Static Contract Checking for Haskell

Dana N. Xu

University of Cambridge

Joint work with

Simon Peyton Jones
Microsoft Research Cambridge

Koen Claessen
Chalmers University of Technology

Program Errors Give Headache!

```
Module UserPgm where

f :: [Int]->Int
f xs = head xs `max` 0

     :
     … f [] …
```

```
Module Prelude where

head :: [a] -> a
head (x:xs) = x
head [] = error "empty list"
```

Glasgow Haskell Compiler (GHC) gives at run-time

```
Exception: Prelude.head: empty list
```

Types >>>>>> Contracts >>>>>>

```
head (x:xs) = x

head :: [Int] -> Int

…(head 1)…
```

Type

Bug!

```
not :: Bool -> Bool
not True = False
not False = True

null :: [a] -> Bool
null [] = True
null (x:xs) = False
```

```
head ∈ {xs | not (null xs)} -> {r | True}
```

Contract (original Haskell boolean expression)

```
…(head [])…
```

Bug!

Contract Checking

```
head ∈ {xs | not (null xs)} -> {r | True}
head (x:xs') = x

f xs = head xs `max` 0

Warning: f [] calls head
         which may fail head's precondition!

f_ok xs = if null xs then 0
          else head xs `max` 0
```

No more warnings from compiler!

Satisfying a predicate contract

Arbitrary boolean-valued Haskell expression

e ∈ {x | p} if (1) p[e/x] gives True *and*

(2) e is crash-free.

```
Recursive function,
higher-order function,
partial function
can be called!
```

Expressiveness of the Specification Language

```
data T = T1 Bool | T2 Int | T3 T T

sumT :: T -> Int
sumT ∈ {x | noT1 x} -> {r | True}
sumT (T2 a) = a
sumT (T3 t1 t2) = sumT t1 + sumT t2

noT1 :: T -> Bool
noT1 (T1 _) = False
noT1 (T2 _) = True
noT1 (T3 t1 t2) = noT1 t1 && noT1 t2
```

Expressiveness of the Specification Language

```
sumT :: T -> Int
sumT ∈ {x | noT1 x} -> {r | True}
sumT (T2 a) = a
sumT (T3 t1 t2) = sumT t1 + sumT t2

rmT1 :: T -> T
rmT1 ∈ {x | True} -> {r | noT1 r}
rmT1 (T1 a) = if a then T2 1 else T2 0
rmT1 (T2 a) = T2 a
rmT1 (T3 t1 t2) = T3 (rmT1 t1) (rmT1 t2)
```

For all crash-free `t::T`, `sumT (rmT1 t)` will not crash.

Higher Order Functions

```
all :: (a -> Bool) -> [a] -> Bool
all f [] = True
all f (x:xs) = f x && all f xs

filter :: (a -> Bool) -> [a] -> [a]
filter ∈ {f | True} -> {xs | True} -> {r | all f r}
filter f [] = []
filter f (x:xs') = case (f x) of
                     True -> x : filter f xs'
                     False -> filter f xs'
```

Contracts for higher-order function's parameter

```
f1 :: (Int -> Int) -> Int
f1 ∈ ({x | True} -> {y | y >= 0}) -> {r | r >= 0}
f1 g = (g 1) - 1

f2:: {r | True}
f2 = f1 (\x -> x - 1)
```

```
Error: f1's postcondition fails
       because (g 1) >= 0 does not imply
               (g 1) - 1 >= 0
Error: f2 calls f1
       which fails f1's precondition
```


[Findler&Felleisen:ICFP'02, Blume&McAllester:ICFP'04]

Functions without Contracts

```
data T = T1 Bool | T2 Int | T3 T T

noT1 :: T -> Bool
noT1 (T1 _) = False
noT1 (T2 _) = True
noT1 (T3 t1 t2) = noT1 t1 && noT1 t2

(&&) True x = x
(&&) False x = False
```

No abstraction is more compact than the function definition itself!

Contract Synonym

```
contract Ok = {x | True}
contract NonNull = {x | not (null x)}

head :: [Int] -> Int
head ∈ NonNull -> Ok
head (x:xs) = x
```

Actual Syntax

```
{-# contract Ok = {x | True} -#}
{-# contract NonNull = {x | not (null x)} #-}
{-# contract head :: NonNull -> Ok #-}
```

Questions on e ∈ t

bot = bot

λ x. x	**∈**	**{x | bot} → {r | True}**	**?**
λ x. error "f"	**∈**	**{x | bot} → {r | True}**	**?**
λ x. x	**∈**	**{x | True} → {r | bot}**	**?**
λ x. head []	**∈**	**{x | True} → {r | bot}**	**?**
(True, 2)	**∈**	**{x | (snd x) > 0}**	**?**
(head [], 3)	**∈**	**{x | (snd x) > 0}**	**?**
error "f"	**∈**	**?**	
?	**∈**	**{x | False}**	
?	**∈**	**{x | head []}**	

Language Syntax

following Haskell's lazy semantics

pgm	$\in$	**Program**	
pgm	$:=$	$def_1, \ldots, def_n$	
def	$\in$	**Definition**	
def	$:=$	$decl$	
	$\mid$	$f \in t$	**Contract attribution**
	$\mid$	$f\ \vec{x} = e$	**Top-level definition**
$decl$	$\in$	**Data Type**	
$decl$	$:=$	**data** $T\ \vec{\alpha}$ **where** $\overrightarrow{K \in \vec{\tau_i} \rightarrow T\ \vec{\alpha}}$	**Data Constructors**
a, e, p	$\in$	**Exp**	**Expression**
a, e, p	$::=$	n	**Integers**
	$\mid$	$v \mid \lambda(x :: \tau).e \mid e_1\ e_2$	
	$\mid$	**case** e_0 **of** $alts$	
	$\mid$	$K\ \vec{e}$	**Constructor**
	$\mid$	**BAD**	**A crash**
	$\mid$	**UNR**	**Unreachable**
$alts$	$::=$	$alt_1 \ldots alt_n$	
alt	$::=$	$pt \rightarrow e$	**Case alternative**
pt	$::=$	$K\ (x_1 :: \tau_1) \ldots (x_n :: \tau_n)$	**Pattern**
	$\mid$	**DEFAULT**	
τ	$\in$	**Types**	
τ	$\mid$	**Int** $\mid$ **Bool** $\mid \ldots$	**Base types**
	$\mid$	T	**Data type**
val	$\in$	**Value**	
val	$::=$	$n \mid K\ \vec{e} \mid \lambda(x :: \tau).e \mid$ **UNR** $\mid$ **BAD**	

Two special constructors

- **`BAD` is an expression that *crashes*.**

```
error :: String -> a
error s = BAD

head (x:xs) = x
head [] = BAD
```

- **UNR (short for "unreachable") is an expression that gets stuck. This is *not* a crash, although execution comes to a halt without delivering a result. (identifiable infinite loop)**

Syntax of Contracts

(related to [Findler:ICFP02,Blume:ICFP04,Hinze:FLOPS06,Flanagan:POPL06])

$t \in$ Contract	
$t ::= \{x \mid p\}$	Predicate Contract
$\mid \; x{:}t_1 \rightarrow t_2$	Dependent Function Contract
$\mid \; (t_1, t_2)$	Tuple Contract
$\mid$ Any	Polymorphic Any Contract

Full version: `x′:{x | x >0} -> {r | r > x′}`
Short hand: `{x | x > 0} -> {r | r > x}`

```
k:({x | x > 0} -> {y | y > 0}) -> {r | r > k 5}
```

Contract Satisfaction

(related to [Findler:ICFP02,Blume:ICFP04,Hinze:FLOPS06])

Given $\vdash e :: \tau$ and $\vdash_c t :: \tau$, we define $e \in t$ as follows:

$e \in \{x \mid p\} \quad \Leftrightarrow \quad e\uparrow$ or (e is crash-free and $p[e/x] \not\rightarrow^* \{BAD, False\}$ [A1]

$e \in x{:}t_1 \rightarrow t_2 \quad \Leftrightarrow \quad e\uparrow$ or $\forall\, e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x]$ [A2]

$e \in (t_1, t_2) \quad \Leftrightarrow \quad e\uparrow$ or ($e \rightarrow^* (e_1,e_2)$ and $e_1 \in t_1$ and $e_2 \in t_2$) [A3]

$e \in$ `Any` $\quad \Leftrightarrow \quad$ True [A4]

$e\uparrow$ means e diverges or $e \rightarrow^*$ `UNR`

Answers on e ∈ t

λ x. x ∈ {x | bot} → {r | True} ✓

λ x. BAD ∉ {x | bot} → {r | True}

λ x. x ∈ {x | True} → {r | bot} ✓

λ x. BAD ∉ {x | True} → {r | bot}

λ x. BAD ∈ {x | True} → Any ←

(True, 2) ∈ {x | (snd x) > 0} ✓

(BAD, 3) ∉ {x | (snd x) > 0}

BAD ∈ Any

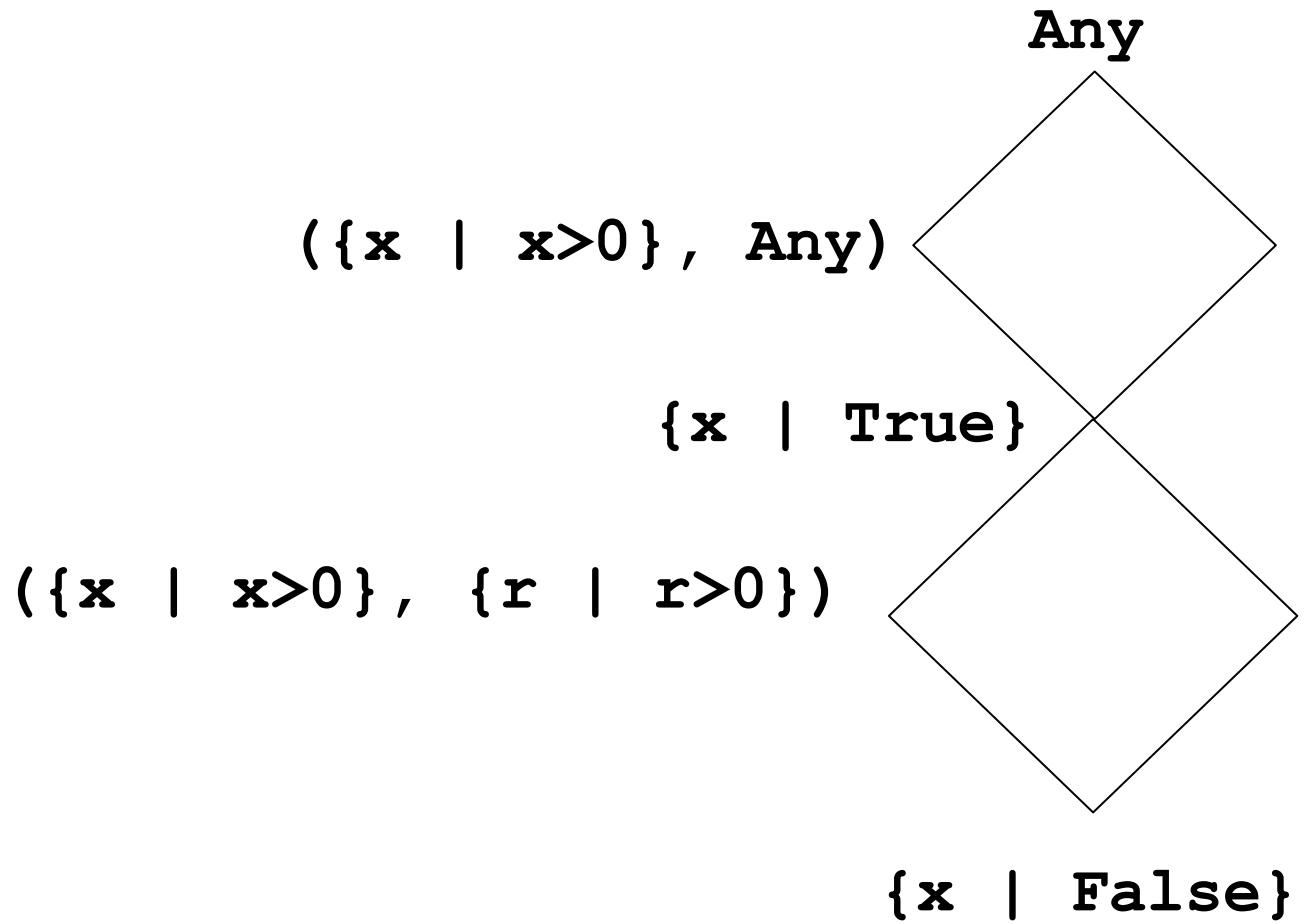
UNR, bot ∈ {x | False}

UNR, bot ∈ {x | BAD}

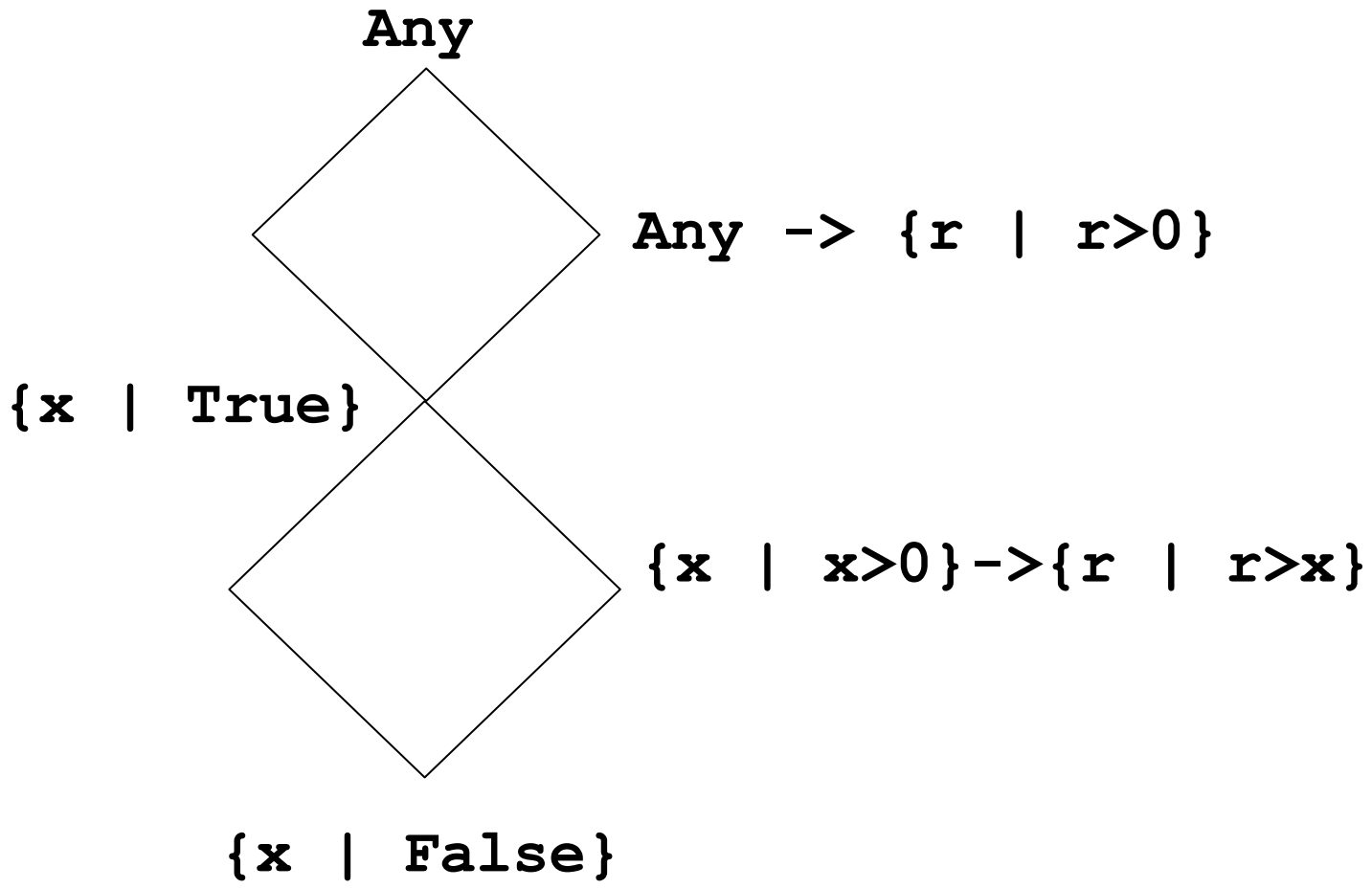
BAD ∉ (Any, Any)

BAD ∉ Any → Any

Lattice for Contracts



Lattice for Contracts



Laziness

```
fst ∈ ({x | True}, Any) -> {r | True}
fst (a,b) = a

…fst (5, error "f")…
```

Every expression satisfies Any

```
fstN :: (Int, Int) -> Int -> Int
fstN ∈ ({x | True}, Any)
     -> {n | True} ->{r | True}
fstN (a, b) n = if n > 0
                then fstN (a+1, b) (n-1)
                else a

g2 = fstN (5, error "fstN") 100
```

What to Check?

Does function *f* satisfies its contract *t* (written $f \in t$)?

At the definition of each function *f*,

Check $f \in t$ assuming all functions called in *f* satisfy their contracts.

Goal: main $\in$ {x | True}

How to Check?

This Talk

Define
`e ∈ t`

Grand Theorem
e ∈ t ⇔ e ▷ t is crash-free

(related to Blume&McAllester:ICFP'04)

ESC/Haskell (Xu:HW'06)

Construct
`e ▷ t`
`(e "ensures" t)`

Simplify (e ▷ t)

Normal form `e'`

If `e'` is syntactically safe, then Done!

What we can't do?

```
g1, g2 ∈ Ok -> Ok
g1 x = case (prime x > square x) of
           True -> x
           False -> error "urk"

g2 xs ys =
  case (rev (xs ++ ys) == rev ys ++ rev xs) of
    True -> xs
    False -> error "urk"
```

Crash!

Crash!

Hence, three possible outcomes:

(1) Definitely Safe (no crash, but may loop)

(2) Definite Bug (definitely crashes)

(3) Possible Bug

Crashing

Definition (Crash).

A closed term e crashes **iff** $e \rightarrow^* \mathbf{BAD}$

Definition (Crash-free Expression)

An expression e is crash-free **iff**

$\forall$ **C. BAD** $\notin$ **C,** $\vdash$ **C[[e]] :: (), C[[e]]** $\not\rightarrow^*$ **BAD**

Definition of ▷ and ◁

(▷ pronounced ensures ◁ pronounced requires)

```
e ▷ {x | p}
= case p[e/x] of
     True -> e
     False -> BAD

e ◁ {x | p}
= case p[e/x] of
     True -> e
     False -> UNR
```

Example:

```
5 ∈ {x | x > 0}

5 ▷ {x | x > 0}
= case (5 > 0) of
     True -> 5
     False -> BAD
```

related to [Findler:ICFP02,Blume:ICFP04,Hinze:FLOPS06]

Definition of $\triangleright$ and $\triangleleft$ (cont.)

$e \triangleright x{:}t_1 \rightarrow t_2$
$= \lambda\ v.\ (e\ (v \triangleleft t_1)) \triangleright t_2[v \triangleleft t_1/x]$

$e \triangleleft x{:}t_1 \rightarrow t_2$
$= \lambda\ v.\ (e\ (v \triangleright t_1)) \triangleleft t_2[v \triangleright t_1/x]$

$e \triangleright (t_1, t_2)$
$=$ case e of
  $(e_1, e_2) \rightarrow (e_1 \triangleright t_1, e_2 \triangleright t_2)$

$e \triangleleft (t_1, t_2)$
$=$ case e of
  $(e_1, e_2) \rightarrow (e_1 \triangleleft t_1, e_2 \triangleleft t_2)$

Higher-Order Function

```
f1 :: (Int -> Int) -> Int
f1 ∈ ({x | True} -> {y | y >= 0}) -> {r | r >= 0}
f1 g = (g 1) - 1

f2:: {r | True}
f2 = f1 (\x -> x - 1)
```

```
f1 ▷ ({x | True} -> {y | y >= 0}) -> {r | r >= 0}
= …   ▷            ◁                  ▷
= λ v1. case (v1 1) >= 0 of
         True -> case (v1 1) - 1 >= 0 of
                   True -> (v1 1) -1
                   False -> BAD
         False -> UNR
```

Modified Definition of ▷ and ◁

```
e ▷ {x | p}
= case p[e/x] of
      True -> e
      False -> BAD
```

old

if `e` $\rightarrow^*$ `BAD`,

then `(fin e)` $\rightarrow$ `False`

else `(fin e)` $\rightarrow$ `e`

```
e ▷ {x | p}
= e `seq` case fin (p[e/x]) of
              True -> e
              False -> BAD
```

new

```
e_1 `seq` e_2 = case e_1 of {DEFAULT -> e_2}
```

Why need `seq` and `fin`?

```
e ▷ {x | p}
= e `seq` case fin (p[e/x]) of
              True -> e
              False -> BAD
```

Without seq

(UNR ▷ {x | False}) →* BAD

But UNR ∈ {x | False}

Without fin

λx. x ∈ {x | BAD} → {x | True}

But λ x. x ▷ {x | BAD} → {x | True}

→* λ v. (v `seq` BAD) which is NOT crash-free.

Definition of ▷ and ◁ for Any

```
e ▷ Any = UNR
e ◁ Any = BAD
```

```
f5 ∈ Any -> {r | True}
f5 x = 5

error :: String -> a          -- HM Type
error ∈ {x | True} -> Any     -- Contract
```

Properties of $\triangleright$ and $\triangleleft$

Key Lemma:
For all closed, crash-free *e*, and closed *t*,
$(e \triangleleft t) \in t$

Projections: (related to Findler&Blume:FLOPS'06)
For all e and t, if $e \in t$, then

(a) $e \preceq e \triangleright t$

(b) $e \triangleleft t \preceq e$

Definition (Crashes-More-Often):
$e_1 \preceq e_2$ **iff for all C,** $\vdash C[[e_i]] :: ()$ **for i=1,2 and**
$C[[e_2]] \rightarrow^* BAD \Rightarrow C[[e_1]] \rightarrow^* BAD$

More Lemmas ☺

Lemma [Monotonicity of Satisfaction]:

If $e_1 \in t$ and $e_1 \preceq e_2$, then $e_2 \in t$

Lemma [Congruence of $\preceq$]:

$e_1 \preceq e_2 \Rightarrow \forall C.\ C[[e_1]] \preceq C[[e_2]]$

Lemma [Idempotence of Projection]:

$\forall e, t.\ \ e \rhd t \rhd t \equiv e \rhd t$

$\forall e, t.\ \ e \lhd t \lhd t \equiv e \lhd t$

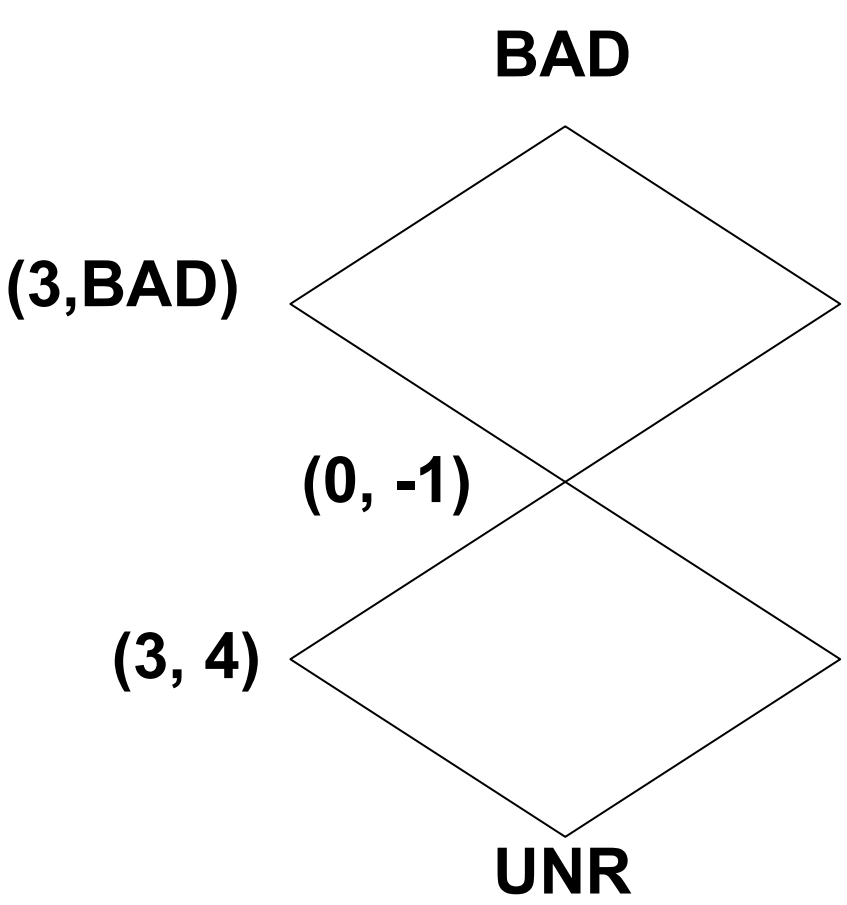
Lemma [A Projection Pair]:

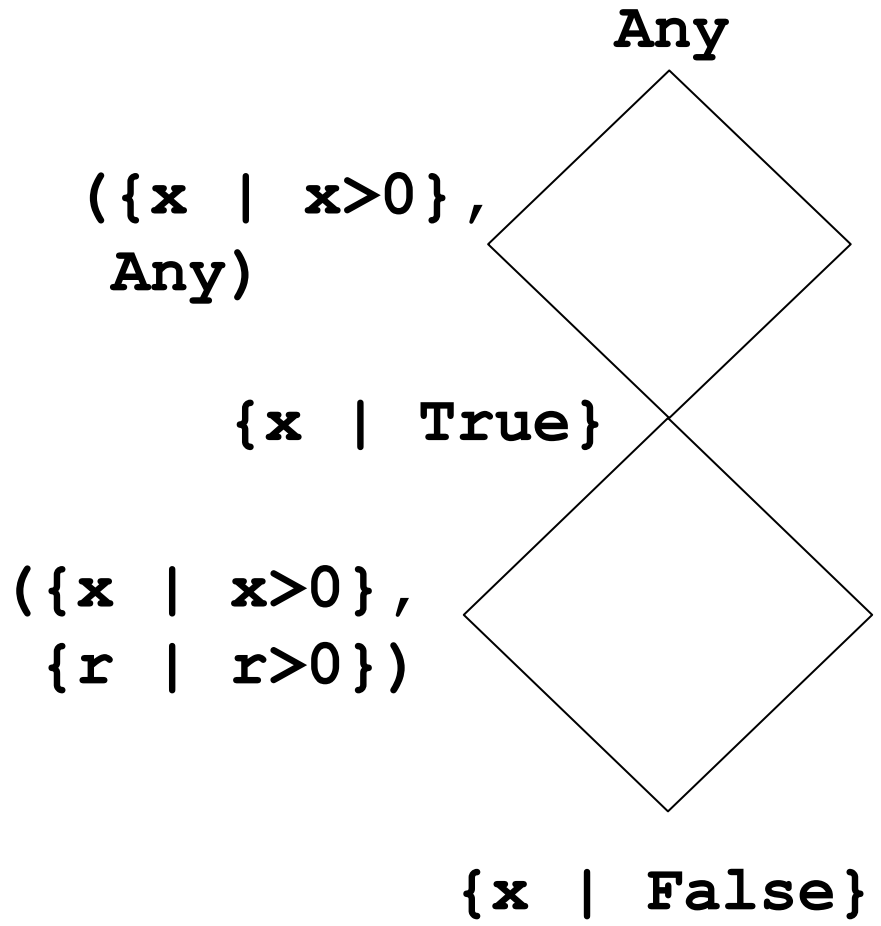
$\forall e, t.\ \ e \rhd t \lhd t \preceq e$

Lemma [A Closure Pair]:

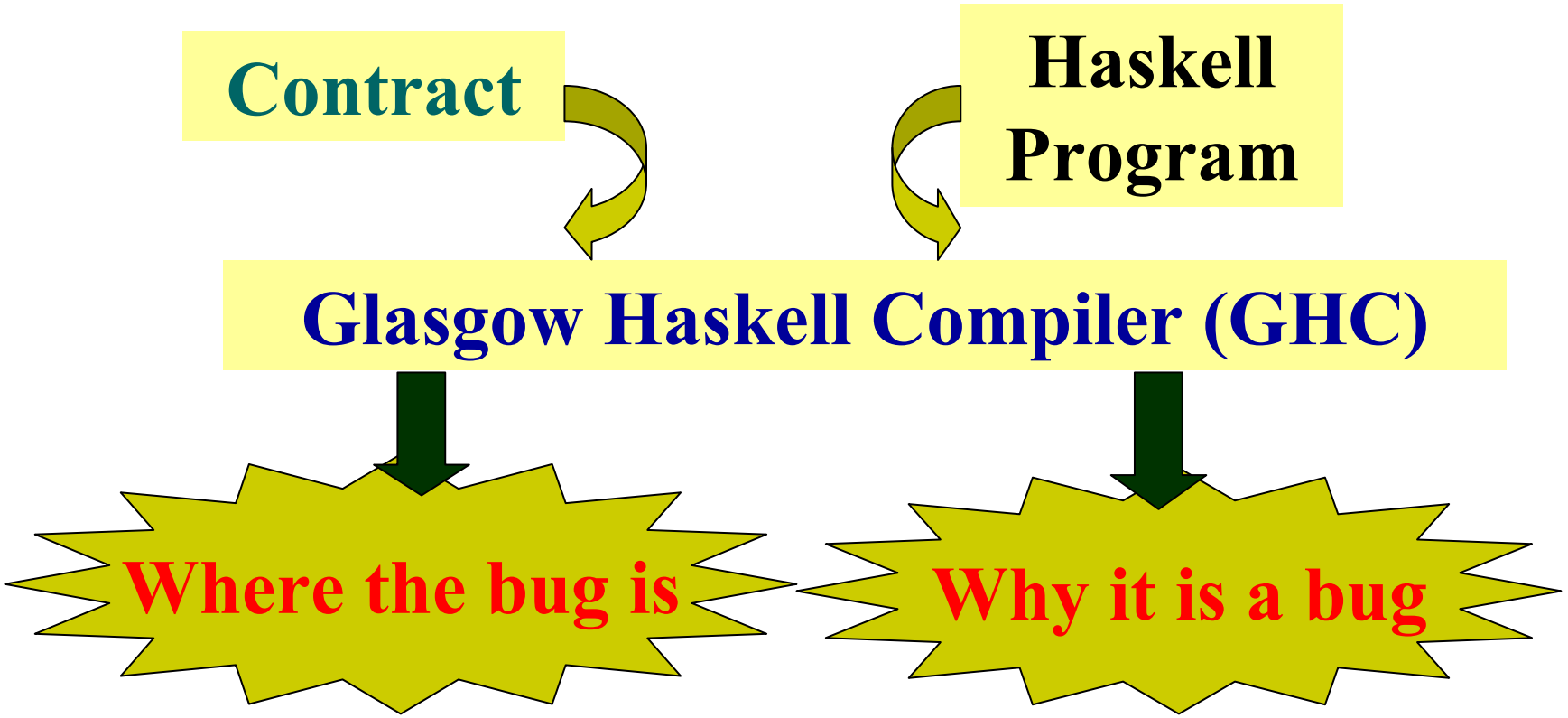
$\forall e, t.\ \ e \preceq e \lhd t \rhd t$

Lattice for Expressions ($\preceq$)





Conclusion


Contract
Haskell Program
Glasgow Haskell Compiler (GHC)
Where the bug is
Why it is a bug

Various Examples

```
zip :: [a] -> [b] -> [(a,b)]
zip :: {xs | True} -> {ys | sameLen xs ys}
    -> {rs | sameLen rs xs }

sameLen [] [] = True
sameLen (x:xs) (y:ys) = sameLen xs ys
sameLen _ _ = False
```

```
f91 :: Int -> Int
f91 :: { n <= 101 } -> {r | r == 91 }
f91 n = case (n <= 100) of
          True -> f91 (f91 (n + 11))
          False -> n - 10
```

Sorting

```
(==>) True x = x
(==>) False x = True
```

```
sorted [] = True
sorted (x:[]) = True
sorted (x:y:xs) = x <= y && sorted (y : xs)

insert :: {i | True} -> {xs | sorted xs}
       -> {r | sorted r}

merge :: [Int] -> [Int] -> [Int]
merge :: {xs | sorted xs} -> {ys | sorted ys}
      -> {r | sorted r}

bubbleHelper :: [Int] -> ([Int], Bool)
bubbleHelper :: {xs | True}
             -> {r | not (snd r) ==> sorted (fst r)}

Insertsort, mergesort, bubblesort :: {xs | True}
                                  -> {r | sorted r}
```

Contributions

- Automatic static contract checking instead of dynamic contract checking.
- Compared with ESC/Haskell
  - Allow pre/post specification for higher-order function's parameter through contracts.
  - Reduce more false alarms caused by Laziness and in an efficient way.
  - Allow user-defined data constructors to be used to define contracts
  - Specifications are more type-like, so we can define contract synonym.
- We develop a concise notation ($\triangleright$ and $\triangleleft$) for contract checking, which enjoys many properties. We give a new and relatively simpler proof of the soundness and completeness of dynamic contract checking, this proof is much trickier than it looks.
- We implement the idea in GHC
  - Accept full Haskell
  - Support separate compilation as the verification is modular.
  - Can check functions without contract through CEG unrolling.