Static Contract Checking for Haskell

Dana N. Xu

INRIA France

Work done at 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
```

From Types to Contracts

```
head (x:xs) = x

head :: [Int] -> Int
```

Type

... `(head 1)` ...

Bug!

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

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

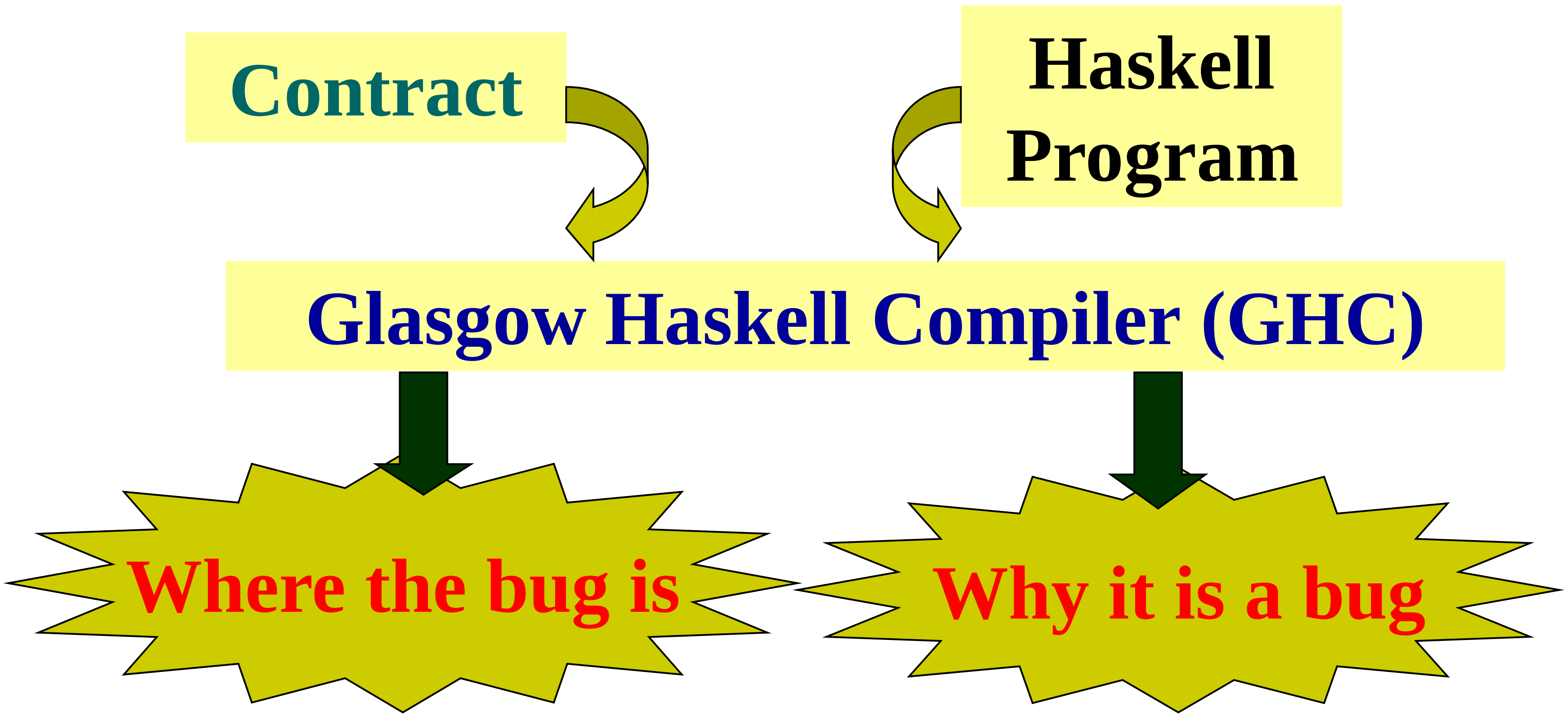
```
head ∈ {xs | not (null xs)} -> {r | True}
```

Contract
(original Haskell
boolean expression)

... `(head [])` ...

Bug!

What we want?


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

Satisfying a Predicate Contract

Arbitrary boolean-valued Haskell expression

$e \in \{x \mid p\}$ 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
       when (g 1) >= 0 holds
            (g 1) - 1 >= 0 does not hold
Error: f2 calls f1
       which fails f1's precondition
```


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

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 | True} -> {r | (n<=100 && r==91)
                         || r==n-10}
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 :: Int -> [Int] -> [Int]
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}
```

AVL Tree

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

```
balanced :: AVL -> Bool
balanced L = True
balanced (N  t u) = balanced t && balanced u &&
                     abs (depth t - depth u) <= 1

data AVL = L | N Int AVL AVL
insert, delete :: AVL -> Int -> AVL
insert ∈ {x | balanced x} -> {y | True} ->
          {r | notLeaf r && balanced r     &&
               0 <= depth r - depth x      &&
                    depth r - depth x <= 1   }

delete ∈ {x | balanced x} -> {y | True} ->
          {r | balanced r && 0 <= depth x - depth r &&
                                depth x - depth r <= 1}
```

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!

Lots of Questions

- What does "crash" mean?
- What is "a contract"?
- What does it mean to "satisfy a contract"?
- How can we verify that a function does satisfy a contract?
- What if the contract itself diverges? Or crashes?

It's time to get precise...

What is the Language?

- Programmer sees Haskell
- Translated (by GHC) into Core language
  - Lambda calculus
  - Plus algebraic data types, and case expressions
  - BAD and UNR are (exceptional) values
  - Standard reduction semantics $e_1 \rightarrow e_2$

$$
\begin{array}{lll}
a, e, p & ::= & n \mid v \mid \lambda(x :: \tau).e \mid e_1\ e_2 \mid K\ \overrightarrow{e} \\
 & \mid & \mathtt{case}\ e_0\ \mathtt{of}\ alt_1 \ldots alt_n \mid \mathtt{BAD} \mid \mathtt{UNR} \\
alt & ::= & pt \rightarrow e \\
pt & ::= & K\ \overrightarrow{(x :: \tau)} \mid \mathtt{DEFAULT}
\end{array}
$$

Two Exceptional Values

Real Haskell Program

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

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

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

```
div x y =
 case y == 0 of
 True -> error “divide by zero”
 False -> x / y

head (x:xs) = x
```

- **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)**

Crashing

Definition (Crash).

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

Definition (Crash-free Expression)

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

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

Non-termination is not a crash

(i.e. partial correctness).

Crash-free Examples

	Crash-free?
(1,BAD)	NO
(1, True)	YES
`\x -> x`	YES
`\x -> if x > 0 then x else (BAD, x)`	NO
`\x -> if x*x >= 0 then x + 1 else BAD`	Hmm.. YES

Lemma: For all closed *e*,
***e* is syntactically safe $\Rightarrow$ *e* is crash-free.**

What is a Contract

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

$\mathtt{t} \in$ **Contract**		
`t ::=`	`{x	p}`	**Predicate Contract**
`|`	$\mathtt{x}:\mathtt{t}_1 \to \mathtt{t}_2$	**Dependent Function Contract**
`|`	$(\mathtt{t}_1, \mathtt{t}_2)$	**Tuple Contract**
`|`	`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}`

Questions on e ∈ t

```
3 ∈ {x | x > 0}
5 ∈ {x | True}

(True, 2)      ∈ {x | (snd x) > 0}  ?
(head [], 3)   ∈ {x | (snd x) > 0} ?
BAD            ∈ ?
  ?            ∈ {x | False}
  ?            ∈ {x | BAD}
\x-> BAD       ∈ {x | False} -> {r | True} ?
\x-> BAD       ∈ {x | True} ->  ?
\x-> x         ∈ {x | True} ?
```

What exactly does it mean to say that

e “satisfies” contract t?

$$e \in t$$

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 ($e \rightarrow^* \lambda x.e'$ and $\forall e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x]$) **[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`

Only Crash-free Expression Satisfies a Predicate Contract

$e \in \{x \mid p\}$	$\Leftrightarrow$	$e\uparrow$ or (e is crash-free and $p[e/x] \not\rightarrow^{*} \{BAD, False\}$
$e \in x{:}t_1 \rightarrow t_2$	$\Leftrightarrow$	$e\uparrow$ or ($e \rightarrow^{*} \lambda x.e'$ and $\forall e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x]$)
$e \in (t_1, t_2)$	$\Leftrightarrow$	$e\uparrow$ or ($e \rightarrow^{*} (e_1,e_2)$ and $e_1 \in t_1$ and $e_2 \in t_2$)
$e \in$ `Any`	$\Leftrightarrow$	True

	YES or NO?
`(True, 2)` $\in$ `{x	(snd x) > 0}`	YES
`(head [], 3)` $\in$ `{x	(snd x) > 0}`	NO
`\x-> x` $\in$ `{x	True}`	YES
`\x-> x` $\in$ `{x	loop}`	YES
`5` $\in$ `{x	BAD}`	NO
`loop` $\in$ `{x	False}`	YES
`loop` $\in$ `{x	BAD}`	YES

All Expressions Satisfy `Any`

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

g x = fst (x, BAD)
```

Inlining may help, but breaks down when function definition is big or recursive

	YES or NO?
`5 ∈ Any`	YES
`BAD ∈ Any`	YES
`(head [], 3) ∈ (Any, {x | x> 0})`	YES
`\x -> x ∈ Any`	YES
`BAD ∈ Any -> Any`	NO
`BAD ∈ (Any, Any)`	NO

All Contracts are Inhabited

$e \in \{x \mid p\}$	$\Leftrightarrow$	$e\uparrow$ or (e is crash-free and $p[e/x] \not\to^* \{BAD, False\}$
$e \in x{:}t_1 \to t_2$	$\Leftrightarrow$	$e\uparrow$ or ($e \to^* \lambda x.e'$ and $\forall\, e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x]$)
$e \in (t_1, t_2)$	$\Leftrightarrow$	$e\uparrow$ or ($e \to^* (e_1,e_2)$ and $e_1 \in t_1$ and $e_2 \in t_2$)
$e \in$ `Any`	$\Leftrightarrow$	True

	YES or NO?
`\x-> BAD ∈ Any -> Any`	YES
`\x-> BAD ∈ {x	True} -> Any`	YES
`\x-> BAD ∈ {x	False} -> {r	True}`	NO

Blume&McAllester[JFP'06] say YES

What to Check?

Does function *f* satisfy 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}

(`main` is *crash-free*, hence the program cannot crash)

How to Check?

Part I

Define
$e \in t$

Grand Theorem
$e \in t \quad \Leftrightarrow \quad e \triangleright t$ is crash-free

(related to Blume&McAllester:JFP'06)

Construct
$e \triangleright t$
(e "ensures" t)

Part II

Simplify $(e \triangleright t)$

some 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"
```

Crash!

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

Crash!

Hence, three possible outcomes:

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

(2) Definite Bug (definitely crashes)

(3) Possible Bug

Wrappers $\triangleright$ and $\triangleleft$
($\triangleright$ pronounced ensures $\triangleleft$ pronounced requires)

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

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

$$e \triangleright (t_1,\; t_2) = \texttt{case } e \texttt{ of}$$
$$(e_1,\; e_2) \texttt{ -> } (e_1 \triangleright t_1,\; e_2 \triangleright t_2)$$

```
e ▷ Any = UNR
```

related to [Findler:ICFP02,Blume:JFP06,Hinze:FLOPS06]

Wrappers $\rhd$ and $\lhd$
($\rhd$ pronounced ensures $\lhd$ pronounced requires)

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

$e \lhd x{:}t_1 \rightarrow t_2$
$\quad = \lambda v.\ (e\ (v \rhd t_1)) \lhd t_2[v \rhd t_1/x]$

$e \lhd (t_1, t_2) = \mathtt{case}\ e\ \mathtt{of}$
$\qquad (e_1, e_2) \to (e_1 \lhd t_1, e_2 \lhd t_2)$

$e \lhd \mathtt{Any} = \mathtt{BAD}$

related to [Findler:ICFP02,Blume:JFP06,Hinze:FLOPS06]

Example

```
head:: [a] -> a
head []      = BAD
head (x:xs)  = x
```

head ∈ { xs | not (null xs) } -> Ok

head ▷{xs | not (null xs)} -> Ok

= \v. head (v ◁ {xs | not (null xs)}) ▷ Ok

e ▷ *Ok* = *e*

= \v. head (v ◁ {xs | not (null xs)})

```
= \v. head (case not (null v) of
                True -> v
                False -> UNR)
```

```
\v. head (case not (null v) of
               True -> v
               False -> UNR)
```

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

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

Now inline 'not' and 'null'

```
= \v. head (case v of
               [] -> UNR
               (p:ps) -> p)
```

Now inline 'head'

```
= \v. case v of
        []      -> UNR
        (p:ps)  -> p
```

So head [] fails with UNR, not BAD, blaming the caller

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
```

Grand Theorem
$e \in t \iff e \rhd t$ is crash-free

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

```
loop ∈{x | False}
loop ▷{x | False}
= case False of {True -> loop; False -> BAD}
= BAD,
```
which is not crash-free

```
BAD ∉ Ok -> Any
BAD ▷ Ok -> Any
= \v -> ((BAD (v ◁ Ok)) ▷ Any
= \v -> UNR,
```
which is crash-free

☹

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

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

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

```
loop ∈{x | False}
loop ▷{x | False}
= loop `seq` case False of {…}
= loop,
```
which is crash-free

```
BAD ∉ Ok -> Any
BAD ▷ Ok -> Any
= BAD `seq` \v -> ((BAD (v ◁ Ok)) ▷ Any
= BAD,
```
which is not crash-free

☺

Contracts that Diverge

`\x->BAD ∈ {x | loop} ?` NO

But

`\x->BAD ▷ {x | loop}`

```
= \x->BAD `seq` case loop of
                    True -> \x -> BAD
                    False -> BAD
```

crash-free

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

`fin` converts divergence to `True`

Contracts that Crash

Grand Theorem
$e \in t \quad \Leftrightarrow \quad e \triangleright t$ is crash-free

- … much trickier
  - ($\Rightarrow$) does not hold, ($\Leftarrow$) still holds
- Open Problem
  - Suppose `fin` converts `BAD` to `False`
  - Not sure if Grand Theorem holds because NO proof, and NO counter example either.

Well-formed Contracts

Grand Theorem

$e \in t \quad \Leftrightarrow \quad e \triangleright t$ is crash-free

Well-formed t

t is Well-formed (WF) iff

t = {x | p} and p is crash-free

or t = $x{:}t_1 \rightarrow t_2$ and t_1 is WF and $\forall e_1 \in t_1$, $t_2[e_1/x]$ is WF

or t = (t_1, t_2) and both t_1 and t_2 are WF

or t = Any

Properties of $\rhd$ and $\lhd$

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

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

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

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

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

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 $\preceq$]:

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

Lemma [Idempotence of Projection]:

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

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

Lemma [A Projection Pair]:

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

Lemma [A Closure Pair]:

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

How to Check?

Part I

Define
$e \in t$

Grand Theorem
$e \in t \iff e \triangleright t$ is crash-free

(related to Blume&McAllester:ICFP'04)

Construct
$e \triangleright t$
(e "ensures" t)

Part II

Simplify ($e \triangleright t$)

Normal form e'

If e' is syntactically safe,
then Done!

Simplification Rules

$$(\lambda x.e_1)\, e_2 \;\Longrightarrow\; e_1[e_2/x] \qquad \text{(BETA)}$$

$$(\mathsf{case}\ e_0\ \mathsf{of}\ \{K_i\ \vec{x_i} \to e_i\})\ a \;\Longrightarrow\; \mathsf{case}\ e_0\ \mathsf{of}\ \{K_i\ \vec{x_i} \to (e_i\ a)\} \quad fv(a) \cap \vec{x_i} = \emptyset \qquad \text{(CASEOUT)}$$

$$\mathsf{case}\ (\mathsf{case}\ e_0\ \mathsf{of}\ \{K_i\ \vec{x_i} \to e_i\})\ \mathsf{of}\ alts \;\Longrightarrow\; \mathsf{case}\ e_o\ \mathsf{of}\ \{K_i\ \vec{x_i} \to \mathsf{case}\ e_i\ \mathsf{of}\ alts\}\quad fv(alts) \cap \vec{x_i} = \emptyset \qquad \text{(CASECASE)}$$

$$\mathsf{case}\ K_j \vec{e_j}\ \mathsf{of}\ \{K_i\ \vec{x_i} \to e_i\} \;\Longrightarrow\; \mathtt{UNR} \quad \forall i. K_j \neq K_i \qquad \text{(NOMATCH)}$$

$$\mathsf{case}\ e_0\ \mathsf{of}\ \{K_i\ \vec{x_i} \to e_i; K_j\ \vec{x_j} \to \mathtt{UNR}\} \;\Longrightarrow\; \mathsf{case}\ e_0\ \mathsf{of}\ \{K_i\ \vec{x_i} \to e_i\} \qquad \text{(UNREACHABLE)}$$

$$\mathsf{case}\ e_0\ \mathsf{of}\ \{K_i\ \vec{x_i} \to e_i\} \;\Longrightarrow\; e_1 \quad \text{patterns are exhaustive and for all } i, fv(e_i) \cap \vec{x_i} = \emptyset \text{ and } e_1 = e_i \qquad \text{(SAMEBRANCH)}$$

$$\mathsf{case}\ e_0\ \mathsf{of}\ \{K_i\ \vec{x_i} \to e\} \;\Longrightarrow\; e_0 \quad e_0 \in \{\mathtt{BAD}\ lbl, \mathtt{UNR}\} \qquad \text{(STOP)}$$

$$\mathsf{case}\ K_i\ \vec{y_i}\ \mathsf{of}\ \{K_i\ \vec{x_i} \to e_i\} \;\Longrightarrow\; e_i[y_i/x_i] \qquad \text{(MATCH)}$$

$$\mathsf{case}\ e_0\ \mathsf{of}\ \{K_i\ \vec{x_i} \to \ldots \mathsf{case}\ e_0\ \mathsf{of}\{K_i\ \vec{x_i} \to e_i\} \ldots\} \;\Longrightarrow\; \mathsf{case}\ e_0\ \mathsf{of}\ \{K_i\ \vec{x_i} \to \ldots e_i \ldots\} \qquad \text{(SCRUT)}$$

Arithmetic
via External Theorem Prover

```
goo ▷ t_goo = \i ->
              case (i+8 > i) of
              False -> BAD "foo"
              True -> …
```

```
>>ThmProver
i+8>i
>>Valid!
```

```
>>ThmProver
push(i>j)
push(not (j<0))
(i>0)
>>Valid!
```

```
case i > j of
 True -> case j < 0 of
           False -> case i > 0 of
                      False -> BAD "f"
```

Counter-Example Guided Unrolling

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

After simplifying (sumT ▷ t_{sumT}) , we may have:

```
case (noT1 x) of
True -> case x of
        T1 a -> BAD
        T2 a -> a
        T3 t1 t2 -> case (noT1 t1) of
                    False -> BAD
                    True -> case (noT1 t2) of
                            False -> BAD
                            True -> sumT t1 + sumT t2
```

Step 1:
Program Slicing – Focus on the BAD Paths

```
case (noT1 x) of
True -> case x of
          T1 a -> BAD
          T3 t1 t2 -> case (noT1 t1) of
                        False -> BAD
                        True -> case (noT1 t2) of
                                  False -> BAD
```

Step 2: Unrolling

```
case (case x of
            T1 a -> False
            T2 a -> True
            T3 t1 t2 -> noT1 t1 && noT1 t2) of
True -> case x of
        T1 a -> BAD
        T3 t1 t2 -> case (noT1 t1) of
                    False -> BAD
                    True -> case (noT1 t2) of
                            False -> BAD
```

Counter-Example Guided Unrolling – The Algorithm

$$
\begin{array}{l}
\mathtt{escH}\ rhs\ 0 = \text{“Counter-example :”} +\!\!+\ \mathtt{report}\ rhs \\
\mathtt{escH}\ rhs\ n = \\
\mathtt{let}\ rhs' = \mathit{simplifier}\ rhs \\
\qquad b = \mathtt{noBAD}\ rhs' \\
\mathtt{in\ case}\ b\ \mathtt{of} \\
\quad \mathtt{True} \longrightarrow \text{“No\ \ Bug.”} \\
\quad \mathtt{False} \longrightarrow \mathtt{let}\ s = \mathtt{slice}\ rhs' \\
\qquad\qquad\qquad \mathtt{in\ case\ noFunCall}\ s\ \mathtt{of} \\
\qquad\qquad\qquad\quad \mathtt{True} \longrightarrow \mathtt{let}\ eg = \mathtt{oneEg}\ s \\
\qquad\qquad\qquad\qquad\qquad\quad \mathtt{in}\ \text{“Definite Bug :”} +\!\!+\ \mathtt{report}\ eg \\
\qquad\qquad\qquad\quad \mathtt{False} \longrightarrow \mathtt{let}\ s' = \mathtt{unrollCalls}\ s \\
\qquad\qquad\qquad\qquad\qquad\quad \mathtt{in\ escH}\ s'\ (n-1)
\end{array}
$$

Tracing

(Achieve the same goal as [Meunier, Findler, Felleisen:POPL06]

$g \in t_g$

$g = \ldots$

$f \in t_f$

$f = \ldots g \ldots$

$f \triangleright t_f = \ldots g \triangleleft t_g \ldots$

Inside "g" lc $(g \triangleleft t_g)$

```
case fin p[e/x] of
  True  -> e
  False -> BAD "f"
```

$(\backslash g \to \ldots g \ldots) \triangleright t_g \to t_f$

Counter-Example Generation

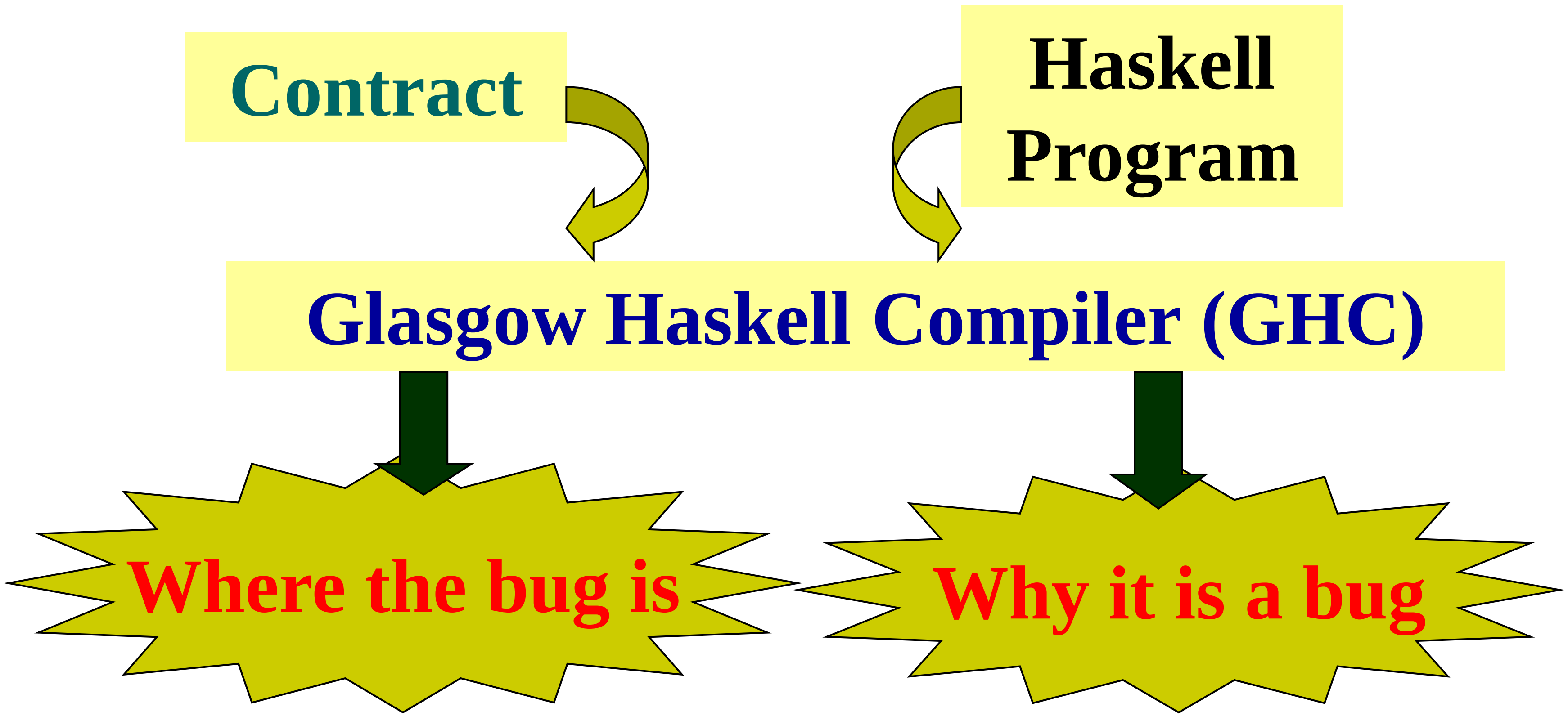
```
f1 ∈ x:Ok -> { x < z } -> Ok
f2 x z = 1 + f1 x z
```

```
f3 [] z = 0
f3 (x:xs) z = case x > z of
                True -> f2 x z
                False -> ...
```

```
f3 ▷ Ok = \xs -> \z ->
 case xs of
 [] -> 0
 (x:y) -> case x > z of
          True -> Inside "f2" <l2>
                       (Inside "f1" <l1> (BAD "f1"))
          False -> …
```

```
Warning <l3>: f3 (x:y) z where x>z
              calls f2
              which calls f1
              which may fail f1's precondition!
```

Conclusion



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

Summary

- Static contract checking is a fertile and under-researched area
- Distinctive features of our approach
  - Full Haskell in contracts; absolutely crucial
  - Declarative specification of "satisfies"
  - Nice theory (with some very tricky corners)
  - Static proofs
  - Modular Checking
  - Compiler as theorem prover

Contract Synonym

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

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

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

Actual Syntax

Recursion

```
f ▷t
=\f->f ▷ t->t
= …
=(… (f ◁ t)…) ▷ t
```

Suppose `t = t1 -> t_2`

```
f ▷ t1 -> t2
= \f->f ▷(t1 -> t2) -> (t1 -> t2)
= …
=(… (f ◁ t1 -> t2)…) ▷ t1 -> t2
=\v2.((…(\v1.((f (v1 ▷t1)) ◁ t2)) (v2◁ t1) …) ▷ t2)
```