Refinement Types For Haskell

Presented by Gan Shen

Diverge vs. Converge

- Diverge = Does Not Terminate
- Converge = Terminate

```
fact(n) = {
  if (n <= 0)
    return 1;
  else
    return n * fact(n+1);
}

fact(5) -> fact(6) ->
fact(7) -> ...
```

Eager vs. Lazy

- Eager (Strict) Evaluation = Call-by-value
- Lazy Evaluation = Call-by-name(need)
- Value is the endpoint of evaluation
- x is a value = x is convergent

Eager vs. Lazy

```
double(x) = x + x;
```

Eager:

```
double(3+5) -> double(8) -> 8 + 8 -> 16
```

Lazy:

```
double(3+5) -> (3+5)+(3+5) -> 8 + 8 -> 16
```

Eager vs. Lazy

```
f(x,y) = y + 1;
```

Eager:

```
f(1/0, 3) -> crash!
```

Lazy:

```
f(1/0, 3) -> 3 + 1 -> 4
```

Overview

- Bring refinement types (Liquid Types) to Haskell
- The classical translation of refinement types to verification conditions is unsound under lazy evaluation
- Reason about divergent binders
- Stratified type system

Recap

- Refinement (Liquid) types compose types with SMT-decidable refinement predicates

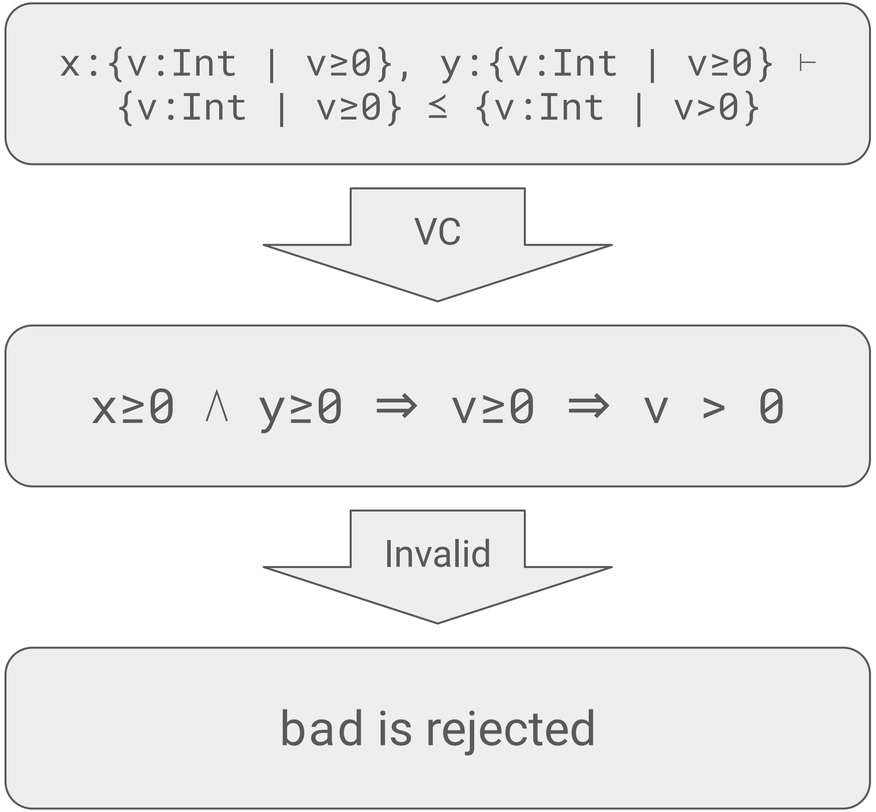
```
data Pos = {v:Int | v > 0}

data Nat = {v:Int | v >= 0}
```

- Encode pre- and post- conditions using refined function types

```
div :: n:Nat -> d:Pos -> {v:Nat | v <= n}
```

Recap

```
bad :: Nat -> Nat -> Int
bad x y = x `div` y
```



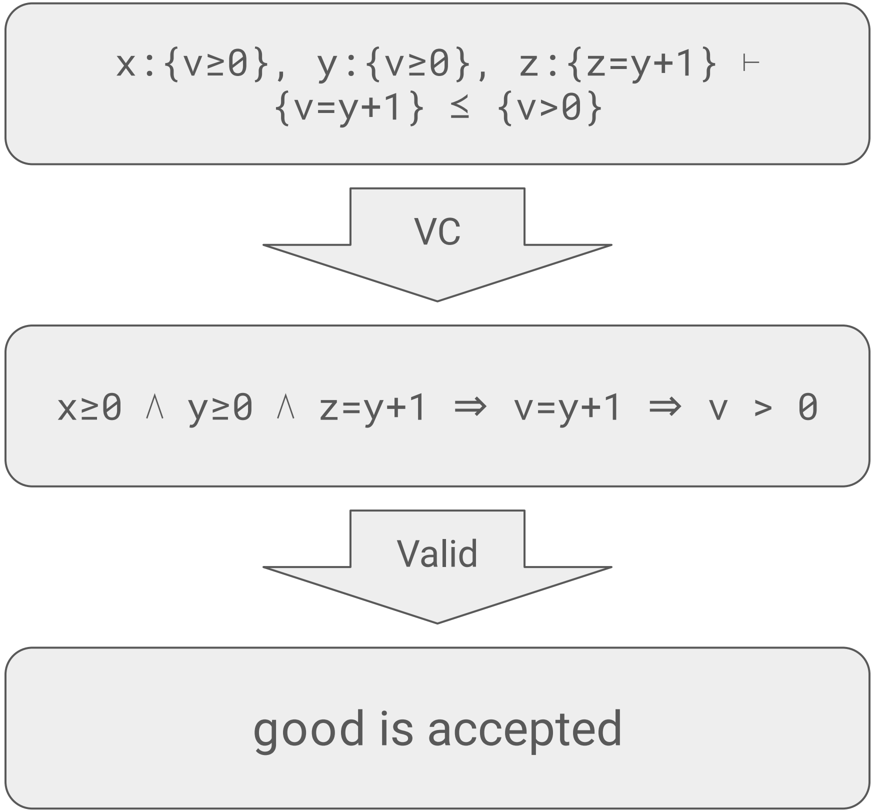
```
x:{v:Int | v≥0}, y:{v:Int | v≥0} ⊢
  {v:Int | v≥0} ≤ {v:Int | v>0}
```

$x \geq 0 \land y \geq 0 \Rightarrow v \geq 0 \Rightarrow v > 0$

bad is rejected

Recap

```haskell
good :: Nat -> Nat -> Int
good x y = let z = y + 1
               in x `div` z
```



```
x:{v≥0}, y:{v≥0}, z:{z=y+1} ⊢
{v=y+1} ≤ {v>0}
```

```
x≥0 ∧ y≥0 ∧ z=y+1 ⇒ v=y+1 ⇒ v > 0
```

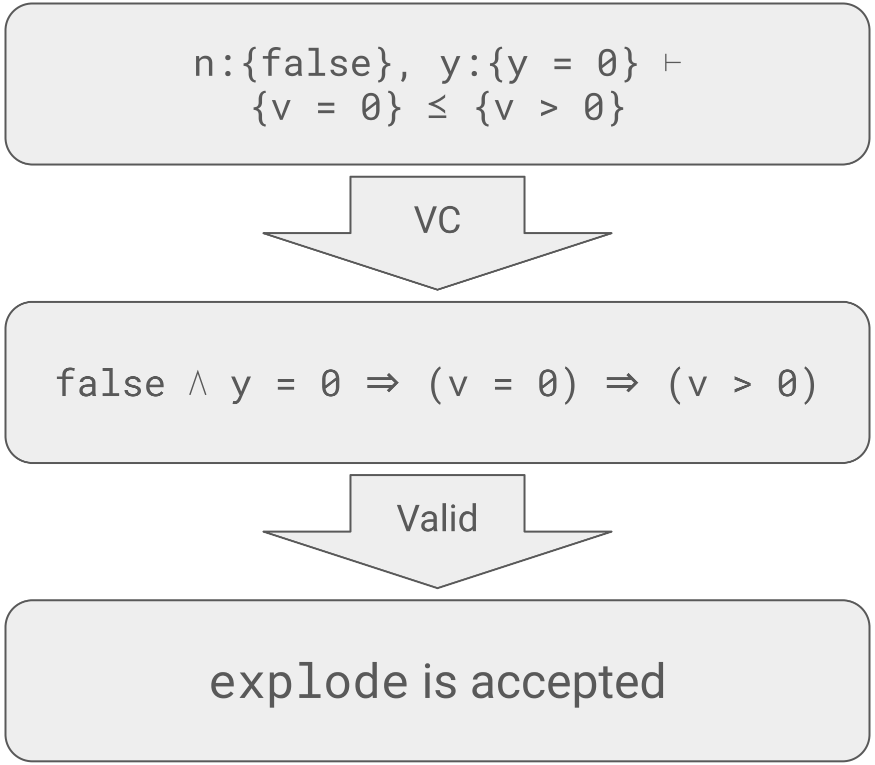
good is accepted

Subtyping To Verification Conditions

$$
\begin{array}{rcl}
\llparenthesis \Gamma \vdash b_1 \preceq b_2 \rrparenthesis & \doteq & \llparenthesis \Gamma \rrparenthesis \Rightarrow \llparenthesis b_1 \rrparenthesis \Rightarrow \llparenthesis b_2 \rrparenthesis \\
\llparenthesis \{x{:}\mathtt{Int} \mid r\} \rrparenthesis & \doteq & r \\
\llparenthesis x{:}\{v{:}\mathtt{Int} \mid r\} \rrparenthesis & \doteq & r\,[x/v] \\
\llparenthesis x{:}(y{:}\tau_y \to \tau) \rrparenthesis & \doteq & \mathtt{true} \\
\llparenthesis x_1{:}\tau_1, \ldots, x_n{:}\tau_n \rrparenthesis & \doteq & \llparenthesis x_1{:}\tau_1 \rrparenthesis \wedge \ldots \wedge \llparenthesis x_n{:}\tau_n \rrparenthesis
\end{array}
$$

Unsound Under Lazy Evaluation

```
diverge :: Int -> {v:Int | false}
diverge n = diverge n

explode :: Int -> Int
explode x = let { n = diverge 1;
                  y = 0 }
            in x `div` y
```



Unsound Under Lazy Evaluation

```
diverge :: Int -> {v:Int | false}
diverge n = diverge n

explode :: Int -> Int
explode x = let { n = diverge 1;
                  y = 0 }
            in x `div` y
```

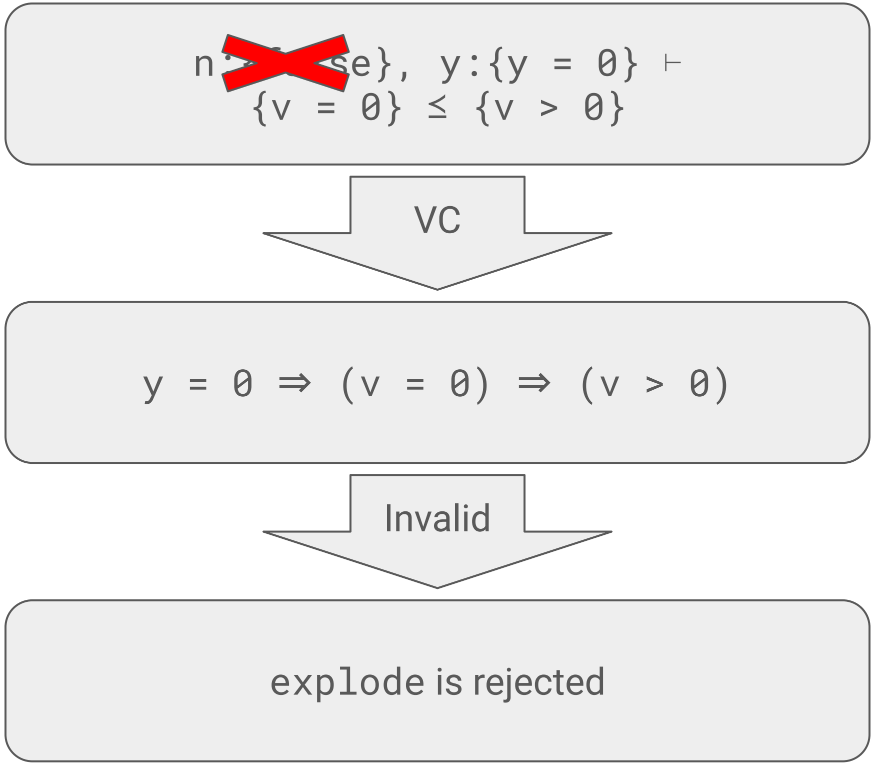
Under eager evaluation, program is stuck here, so the divide by zero error is never triggered.

Under lazy evaluation, program can pass here, so the divide by zero error is triggered.

In the eager setting, the inconsistent antecedent in the VC logically encodes the fact that the call to `div` is *dead code*. However, this conclusion is wrong under Haskell's lazy semantics.

Fix

```
diverge :: Int -> {v:Int | false}
diverge n = diverge n

explode :: Int -> Int
explode x = let { n = diverge 1;
                  y = 0 }
            in x `div` y
```



~~n:{false}~~, y:{y = 0} ⊢
{v = 0} ≤ {v > 0}

VC

y = 0 ⇒ (v = 0) ⇒ (v > 0)

Invalid

explode is rejected

Adding Guards To Translation

$$
\begin{aligned}
\llparenthesis \Gamma \vdash b_1 \preceq b_2 \rrparenthesis &\doteq \llparenthesis \Gamma \rrparenthesis \Rightarrow \llparenthesis b_1 \rrparenthesis \Rightarrow \llparenthesis b_2 \rrparenthesis \\
\llparenthesis \{x{:}\mathtt{Int} \mid r\} \rrparenthesis &\doteq r \\
\llparenthesis x{:}\{v{:}\mathtt{Int} \mid r\} \rrparenthesis &\doteq r\,[x/v] \\
\llparenthesis x{:}(y{:}\tau_y \rightarrow \tau) \rrparenthesis &\doteq \mathtt{true} \\
\llparenthesis x_1{:}\tau_1, \ldots, x_n{:}\tau_n \rrparenthesis &\doteq \llparenthesis x_1{:}\tau_1 \rrparenthesis \wedge \ldots \wedge \llparenthesis x_n{:}\tau_n \rrparenthesis
\end{aligned}
$$

Reasoning About Divergence

One straightforward solution:

Enrich the underlying logic with a new predicate “x is a value”, then use SMT solvers to reason about it.

Problems:

1. Require three-valued logics.
2. Cannot re-use existing SMT solvers.

Reason about divergence inside the type system.

Stratified Type System

- Label each type with label
  - τ Div-type, types may diverge
  - $\tau^{\downarrow}$ Wnf-type, types will reduce to Weak Head Normal Form (WHNF)
  - $\tau^{\Downarrow}$ Fin-type, types will reduce to finite values with no redexes
- Omit any refinement predicates for binders may diverge

$$(\!| x{:}\{v{:}B \mid r\} |\!) \doteq \begin{cases} \texttt{true}, & \text{if } B \text{ is a Div type} \\ r\,[x/v]\,, & \text{otherwise} \end{cases}$$

Stratified Type System (Cont.)

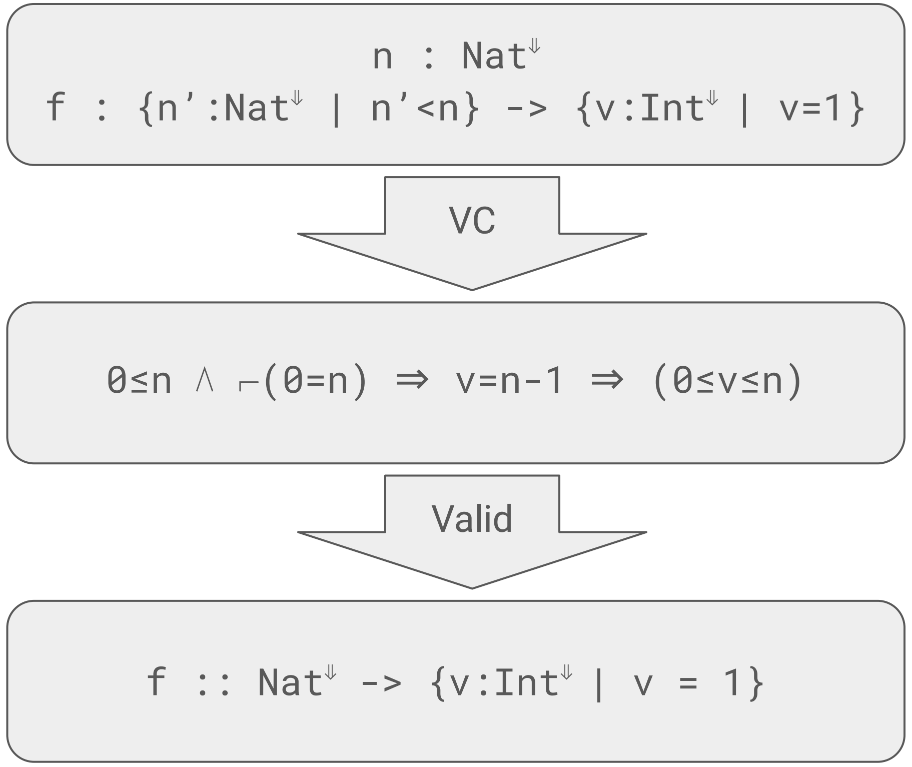
- Type most primitives non-Div type, since they don't diverge
  - Integers, booleans, primitive operations (+, -, / …)...

$$
\begin{aligned}
\mathsf{Ty}(n) &\doteq \{v{:}\mathtt{Int}^{\Downarrow} \mid v = n\} \\
\mathsf{Ty}(=) &\doteq x{:}B^{\Downarrow} \rightarrow y{:}B^{\Downarrow} \rightarrow \{v{:}\mathtt{Bool}^{\Downarrow} \mid v \Leftrightarrow x = y\} \\
\mathsf{Ty}(+) &\doteq x{:}\mathtt{Int}^{\Downarrow} \rightarrow y{:}\mathtt{Int}^{\Downarrow} \rightarrow \{v{:}\mathtt{Int}^{\Downarrow} \mid v = x + y\} \\
\mathsf{Ty}(\wedge) &\doteq x{:}\mathtt{Bool}^{\Downarrow} \rightarrow y{:}\mathtt{Bool}^{\Downarrow} \rightarrow \{v{:}\mathtt{Bool}^{\Downarrow} \mid v \Leftrightarrow x \wedge y\}
\end{aligned}
$$

- Type constructs that force evaluation non-Div type
  - `seq` operator, bang-pattern, `case-of` primitive…
- Type recursive functions based on termination verification
  - Prove termination via decreasing metrics.
  - Each recursive call is made with arguments of a strictly smaller size, where the size is itself a well founded metric, e.g. a natural number.

Function Termination Verification

```
f :: Nat -> {v:Int | v = 1}

f n = if n == 0

      then 1

      else f (n - 1)
```



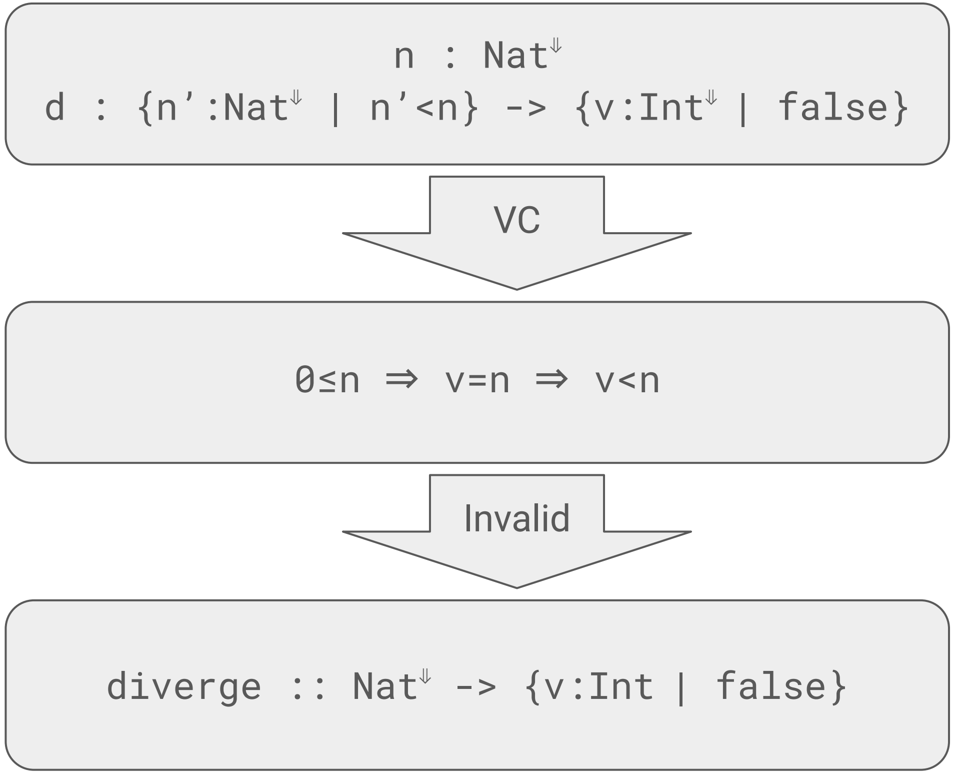
```
n : Nat⇓
f : {n':Nat⇓ | n'<n} -> {v:Int⇓ | v=1}
```

```
0≤n ∧ ¬(0=n) ⇒ v=n-1 ⇒ (0≤v≤n)
```

```
f :: Nat⇓ -> {v:Int⇓ | v = 1}
```

Example

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

diverge n = diverge n
```



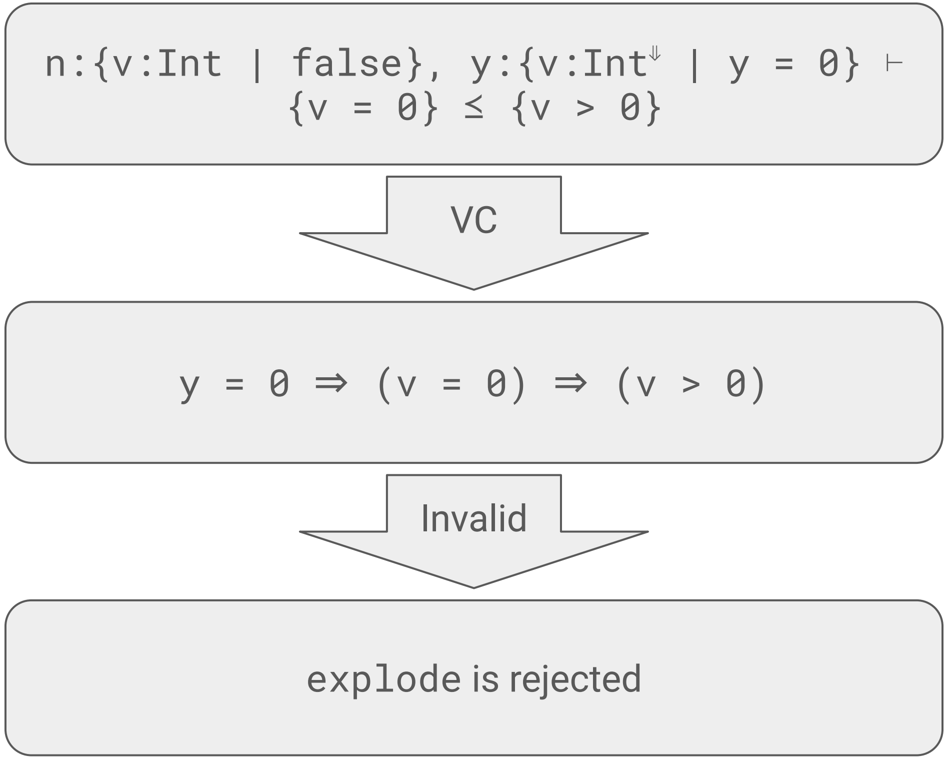
```
n : Nat⇓
d : {n':Nat⇓ | n'<n} -> {v:Int⇓ | false}
```

```
0≤n ⇒ v=n ⇒ v<n
```

```
diverge :: Nat⇓ -> {v:Int | false}
```

Example (Cont.)

```
diverge :: Nat⇓ -> {v:Int | false}
diverge n = diverge n

explode :: Int⇓ -> Int⇓
explode x = let { n = diverge 1;
                  y = 0 }
            in x `div` y
```



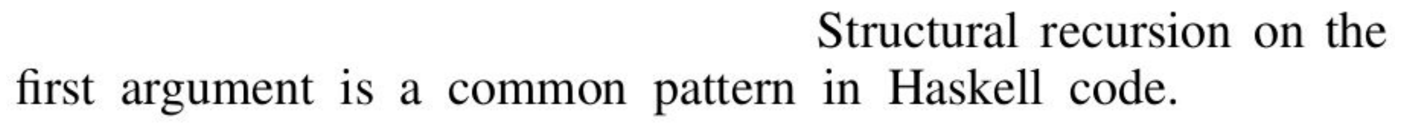
```
n:{v:Int | false}, y:{v:Int⇓ | y = 0} ⊢
{v = 0} ≤ {v > 0}
```

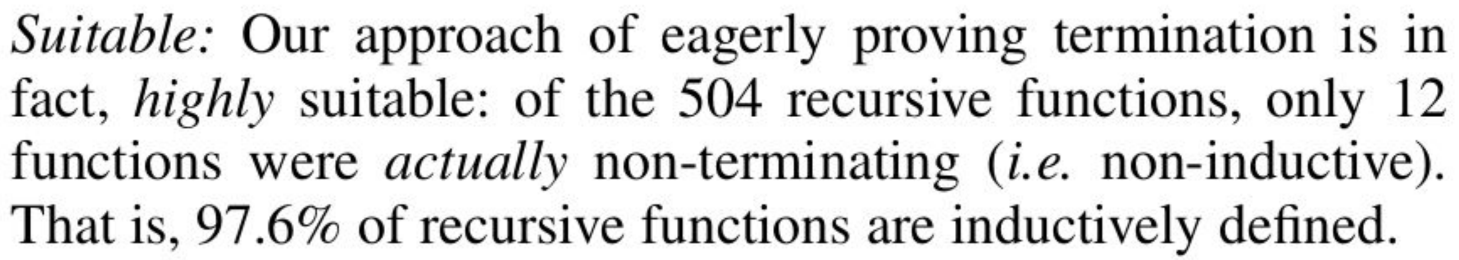
```
y = 0 ⇒ (v = 0) ⇒ (v > 0)
```

explode is rejected

Discussion

- The termination verification only works with functions recursing over well-founded metrics (e.g. natural numbers, lists), how does this affect the utility of the verifier?


Structural recursion on the first argument is a common pattern in Haskell code.


Suitable: Our approach of eagerly proving termination is in fact, *highly* suitable: of the 504 recursive functions, only 12 functions were *actually* non-terminating (*i.e.* non-inductive). That is, 97.6% of recursive functions are inductively defined.

Discussion

- Is it possible to explicitly reason about divergence in the refinement logic (reason about divergence using SMT solvers)?

$$\begin{array}{l}(\mathsf{Val}(x) \Rightarrow x \geq 0) \\ (\mathsf{Val}(y) \Rightarrow y \geq 0)\end{array} \Rightarrow (v = y + 1) \Rightarrow (v > 0)$$

Discussion

- Strongly normalizing languages?