Refinement Types For Haskell

Niki Vazou,
Eric Seidel,
Ranjit Jhala
(*UC San Diego*)

Dimitrios Vytiniotis,
Simon Peyton-Jones
(*Microsoft*)

Refinement Types

```
{v:Int | v>0}
```

Haskell Type

Predicate*

* From Logical Sub-language

Refinement Types

```
div :: Int -> {v:Int | v > 0} -> Int
```

Haskell Type

Predicate*

* From Logical Sub-language

Refinement Types

```
div :: Int -> {v| v > 0} -> Int
```

Abbreviated

Using Refinement Types

```
div :: Int -> {v|v>0} -> Int
```

Using Refinement Types

```
div :: Int -> {v|v>0} -> Int

good x = let y = 10
         in x `div` y
```

OK

Using Refinement Types

```
div :: Int -> {v|v>0} -> Int

bad x = let y = 0
        in x `div` y
```

Error

Refinement Types for

Array Safety in **ML**

Security Protocols in **F#** **CBV**

Compiler Correctness in **F***

…

How about **Haskell**? **CBN**

A Curious Function…

```
spin   :: Int -> Int
spin x = spin x
```

A Curious Function…

```
spin   :: Int -> {v|false}

spin x = spin x
```

OK

As `spin` does not return *any* value

Using Refinement Types

```
div     :: Int -> {v|v>0} -> Int
spin    :: Int -> {v|false}

ugly x = let y = 0
             z = spin 0
         in x `div` y
```

OK? or **Error?**

Using Refinement Types

```
div     :: Int -> {v|v>0} -> Int
spin    :: Int -> {v|false}

ugly x = let y = 0
             z = spin 0
         in x `div` y
```

OK under **CBV** evaluation

Using Refinement Types

```
div      :: Int -> {v|v>0} -> Int
spin     :: Int -> {v|false}

ugly x = let y = 0
             z = spin 0
         in x `div` y
```

Error under **CBN** evaluation

The Problem

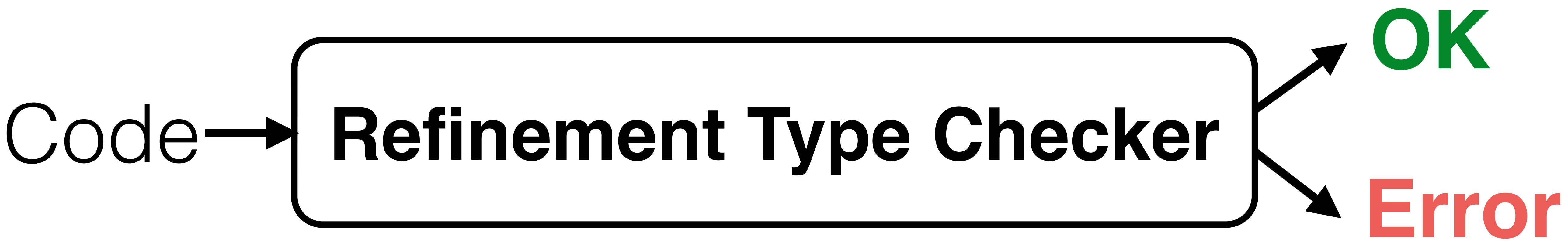
CBV-style typing is *unsound* under CBN!

Reports **Erroneous** code as **OK**

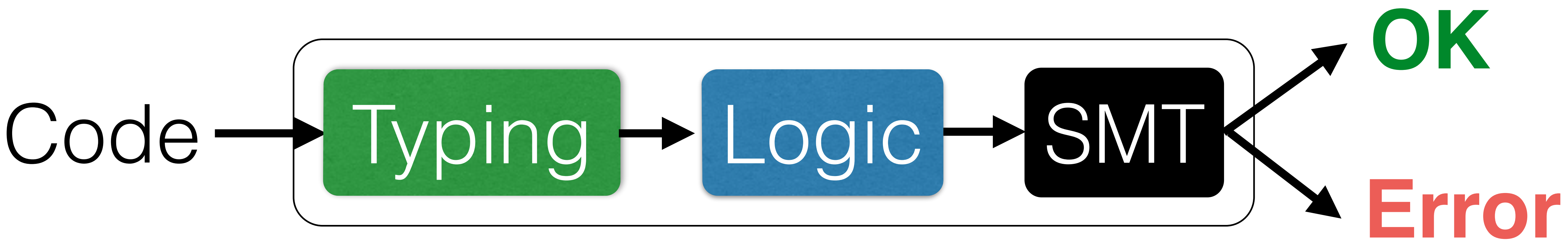
1 Motivation

How to refine types under *CBN*?

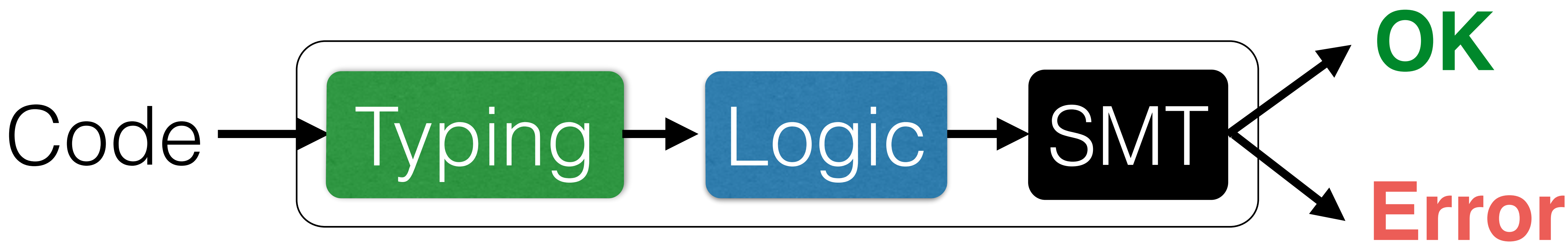
Refinement Typing 101



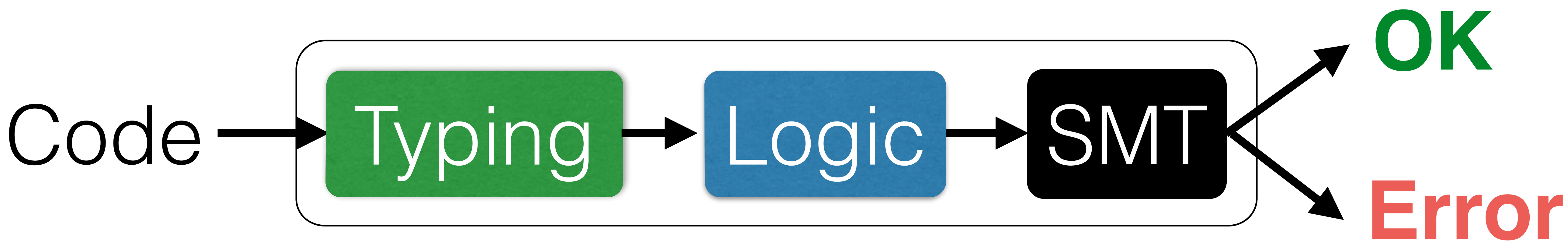
Refinement Typing 101



Refinement Typing 101



1. Source Code to **Typing constraints**
2. **Typing Constraints** to **Logical VC**
3. Check **VC validity** with **SMT Solver**


Code
Typing
Logic
SMT
OK
Error

Code → **Typing**

```
div     :: Int -> {v|v>0} -> Int
spin    :: Int -> {v|false}

ugly x = let y = 0
             z = spin 0
         in x `div` y
```

```
x:{v|true }
y:{v|v=0  }  |- {v|v=0} <: {v|v>0}
z:{v|false}
```

Code → Typing

```
div   :: Int -> {v|v>0} -> Int
spin  :: Int -> {v|false}

ugly x = let y = 0
             z = spin 0
         in x `div` y
```

```
x:{v|true }
y:{v|v=0  }   |- {v|v=0} <: {v|v>0}
z:{v|false}
```

Code → Typing

```
div     :: Int -> {v|v>0} -> Int
spin    :: Int -> {v|false}

ugly x = let y = 0
             z = spin 0
         in x `div` y
```

```
x:{v|true }
y:{v|v=0  }  |- {v|v=0} <: {v|v>0}
z:{v|false}
```

Code → **Typing**

```
div     :: Int -> {v|v>0} -> Int
spin    :: Int -> {v|false}

ugly x = let y = 0
             z = spin 0
         in x `div` y
```

```
x:{v|true }
y:{v|v=0  }   |- {v|v=0} <: {v|v>0}
z:{v|false}
```

Code → Typing

```
div    :: Int -> {v|v>0} -> Int
spin   :: Int -> {v|false}

ugly x = let y = 0
             z = spin 0
         in x `div` y
```

```
x:{v|true }
y:{v|v=0  }   |- {v|v=0} <: {v|v>0}
z:{v|false}
```

Code → Typing

```
div     :: Int -> {v|v>0} -> Int
spin    :: Int -> {v|false}

ugly x = let y = 0
             z = spin 0
         in x `div` y
```

```
x:{v|true }
y:{v|v=0  }   |- {v|v=0} <: {v|v>0}
z:{v|false}
```

Typing → Logic

Encode Subtyping as Logical VC

If VC valid then Subtyping holds

Typing → Logic

Encode Subtyping as Logical VC

```
x:{v|true }
y:{v|v=0  }  |- {v|v=0} <: {v|v>0}
z:{v|false}
```

$\boxed{\texttt{y:\{v|p\}}}$

$\boxed{\texttt{\{v|p\} <: \{v|q\}}}$

Typing → Logic

$$\boxed{y : \{v \mid p\}}$$

Means*: *If* y reduces to a value *then* p[y/v]

Encoded as: "y has a value" => p[y/v]

* Flanagan "Hybrid Type Checking" POPL '06

Typing $\rightarrow$ Logic

$$\boxed{\{v \mid p\} <: \{v \mid q\}}$$

Means: *if* $y : \{v \mid p\}$ *then* $y : \{v \mid q\}$

Encoded as: $p \Rightarrow q$

Typing → Logic

Encode Subtyping ...

```
x:{v|true }
y:{v|v=0  }  |- {v|v=0} <: {v|v>0}
z:{v|false}
```

... as Logical VC

```
  "x has a value"  =>  true
/\"y has a value"  =>  y=0 =>v=0 =>  v>0
/\"z has a value"  =>  false
```

Typing → Logic

Encode Subtyping ...

```
x:{v|true }
y:{v|v=0  }   |- {v|v=0} <: {v|v>0}
z:{v|false}
```

... as Logical VC

```
   "x has a value"  =>  true
/\ "y has a value"  =>  y=0  => v=0  =>  v>0
/\ "z has a value"  =>  false
```

Logic → SMT

```
   "x has a value" => true
/\ "y has a value" => y=0 =>v=0 => v>0
/\ "z has a value" => false
```

How to *encode* "has a value" in VC?

How to *encode* “has a value” in VC?

```
  “x has a value”  =>  true
/\“y has a value”  =>  y=0 =>v=0 => v>0
/\“z has a value”  =>  false
```

CBV: Binders *must be* values

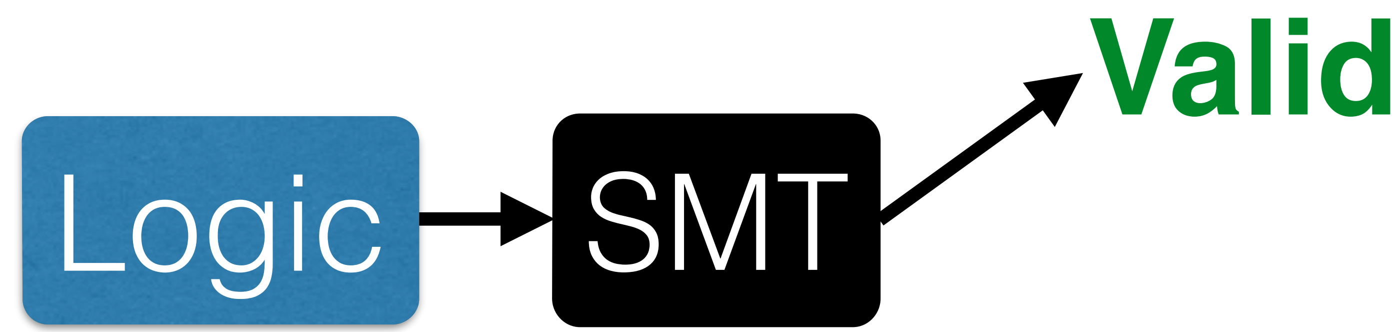
`x`, `y`, and `z` are trivially values

How to *encode* “has a value” in VC?

```
   true
/\ y=0 =>v=0 => v>0
/\ false
```

CBV: Binders *must be* values

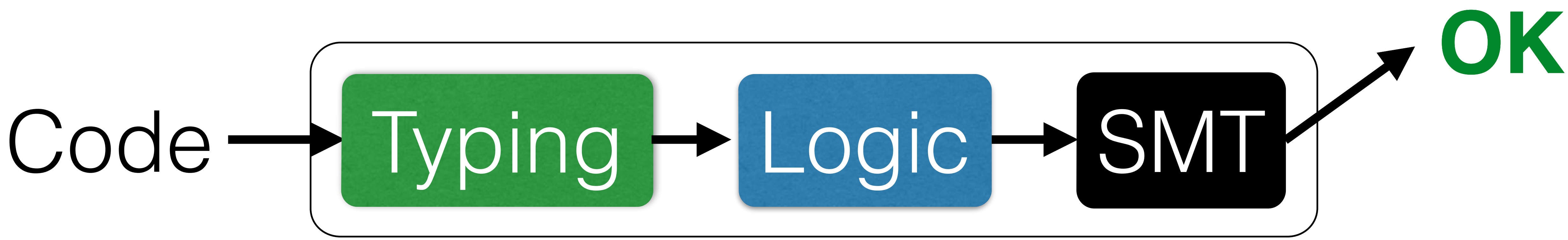
x, y, and z are trivially values




```
        true
/\ y=0 =>v=0 => v>0
/\ false
```

CBV: Binders *must be* values

`x`, `y`, and `z` are trivially values




```
div    :: Int -> {v|v>0} -> Int
spin   :: Int -> {v|false}

ugly x = let y = 0
             z = spin 0
         in x `div` y
```

CBV: Checker *soundly* reports **OK**

CBN: Binders ***may not be*** values

How to *encode* “x has a value” in CBN ?

1 Motivation
How to refine types under *CBN* ?

2 Refinement Typing 101
How to *encode* “x has a value” in CBN ?

How to *encode* “x has a value” in CBN ?

```
  “x has a value”  =>  px
/\“y has a value”  =>  py   =>  p1  =>  p2
/\“z has a value”  =>  pz
```

1. **Ignore** environment
2. **Transform** CBN to CBV
3. **Encode** as logical predicate

How to *encode* “x has a value” in CBN ?

$$
\begin{aligned}
&\boxed{\begin{array}{l}\text{“x has a value”} \\ \text{“y has a value”} \\ \text{“z has a value”}\end{array}}
\end{aligned}
$$

```
   “x has a value”  =>  px
/\ “y has a value”  =>  py   =>  p1  =>  p2
/\ “z has a value”  =>  pz
```

1. ~~**Ignore** environment~~
2. ~~**Transform** CBN to CBV~~
3. ~~**Encode** as logical predicate~~

Non-solutions (see paper)

How to *encode* “x has a value” in CBN ?

Observation:

Most expressions provably *reduce* to a value

How to *encode* “x has a value” in CBN ?

Observation:

Most expressions provably *reduce* to a value

If x reduces to a value,
then *encode* “x has a value” by `true`.

If x reduces to a value,
then *encode* "x has a value" by `true`.

$$\boxed{\texttt{x:\{v:Int|p\}}}$$

Means: "*If* `x` reduces to a value *then* `p[x/v]`"

Encoded as: "`x` has a value" `=>` `p[x/v]`

If x reduces to a value,
then *encode* “x has a value” by `true`.

Label

$$\boxed{\texttt{x:\{v:Int}^{\downarrow}\texttt{|p\}}}$$

Means: “*If* x reduces to a value *then* p[x/v]
and x reduces to a value”

Encoded as: p[x/v]

Stratified Types

$x:\{v:\texttt{Int}^{\downarrow} \mid p\}$

Must reduce to a Value

$x:\{v:\texttt{Int} \mid p\}$

May-not reduce to a Value

Stratified Types to Logic

`x:{v:Int↓ | p}` encoded as `p[x/v]`

`x:{v:Int | p}` encoded as `true`

Code → **Typing**

```
div     :: Int -> {v|v>0} -> Int
succ    :: x:Int -> {v|v>x}
ok n    = let x = 1
              y = succ x
          in n `div` y
```

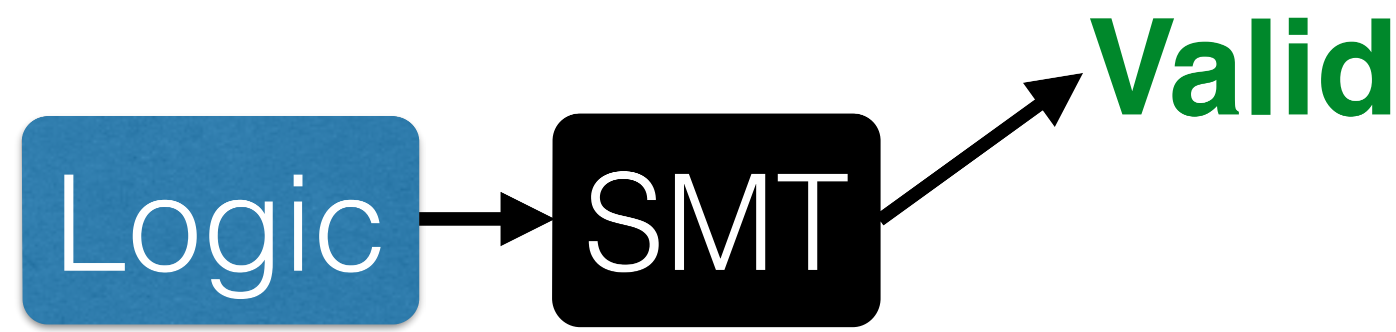
x is a value

```
n:{v:Int|true}
x:{v:Int↓|v=1} |- {v|v>x}<:{v|v>0}
y:{v:Int|v>x }
```

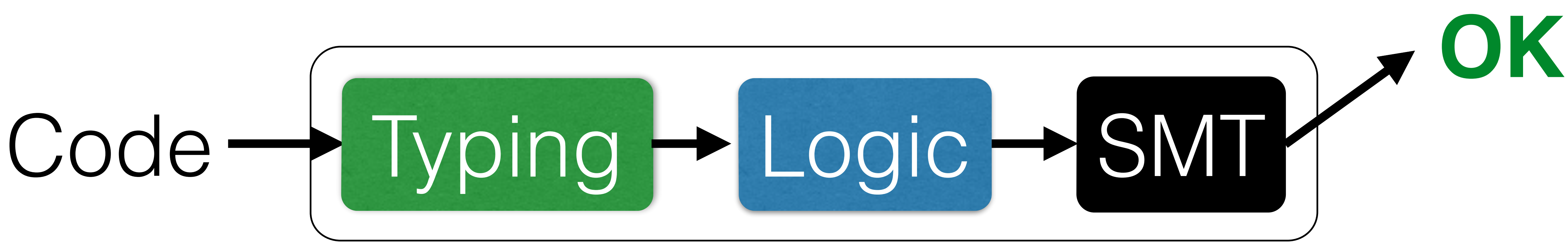
Typing → Logic

```
n:{v:Int|true}
x:{v:Int↓|v=1} |- {v|v>x}<:{v|v>0}
y:{v:Int|v>x }
```

```
   true
/\ x=1          => v>x => v>0
/\ true
```




```
   true
/\ x=1          => v>x => v>0
/\ true
```




```
div     :: Int -> {v|v>0} -> Int
succ    :: x:Int -> {v|v>x}
ok n    = let x = 1
              y = succ x
          in n `div` y
```

x is a value

1 Motivation

How to refine types under *CBN* ?

2 Refinements 101

How to *encode* “x has a value” in CBN ?

3 Stratification

How to *enforce stratification* ?

How to *enforce stratification* ?

$\texttt{x:\{v:Int}^{\downarrow}\texttt{|}\,p\texttt{\}}$

Must have a Value

Terminating expressions must have a value

Solution: Use termination analysis

How to Verify Termination?

Check termination with Refinement Types

Check termination with Refinement Types

```
f   :: n:{v:Int↓|0<=v} -> {v:Int↓|v=1}
f n = if n == 0 then 1 else f (n - 1)
```

`(f n)` has a value, if `f`

Recurses on smaller inputs with a lower bound

Check termination with Refinement Types

```
f   :: n:{v:Int↓|0<=v} -> {v:Int↓|v=1}
f n = if n == 0 then 1 else f (n - 1)
```

Recurses on smaller inputs with a lower bound

Recurses on inputs `v` s.t. `v < n` and `0 <= v`

Check termination with Refinement Types

```
f   :: n:{v:Int↓|0<=v} -> {v:Int↓|v=1}
f n = if n == 0 then 1 else f (n - 1)
```

Recurses on inputs `v` s.t. `v < n` and `0 <= v`

Recurses on inputs `v:{v:Int↓|0<=v /\ v<n}`

Check termination with Refinement Types

```
f   :: n:{v:Int↓|0<=v} -> {v:Int↓|v=1}
f n = if n == 0 then 1 else f (n - 1)
```

Just check f's recursive calls with type

```
f :: {v:Int↓|0<=v /\ v<n}  -> {v:Int↓|v=1}
```

OK: Verifies `(f n)` has a value

Check termination with Refinement Types

Termination Proofs are Semantic

We proved `Greater Common Divisor` terminates, using properties of `mod`.

We cannot always prove termination

We cannot always prove termination

Cannot verify `collatz` terminates...

```
div     :: Int -> {v|v>0} -> Int
collatz :: Int -> {v|v=1}

ok1 n = let y = collatz n in
            42 `div` y
```

`y::{v|v=1}`

Termination is a Luxury not Necessity

We can check “non-terminating” code

1 Motivation

How to refine types under *CBN* ?

2 Refinements 101

How to *encode* “x has a value” in CBN ?

3 Stratification

How to *enforce labels* ?

4 Termination

How is termination *in practice*?

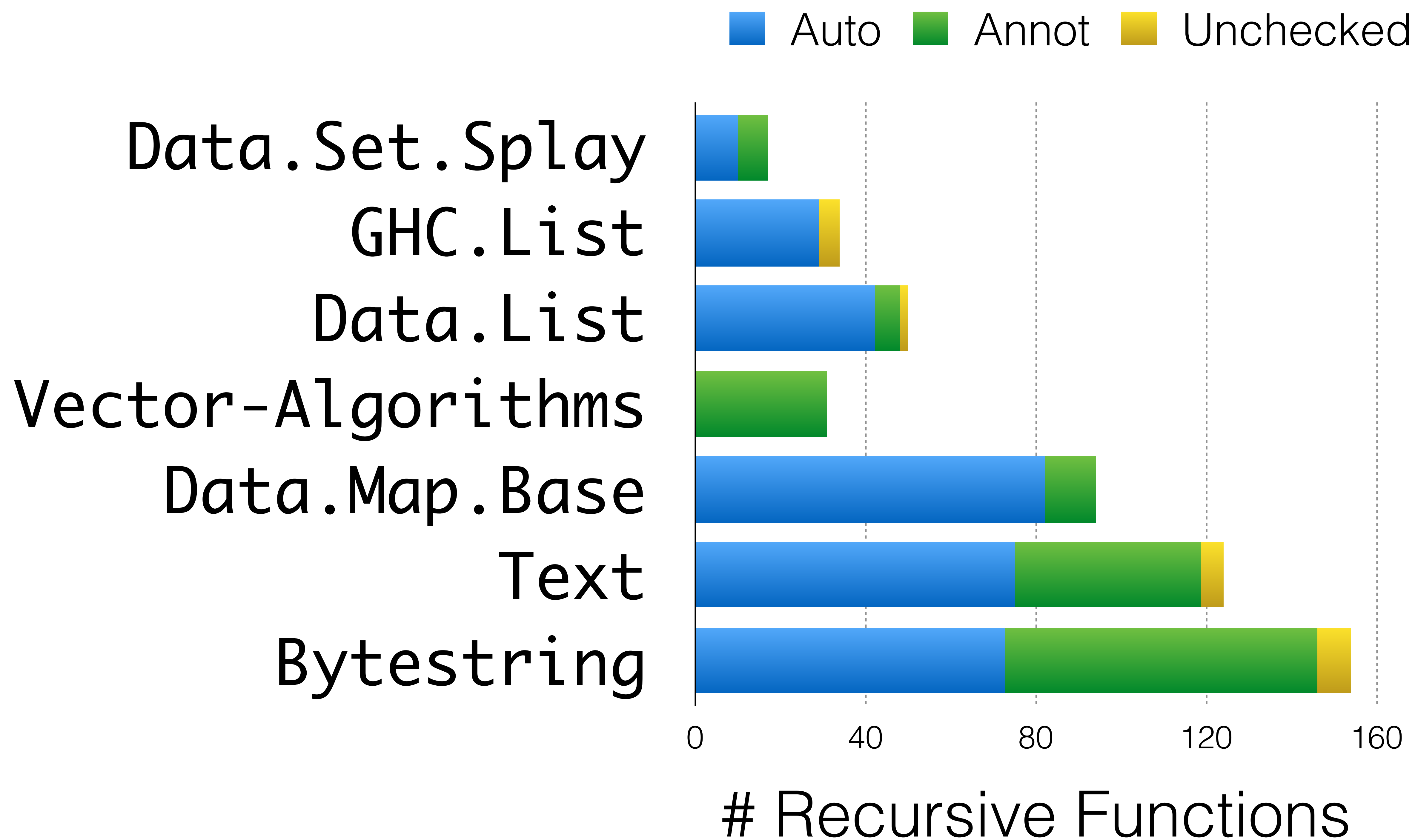
How is termination *in practice*?

LiquidHaskell

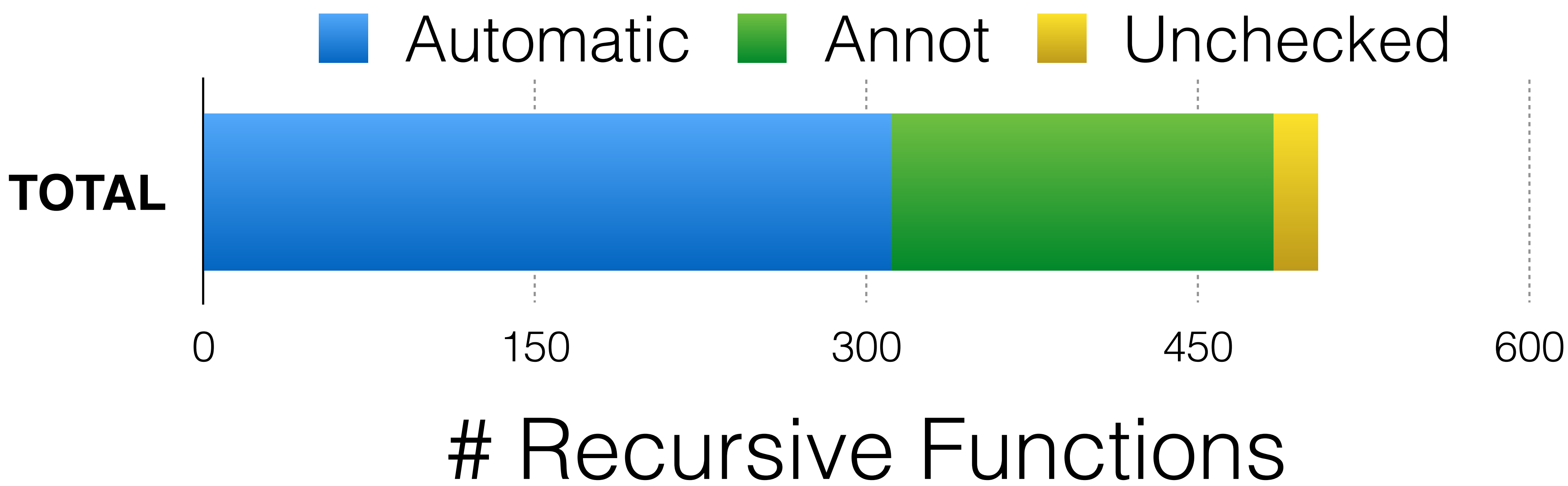
Refinement Type Checker for Haskell

See Eric's HS Talk Tomorrow!

How is termination *in practice*?



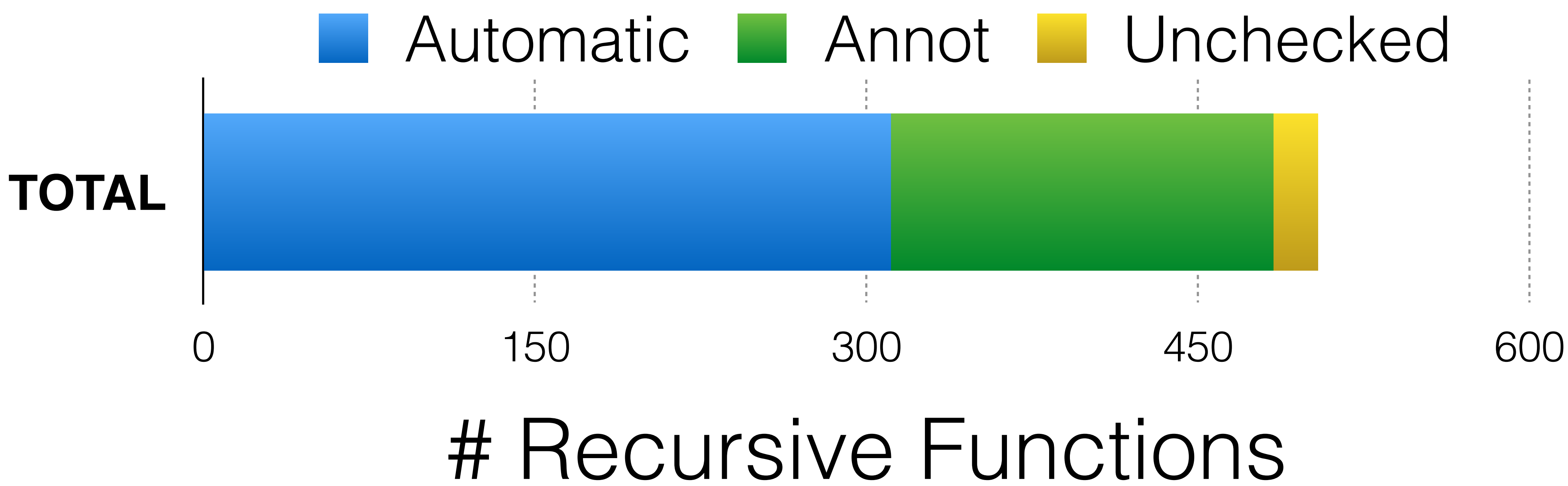
How is termination *in practice*?



Precise

96% recursive functions *proved terminating*

How is termination *in practice*?



Automatic

61% functions proved *automatically*

1 Motivation

How to refine types under *CBN* ?

2 Refinements 101

How to *encode* “x has a value” in CBN ?

3 Stratification

How to *enforce labels* ?

4 Termination

Is termination *practical* ? *Yes*.

5 Evaluation

In The Paper!

- Formalism & Soundness
- Stratified Algebraic Data Types
- Encode Infinite Data
- More Termination Proofs

Refinement Types for Haskell

Problem

Under CBN, Refinement Types “*need*” termination.

Solution

Prove termination *using* Refinement Types…

Evaluation

Which is highly *effective in practice.*

Thank you!

END