STATIC CONTRACT CHECKING FOR HASKELL

Simon Peyton Jones
Microsoft Research

Joint work with
Dana Xu, University of Cambridge

Dec 2007

What we want

- The program should not crash
  - Type systems make a huge contribution
  - But no cigar

```
$ ./myprog
Error: head []
```

- The program should give "the right answer"
  - Too ambitious
  - More feasible: the program should have this property (QuickCheck)

```
prop_rev :: [a] -> Bool
prop_rev xs = length xs == length (reverse xs)
```

Major progress in OO languages

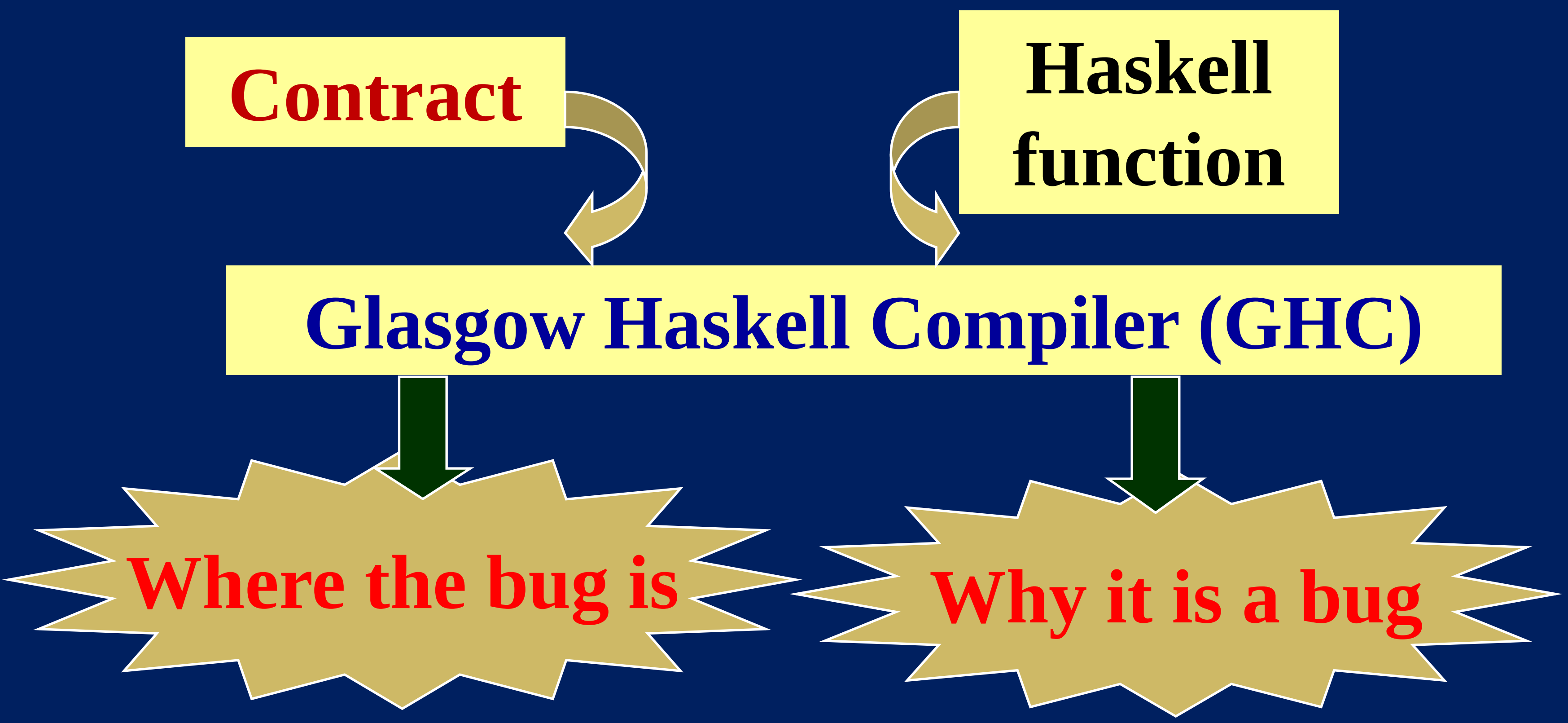
- ESC/Java, JML, Eiffel, Spec#, and others
- Main ideas:
  - Pre and post conditions on methods
  - Invariants on objects
  - Static (ESC, Spec#) or dynamic (Eiffel) checks
  - Heavy duty theorem provers for static checking
- Imperative setting makes it tough going

And in functional programming?

- "Functional programming is good because it's easier to reason about programs". But we don't actually **do** much reasoning.
- "Once you get it past the type checker, the program usually works". But not always!
- Massive opportunity: we start from a much higher base, so we should be able to do more.

What we want

- Contract checking for non-PhD programmers
- Make it like type checking



The contract idea [Findler/Felleisen]

- A generalisation of pre- and post-conditions to **higher order functions**

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

head ∈ {xs | not (null xs)} -> Ok
```

BAD means "Should not happen: crash"

head "satisfies" this contract

Ordinary Haskell

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

Contracts at higher order

```
f :: ([a] -> a) -> [a] -> a -> a
f g [] x = x
f g xs x = g xs

f ∈ ({xs | not (null xs)} -> Ok) -> Ok

...(f head)...
```

f only applies g to a non-empty list

f's contract allows a function that crashes on empty lists

This call is ok
f's contract explains why it is ok

Static and dynamic

[Flanaghan, Mitchell, Pottier, Regis-Giann]

Static checking

Program with contracts

Dynamic checking

[Findler, Felleisen, Blume, Hinze, Loh, Runciman, Chitil]

Compile time error attributes blame to the right place

Run time error attributes blame to the right place

```
f xs = head xs `max` 0

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

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

This call is ok

Static and dynamic

Program with contracts

Static checking

Compile time error attributes blame to the right place

No errors means Program cannot crash

Dynamic checking

Run time error attributes blame to the right place

Or, more plausibly:
If you guarantee that $f \in t$,
then the program cannot crash

Lots of questions

- What does “crash” mean?
- What is “a contract”?
- How expressive are contracts?
- 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...

Our goal

To statically detect crashes

- "Crash"
  - pattern-match failure
  - array-bounds check
  - divide by zero
- Non-termination is not a crash (i.e. partial correctness only)
- "Gives the right answer" is handled by writing properties

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 e1 -> e2

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

What is a contract?

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

3 ∈ { x | x>0 }

3 ∈ { x | True }

True ∈ { x | x }

(3, []) ∈ ({x | True}, {ys | null ys})

3 ∈ Any

True ∈ Any

(3, []) ∈ (Any, {ys | null ys})

(3, []) ∈ Any

The "p" in a predicate contract can be an arbitrary Haskell expression

Ok = { x | True}

What is a contract?

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

abs ∈ Ok -> { x | x >= 0 }

prop_rev ∈ Ok -> { x | x }

Guarantees to return True

sqrt ∈ x:{x | x>=0} -> {y | y*y == x}

Postcondition can mention argument

Precondition

Postcondition

Ok = { x | True}

Examples

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

sumT :: T -> Int
sumT ∈ {x | noT1 x} -> Ok
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
```

No case for T1

Examples

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

rmT1 :: T -> T
rmT1 ∈ Ok -> {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)

f :: T -> Int
rmT1 ∈ Ok -> Ok
f t = sumT (rmT1 t)
```

Removes T1

f does not crash despite calling sumT

Invariants on data structures

```
data Tree = Leaf | Node Int Tree Tree
Node ∈ t1:Ok -> {t2 | bal t1 t2} -> Ok

height :: Tree -> Int
height Leaf = 0
height (Node _ t1 t2) = height t1 `max` height t2

bal :: Tree -> Tree -> Bool
bal t1 t2 = abs (height t1 - height t2) <= 1
```

- Invariant is **required** when building a Node
- But is **available** when pattern-matching Node:

```
reflect (Node i t1 t2) = Node i t2 t1
```

What exactly does it mean to say that f “satisfies” a contract t?

$f \in t$

When does f "satisfy" a contract?

$$
\begin{array}{rcl}
e \in \{x \mid p\} & \Longleftrightarrow & e\uparrow \ \text{ or } (e \text{ is crash-free and } p[e/x] \not\to^{*} \{\texttt{BAD}, \texttt{False}\}) \\
e \in x{:}\, t_1 \to t_2 & \Longleftrightarrow & e\uparrow \ \text{ or } \forall e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x] \\
e \in (t_1, t_2) & \Longleftrightarrow & e\uparrow \ \text{ or } (e \to^{*} (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2) \\
e \in \texttt{Any} & \Longleftrightarrow & \texttt{True}
\end{array}
$$

- Brief, intuitive, declarative...

When does f "satisfy" a contract?

$$
\begin{array}{rcl}
e \in \{x \mid p\} & \Longleftrightarrow & e\!\uparrow \text{ or } (e \text{ is crash-free and } p[e/x] \not\rightarrow^* \{\texttt{BAD}, \texttt{False}\}) \\
e \in x{:}\, t_1 \rightarrow t_2 & \Longleftrightarrow & e\!\uparrow \text{ or } \forall e_1 \in t_1.\, (e\ e_1) \in t_2[e_1/x] \\
e \in (t_1, t_2) & \Longleftrightarrow & e\!\uparrow \text{ or } (e \rightarrow^* (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2) \\
e \in \texttt{Any} & \Longleftrightarrow & \texttt{True}
\end{array}
$$

- The delicate one is the predicate contract
- **Question1: what if e diverges?**
- **Our current decision:**
  if e diverges then e $\in$ {x:p}

When does f "satisfy" a contract?

$$\begin{array}{rcl}
e \in \{x \mid p\} & \Longleftrightarrow & e\uparrow \;\text{ or } (e \text{ is crash-free and } p[e/x] \not\to^* \{\texttt{BAD}, \texttt{False}\}) \\
e \in x{:}\, t_1 \to t_2 & \Longleftrightarrow & e\uparrow \;\text{ or } \forall e_1 \in t_1.\, (e\ e_1) \in t_2[e_1/x] \\
e \in (t_1, t_2) & \Longleftrightarrow & e\uparrow \;\text{ or } (e \to^* (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2) \\
e \in \texttt{Any} & \Longleftrightarrow & \texttt{True}
\end{array}$$

- The delicate one is the predicate contract
- **Question 2:** BADs in e:

BAD ∈ { x | True } ???

(BAD, 1) ∈ { x | snd x > 0} ???

head ∈ { x | True } ???

BADs in e

- Our decision:
  $e \in \{ x \mid p \} \Rightarrow$ e is **crash-free**
  regardless of p
- **e is crash-free** iff no blameless context can make e crash

e is crash-free
iff
$\forall C. \quad C[e] \rightarrow^* BAD \quad \Rightarrow \quad BAD \in C$

Crash free

	Crash free?
BAD	NO
(1, BAD)	NO
\x. BAD	NO
\x. case x of { [] -> BAD; (p:ps) -> p }	NO
(1,True)	YES
\x.x+1	YES
\x. if (x*x >= 0) then True else BAD	Umm.. YES

Conclusion: BAD∈e is not enough!
It is undecidable whether or not e is crash-free

When does f "satisfy" a contract?

$$
\begin{array}{rcl}
e \in \{x \mid p\} & \Longleftrightarrow & e\uparrow \;\text{ or } (e \text{ is crash-free and } p[e/x] \not\rightarrow^{*} \{\texttt{BAD}, \texttt{False}\}) \\
e \in x{:}\, t_1 \rightarrow t_2 & \Longleftrightarrow & e\uparrow \;\text{ or } \forall e_1 \in t_1.\, (e\ e_1) \in t_2[e_1/x] \\
e \in (t_1, t_2) & \Longleftrightarrow & e\uparrow \;\text{ or } (e \rightarrow^{*} (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2) \\
e \in \texttt{Any} & \Longleftrightarrow & \texttt{True}
\end{array}
$$

- Hence: e crash free ⇔ e ∈ Ok
- This is why we need Any; e.g.
  fst ∈ (Ok, Any) -> Ok

When does f "satisfy" a contract?

$$
\begin{array}{rcl}
e \in \{x \mid p\} & \Longleftrightarrow & e\uparrow \ \text{ or } (e \text{ is crash-free and } p[e/x] \not\to^* \{\texttt{BAD}, \texttt{False}\}) \\
e \in x{:}\, t_1 \to t_2 & \Longleftrightarrow & e\uparrow \ \text{ or } \forall e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x] \\
e \in (t_1, t_2) & \Longleftrightarrow & e\uparrow \ \text{ or } (e \to^* (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2) \\
e \in \texttt{Any} & \Longleftrightarrow & \texttt{True}
\end{array}
$$

- **Question 3:** what if p diverges?
  e.g. True ∈ { x | loop } ???
- Our decision: yes. (Same reason as for e.)

When does f "satisfy" a contract?

$$
\begin{array}{rcl}
e \in \{x \mid p\} & \Longleftrightarrow & e\uparrow \text{ or } (e \text{ is crash-free and } p[e/x] \not\to^{*} \{\texttt{BAD}, \texttt{False}\}) \\
e \in x{:}\, t_1 \to t_2 & \Longleftrightarrow & e\uparrow \text{ or } \forall e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x] \\
e \in (t_1, t_2) & \Longleftrightarrow & e\uparrow \text{ or } (e \to^{*} (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2) \\
e \in \texttt{Any} & \Longleftrightarrow & \texttt{True}
\end{array}
$$

- **Question 4:** what if p crashes?
  e.g. True ∊ { x | BAD} ???
- Our decision: no. Treat BAD like False.

Back to the big picture

- All of these choices are a matter of DEFINITION for what "satisfies" means.
- Ultimately what we want is:

  $$\text{main} \in \text{Ok}$$

  Hence main is crash-free; and hence the program cannot crash.
- In general, certainly undecidable, but hope for good approximation:
  - "definitely OK"
  - "definite crash"
  - "don't know but the tricky case is this"

How can we mechanically check that f satisfies t?

Checking $e \in t$

- The usual approach:
  - Extract verification conditions from the program
  - Feed to external theorem prover
  - "yes" means "function satisfies contract"
- Huge advantage: re-use mega-brain-power of the automatic theorem prover guys
- Disadvantage: must explain (enough of) language semantics to theorem prover
- Works best when there is a good match between language and prover (e.g. higher order, data structures, polymorphism...)

Our approach: exploit compiler

To prove $e \in t$

1. Form the term $(e \triangleright t)$
2. Use the compiler/optimiser to simplify the term: $(e \triangleright t) \Rightarrow e'$
3. See if BAD $\in e'$
4. If not, we know e' is crash free, and hence $(e \triangleright t)$ is crash free, and hence $e \in t$

Advantage: compiler already knows language semantics!

What is $(e \triangleright t)$?

- $(e \triangleright t)$ is e wrapped in dynamic checks for t (exactly a la Findler/Felleisen)
- Behaves just like e, except that it also checks for t

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

e ▷ x:t1 -> t2  =  \v. e (v ◁ t1) ▷ t2[e/x]

e ▷ Any =  UNR
```

What is $(e \triangleright t)$?

- $(e \triangleleft t)$ is dual,
  but fails with UNR instead of BAD
  and vice versa

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

e ◁ x:t1 -> t2  =  \v. e (v ▷ t1) ◁ t2[e/x]

e ◁ Any =  BAD
```

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)
```

Example

```
\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)  -> v)
```

Now inline 'head'

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

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

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

The big picture

Intuition: $e \triangleright t$

- crashes with BAD if e does not satisfy t
- crashes with UNR if context does not satisfy t

GrandTheorem

$e \in t \iff e \triangleright t$ is crash free

Optimise

e'

Check for BAD in here

Some interesting details

Theory

- Lots of Lovely Lemmas
- Contracts that loop
- Contracts that crash

Practice

- Using a theorem prover
- Finding counter-examples
- Counter-example guided inlining

Lovely lemmas (there are lots more)

Lemma [Monotonicity of Satisfaction]:
If $e_1 \in t$ and $e_1 \leq e_2$, then $e_2 \in t$

Lemma [Congruence of $\leq$]:
$e_1 \leq e_2 \quad \Rightarrow \quad \forall C.\ C[e_1] \leq C[e_2]$

Lemma [Idempotence of Projection]:
$\forall e,t. \quad e \triangleright t \triangleright t \equiv e \triangleright t$

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

Lemma [A Projection Pair]:
$\forall e,t. \quad e \triangleright t \triangleleft t \leq e$

Lemma [A Closure Pair]:
$\forall e,t. \quad e \leq e \triangleleft t \triangleright t$

Crashes more often
$e_1 \leq e_2$ iff
$\forall C.\ C[e_2] \rightarrow^* \text{BAD}$
$\Rightarrow C[e_1] \rightarrow^* \text{BAD}$

Contracts that loop

\x.BAD $\in$ {x | loop} ?

- **NO:** \x.BAD is not cf
- **BUT:** (\x.BAD) $\triangleright$ **{x | loop}**
  = case loop of ... = loop, which is c-f
- **Solution:**

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

  - **Reduction under fin_N decrements N**
  - **fin_0 p** → True
  - **Adjust Grand Theorem slightly**

Contracts that crash

- ...are much trickier
- Not sure if Grand Theorem holds... no proof, but no counter-example
- Weaken slightly:

If t is well-formed then
$e \in t \;\Leftrightarrow\; e \triangleright t$ is crash free

$\{x|p\}$ well-formed if p is crash free
$x{:}t_1 \to t_2$ well-formed if $\forall e \in t_1$, $t_2[e/x]$ well-formed

Practical aspects

Modular checking

```
f ∈ tf
f = ...

g ∈ tg
g = ...f...
```

Treat like:

```
g ∈ tf -> tg
g f = ...f...
```

When checking g's contract

- Replace f with (f $\triangleleft$ tf) in g's RHS
- That is, all g knows about f is what tf tells it

But this is not always what we want

Modular checking

```
null []     = True
null (x:xs) = False

g ∈ tg
g xs = if null xs then 1
       else head xs
```

- It would be a bit silly to write null's contract
  null ∈ `xs:Ok -> {n | n == null xs}`
- We just want to inline it!
- So if a function is not given a contract:
  - We try to prove the contract Ok
    - Success => inline it at need [null]
    - Failure => inline it unconditionally [head]

Counter-example guided inlining

- Suppose we compute that
  $e \triangleright t$ = \xs. not (null xs)

  Should we inline null, not?

  No: null, not can't crash (null$\in$Ok, not$\in$OK)
  And hence $e \triangleright t$ is crash-free

- But suppose
  $e \triangleright t$ = \xs. if not (null xs) then BAD else 1

  then we **should** inline null, not, in the hope of eliminating BAD

Counter-example guided inlining

General strategy to see if ($e \in t$):

Compute ($e \triangleright t$), and then

1. Perform symbolic evaluation, giving e_s
2. See if $BAD \in e_s$
3. If so, inline each function called in e_s, and go to (1)
4. Stop when tired

Using a theorem prover

```
f ∈ x:Ok -> {y | y>x} -> Ok

g ∈ Ok
g i = f i (i+8)
```

```
g ▷Ok = case (i+8 > i) of
               True -> f i (i+8)
               False -> BAD
```

Feed this theorem of arithmetic to an external theorem prover

Remember, we replace f by (f ◁ tf)

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
  - Compiler as theorem prover
- Lots to do
  - Demonstrate at scale
  - Investigate/improve what can and cannot be proved
  - Richer contract language (t1 & t2, t1 | t2, recursive contracts...)

What is (e ▷ t)?

Just e wrapped in dynamic checks for t (exactly a la Findler/Felleisen)

$$r \in \{\texttt{BAD}, \texttt{UNR}\}$$

$$\neg\texttt{BAD} = \texttt{UNR} \qquad \neg\texttt{UNR} = \texttt{BAD} \qquad e \triangleright t = e \overset{\texttt{BAD}}{\bowtie} t \qquad e \triangleleft t = e \overset{\texttt{UNR}}{\bowtie} t$$

$$\begin{array}{lll}
e \overset{r}{\bowtie} \{x \mid p\} & = & e \text{ `}\texttt{seq}\text{` } \texttt{case fin } p[e/x] \texttt{ of } \{\texttt{True} \rightarrow e;\ \texttt{False} \rightarrow r\} \\
e \overset{r}{\bowtie} x{:}\, t_1 \rightarrow t_2 & = & e \text{ `}\texttt{seq}\text{` } \lambda v.\ \texttt{let } \{x = (v \overset{\neg r}{\bowtie} t_1)\} \texttt{ in } (e\ x) \overset{r}{\bowtie} t_2 \\
e \overset{r}{\bowtie} (t_1, t_2) & = & \texttt{case } e \texttt{ of } \{(e_1, e_2) \rightarrow (e_1 \overset{r}{\bowtie} t_1,\ e_2 \overset{r}{\bowtie} t_2)\} \\
e \overset{r}{\bowtie} \texttt{Any} & = & \neg r
\end{array}$$