of extremes



precision

Type Inference

- *-insensitive
- terminating analysis
- constraint graph

Symbolic Execution

- *-sensitive
- may not terminate
- simulation + SMT solver

  - Simple, well-understood algorithms
  - Hard to imagine how to combine in more intricate ways (e.g., in contrast to combining abstract interpreters)

Outline

- Mixing rules
- Examples and idioms for switching blocks
- Preliminary experience with MIXY, a mixed type qualifier inference engine for C

Type checking and symbolic execution at a glance

Type checking

Typing context

x : int, b: bool, y : int

x + (if b then y else 3)

: int

type of the expression

Symbolic execution

Symbolic context

δ ; x = α:int, b = β:bool, y = γ:int

x + (if b then y else 3)

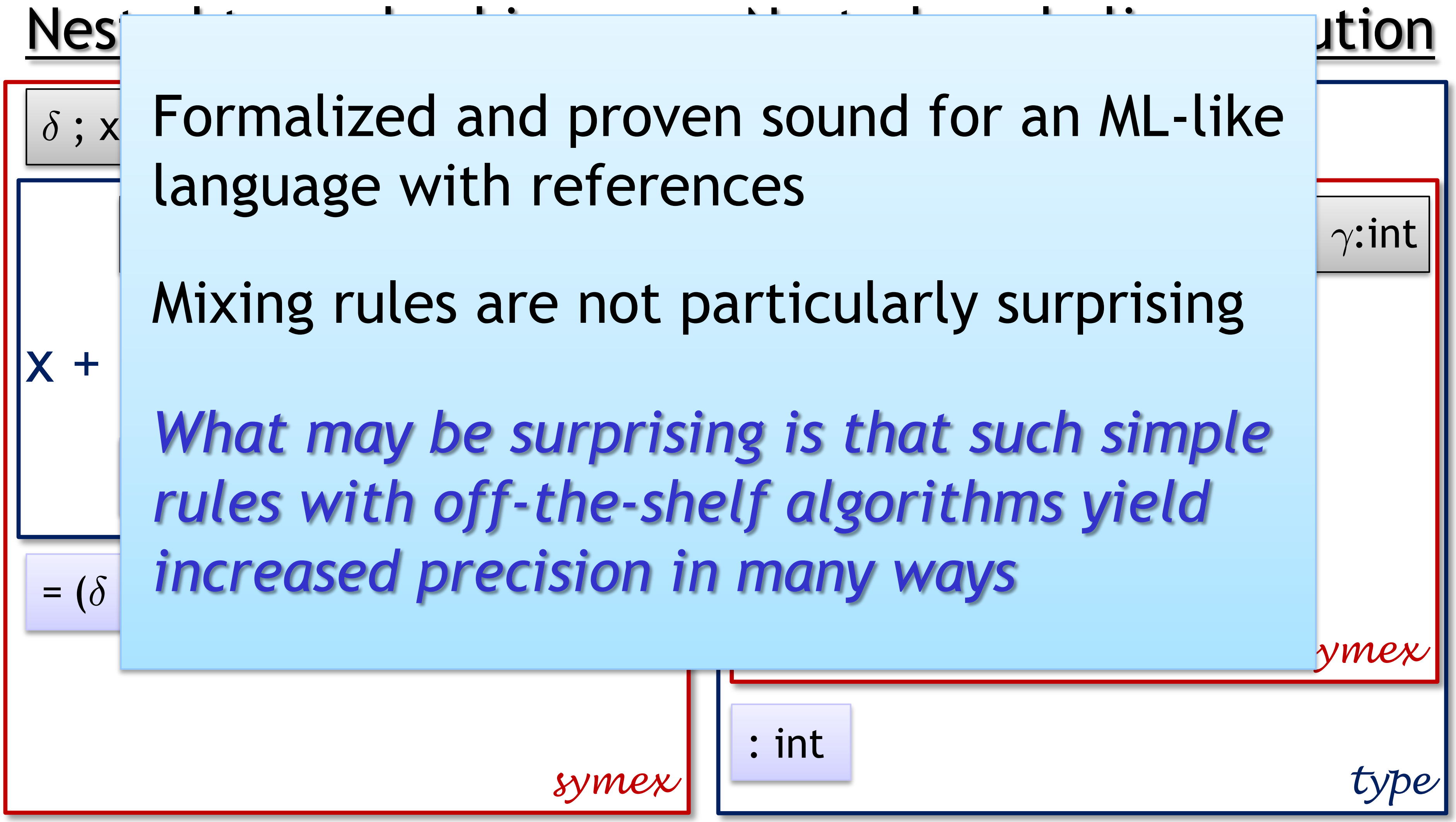
= ($\delta \wedge \beta$; α + γ : int)

= ($\delta \wedge \neg\beta$; α + 3 : int)

path condition

symbolic result along the path

Mixing rules: Conservatively translate states



Formalized and proven sound for an ML-like language with references

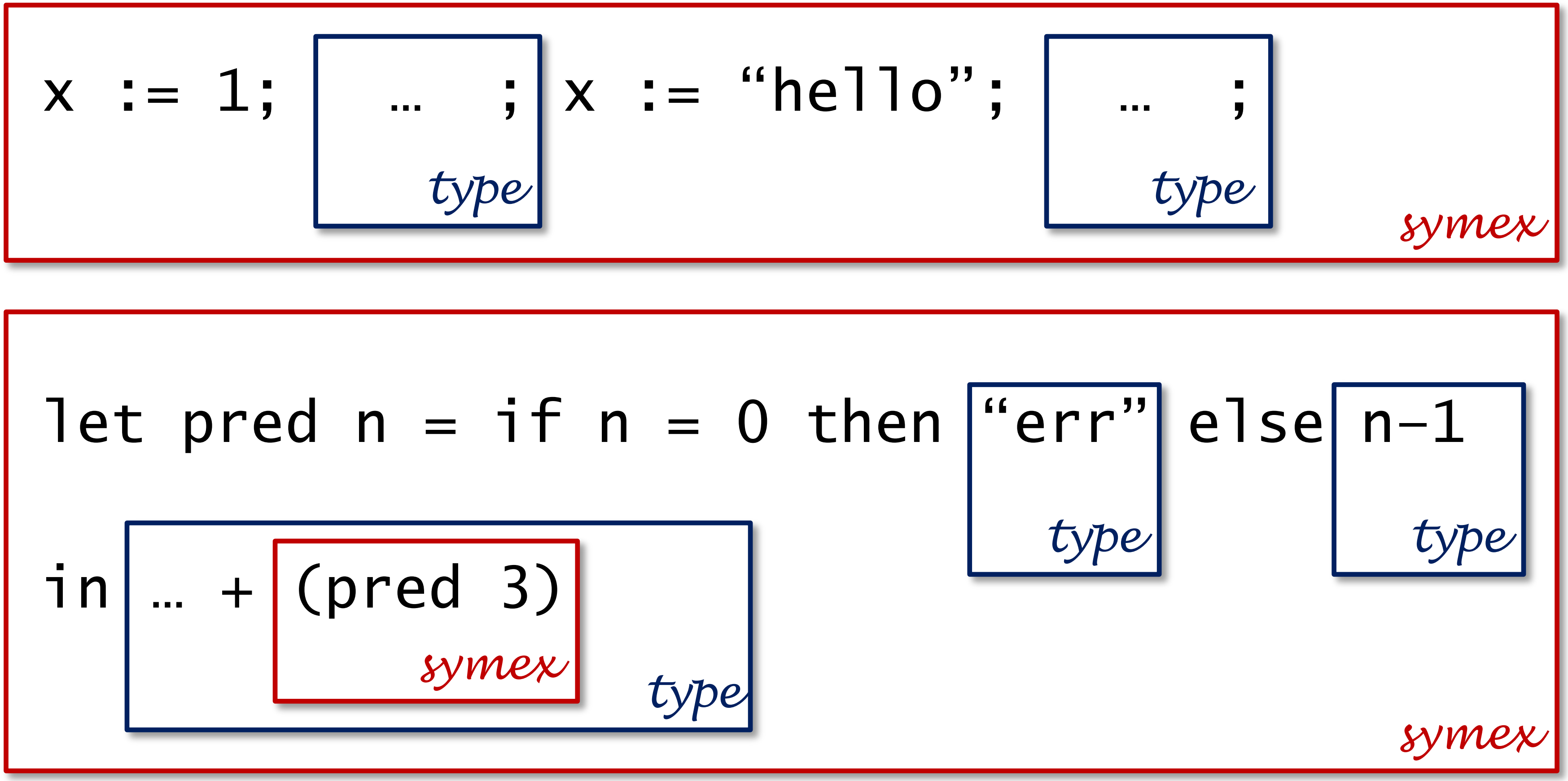
Mixing rules are not particularly surprising

What may be surprising is that such simple rules with off-the-shelf algorithms yield increased precision in many ways

Outline

- Mixing rules
- Examples and idioms for switching blocks
- Preliminary experience with MIXY, a mixed type qualifier inference engine for C

Flow, path, and context sensitivity

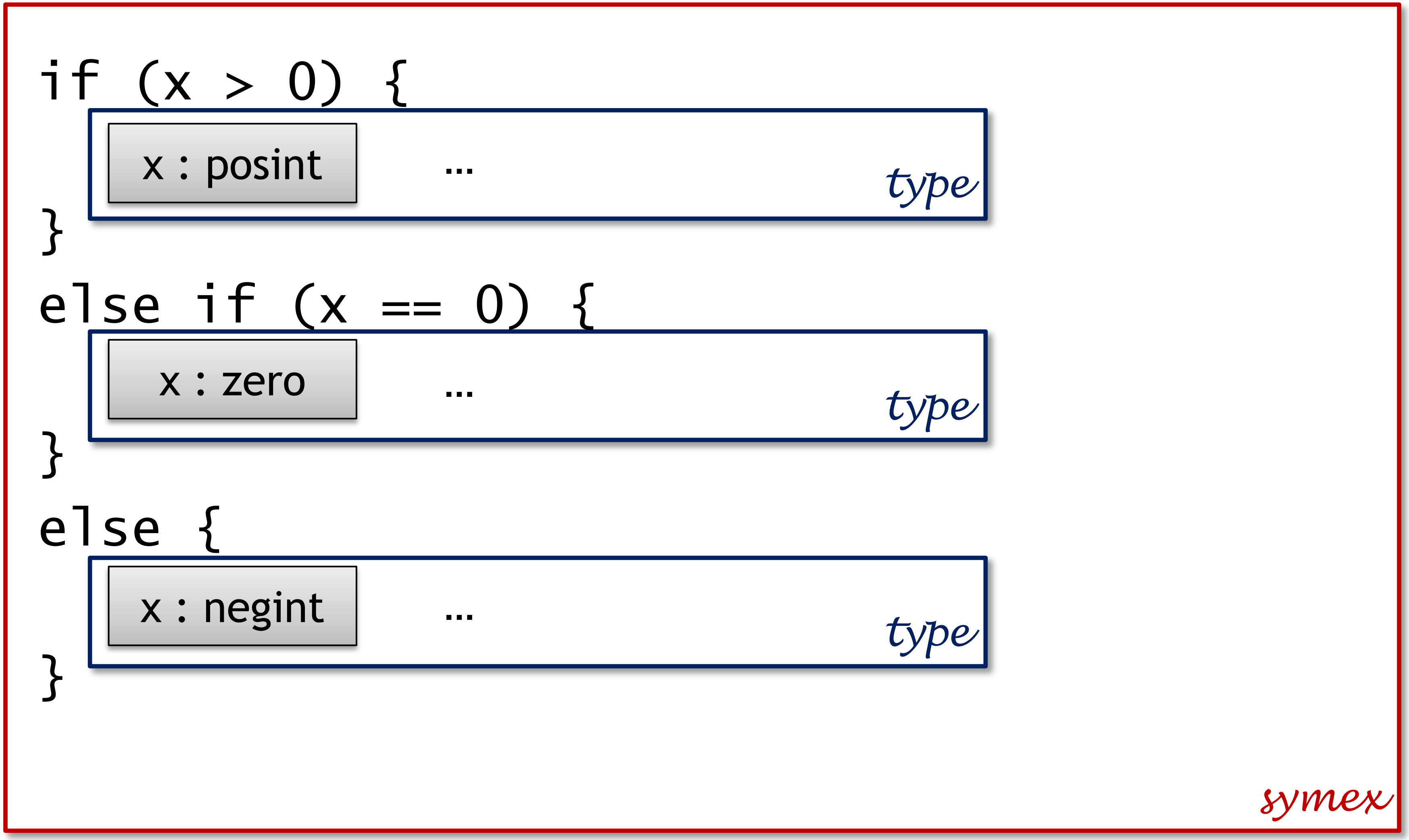


```
x := 1; … ; x := “hello”; … ;
```

```
let pred n = if n = 0 then “err” else n–1
in … + (pred 3)
```

Static type checking for dynamically-typed code

Local refinement



```
if (x > 0) {
    x : posint      …
}
else if (x == 0) {
    x : zero        …
}
else {
    x : negint      …
}
```

Abstraction during symbolic execution

```
let x = unknown_function()         type    in

let y = recursive_function()       type    in

let z = … operation not supported by solver  type  in

…
                                                  symex
```

Outline

- Mixing rules

- Examples and idioms for switching blocks

- Preliminary experience with MIXY, a mixed type qualifier inference engine for C

Preliminary experience

- MIXY, a prototype mixed type qualifier inference engine for C
- Applied to check that a `free` function is called only with a non-null pointer (using `nonnull` type qualifier)
  - On vsftpd 2.0.7
  - Eliminated 2 false warnings
  - A combination of flow, path, and context-sensitivity was required

Conclusion

- New approach for trading off precision and efficiency in static program analysis
- Key: Nestable switching blocks to alternate between different off-the-shelf analyses
- Studied the mixing of type checking and symbolic evaluation
  - Proven soundness of symbolic execution/mix