Extended Static Checking for Haskell (ESC/Haskell)

Dana N. Xu

University of Cambridge

advised by Simon Peyton Jones Microsoft Research, Cambridge

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"
```

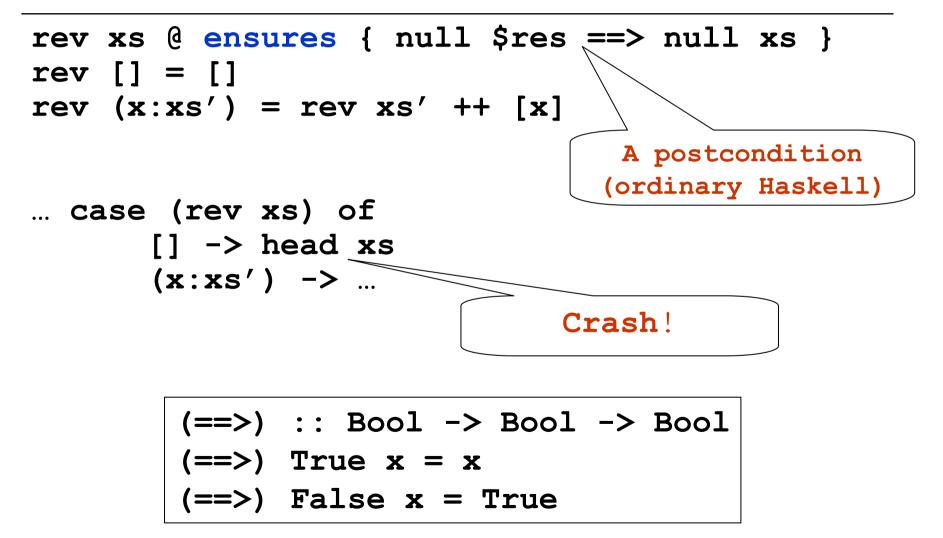
Glasgow Haskell Compiler (GHC) gives at run-time

Exception: Prelude.head: empty list

Preconditions

head xs @ requires { not (null xs) } head (x:xs') = xA precondition f xs = head xs `max` 0(ordinary Haskell) Warning: f [] calls head which may fail head's precondition! f ok xs = if null xs then 0 else head xs `max` 0 null :: [a] -> Bool not :: Bool -> Bool null [] = True not True = Falsenot False = True null (x:xs) = False

Postconditions



Expressiveness of the Specification Language

```
data T = T1 Bool | T2 Int | T3 T T
sumT :: T -> Int
sumT x @ requires { noT1 x }
sumT (T2 a) = a
sumT (T3 t1 t2) = sumT t1 + sumT t2
noT1 :: T \rightarrow 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 @ requires { noT1 x }
sumT (T2 a) = a
sumT (T3 t1 t2) = sumT t1 + sumT t2
rmT1 :: T -> T
rmT1 x @ ensures { noT1 $res }
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.

Functions without Annotations

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!

Higher Order Functions

```
all :: (a -> Bool) -> [a] -> Bool
all f [] = True
all f (x:xs) = f x && all f xs
filter p xs @ ensures { all p $res }
filter p [] = []
filter p (x:xs') = case (p x) of
True -> x : filter p xs'
False -> filter p xs'
```

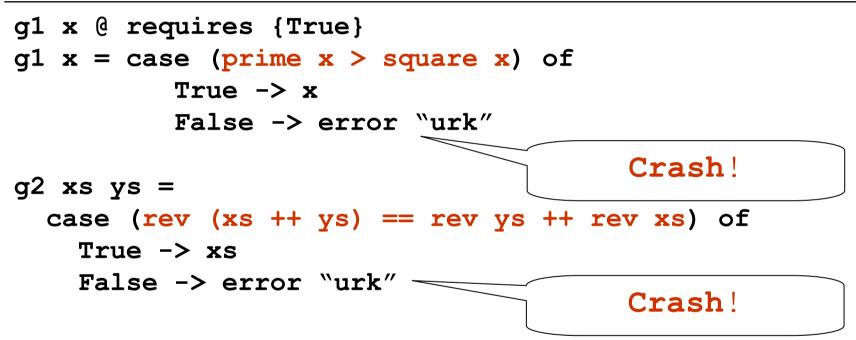
Various Examples

```
zip xs ys @ requires { sameLen xs ys}
zip xs ys @ ensures { sameLen $res xs }
sameLen [] [] = True
sameLen (x:xs) (y:ys) = sameLen xs ys
sameLen = False
```

Sorting

```
sorted [] = True
sorted (x:[]) = True
sorted (x:y:xs) = x \le y \&  sorted (y : xs)
insert i xs @ ensures { sorted xs ==> sorted $res }
insertsort xs @ ensures { sorted $res }
merge xs ys @ ensures { sorted xs & sorted ys
                         ==> sorted $res }
mergesort xs @ ensures { sorted $res }
bubbleHelper :: [Int] -> ([Int], Bool)
bubbleHelper xs @ ensures { not (snd $res) ==>
                           sorted (fst $res) }
bubblesort xs @ ensures { sorted $res }
```

What we can't do



Hence, three possible outcomes:

- (1) Definitely Safe (no crash, but may loop)
- (2) Definite Bug (definitely crashes)
- (3) Possible Bug

Language Syntax

following Haskell's lazy semantics

		def_1, \ldots, def_n	
	:=	Definition $f \vec{x} = e$ $f \vec{x} @ requires { e }$ $f \vec{x} @ ensures { e }$	
a,e	\in	Expression	
		BAD 161	A crash
		OK e	Safe expression
		UNR	Unreachable
		NoInline <i>e</i>	No inlining
		Inside <i>lbl loc e</i> λx.e	A call trace
	i	$e_1 e_2$	An application
	ĺ	case e_0 of $alts$	
		let $x=e_1$ in e_2	
		$C e_1 \dots e_n$	Constructor
		x	Variable
		n	Constant

Program

F

nam

 $alts ::= alt_1 \dots alt_n$ alt $::= p \rightarrow e$ Case alternative $::= C x_1 \dots x_n$ Pattern p

 \in Value val $val ::= n | C e_1 \dots e_n | \lambda x.e$

Preprocessing

1. Filling in missing pattern matchings.

2. Type checking the pre/postconditions.

```
head xs @ requires { xs /= [] }
head :: [a] -> a
head (x:xs) = x
```

Symbolic Pre/Post Checking

- At the definition of each function *f*, assuming the given precondition holds, we check
- 1. No pattern matching failure
- 2. **Precondition of all calls in the body of** *f* holds
- 3. **Postcondition holds for** *f* **itself.**

Given $f \vec{x} = e$, f.pre and f.post

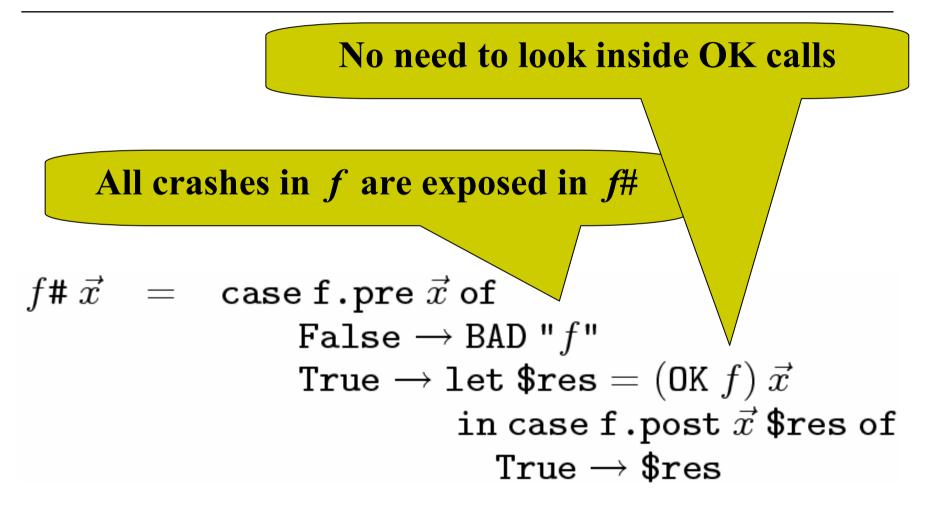
$$f_{ ext{Chk}} \, ec{x} = ext{case f.pre } ec{x} \, ext{of}$$

 $ext{True} o ext{let $res} = e[f_1 # / f_1, \dots, f_n # / f_n]$
 $ext{ in case f.post } ec{x} \, \res
 $ext{True} o \$res$
 $ext{False} o ext{BAD "} post$ "

Goal: show f_{chk} is crash-free!

Theorem: if so, then given precondition of *f* holds:
1. No pattern matching failure
2. Precondition of all calls in the body of *f* holds
3. Postcondition holds for *f* itself

The Representative Function



Simplifier

$\operatorname{let} x = r \operatorname{in} b$	\Rightarrow	b[r/x]	(Inline)	
--	---------------	--------	----------	--

$$(\lambda x.e_1) e_2 \implies e_1[e_2/x]$$
 (Beta)

$$(case \ e_0 \ of \ \{C_i \ \vec{x_i} \to e_i\}) \ a \implies case \ e_0 \ of \ \{C_i \ \vec{x_i} \to (e_i \ a)\} \ fv(a) \cap \vec{x_i} = \emptyset$$
(CASEOUT)

$$\begin{array}{ll} \operatorname{case}\left(\operatorname{case}\,e_{0}\,\operatorname{of}\,\left\{C_{i}\,\vec{x_{i}}\rightarrow e_{i}\right\}\right)\operatorname{of}\,alts \implies & \operatorname{case}\,e_{o}\,\operatorname{of}\,\left\{C_{i}\,\vec{x_{i}}\rightarrow\operatorname{case}\,e_{i}\,\operatorname{of}\,alts\right\}\\ & fv(alts)\cap\vec{x_{i}}=\emptyset \end{array} \tag{CASECASE}$$

$$case C_j \vec{e_j} \text{ of } \{C_i \ \vec{x_i} \to e_i\} \implies \text{UNR} \quad \forall i. C_j \neq C_i \tag{NOMATCH}$$

$$case e_0 \text{ of } \{C_i \ \vec{x_i} \to e_i; C_j \ \vec{x_j} \to \text{UNR}\} \implies case e_0 \text{ of } \{C_i \ \vec{x_i} \to e_i\}$$
(UNREACHABLE)

$$\begin{array}{rcl} \operatorname{case} e_0 \ \operatorname{of} \left\{ C_i \ \vec{x_i} \to e_i \right\} & \Longrightarrow & e_1 & \operatorname{patterns} \ \operatorname{are} \ \operatorname{exhaustive} \ \operatorname{and} \\ & e_0 \ \operatorname{is} \ \operatorname{crash-free} \ \operatorname{and} \\ & & \text{for all} \ i, fv(e_i) \cap \vec{x_i} = \emptyset \ \operatorname{and} \ e_1 = e_i \end{array} \tag{SAMEBRANCH}$$

$$case e_0 \text{ of } \{C_i \ \vec{x_i} \to e\} \implies e_0 \qquad e_0 \in \{BAD \ lbl, UNR\}$$
(STOP)

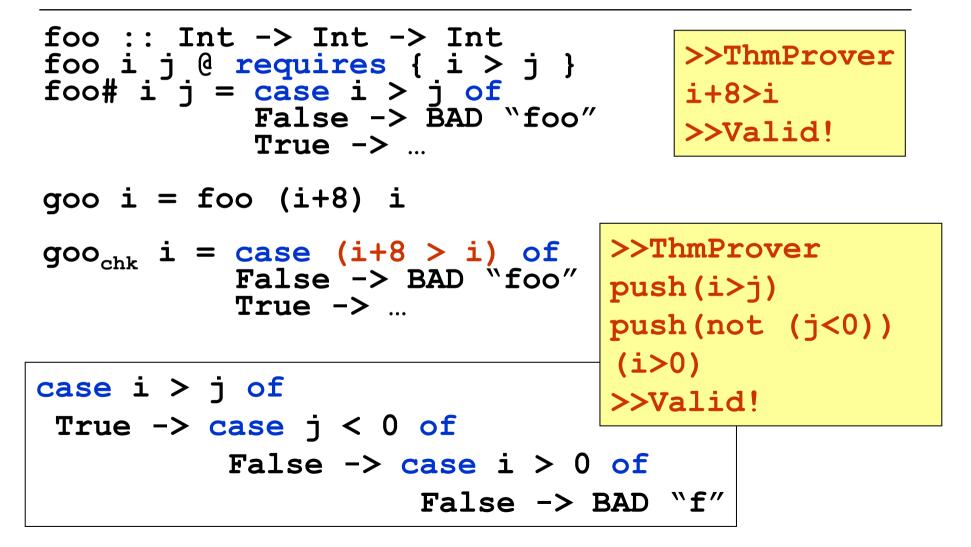
case
$$C_i \ \vec{y_i} \text{ of } \{C_i \ \vec{x_i} \to e_i\} \implies e_i[y_i/x_i]$$
 (MATCH)

 $case e_0 \text{ of } \{C_i \ \vec{x_i} \to \dots case e_0 \text{ of } \{C_i \ \vec{x_i} \to e_i\} \dots\} \implies case e_0 \text{ of } \{C_i \ \vec{x_i} \to \dots e_i \dots\}$ (SCRUT)

Expressive specification does not increase the complication of checking

```
filter f xs @ ensures { all f $res }
filter<sub>chk</sub> f xs =
case xs of
[] -> True
(x:xs') -> case (all f (filter f xs')) of
True -> ... all f (filter f xs') ...
```

Arithmetic via External Theorem Prover



Counter-Example Guided Unrolling

```
sumT :: T -> Int
sumT x @ requires { noT1 x }
sumT (T2 a) = a
sumT (T3 t1 t2) = sumT t1 + sumT t2
After simplifying sumT_{chk}, we may have:
 case ((OK noT1) x) of
 True \rightarrow case x of
           T1 a \rightarrow BAD "sumT"
           T2 a \rightarrow a
           T3 t1 t2 \rightarrow case ((OK noT1) t1) of
                         False -> BAD "sumT"
                         True -> case ((OK noT1) t2) of
                                  False -> BAD "sumT"
                                  True \rightarrow (OK sumT) t1 +
                                            (OK sumT) t2
```

Step 1: Program Slicing – Focus on the BAD Paths

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

Step 2: Unrolling

Keeping Known Information

```
case (case (NoInline ((OK noT1) x)) of
       True \rightarrow case x of
                 T1 a' \rightarrow False
                 T2 a' \rightarrow True
                 T3 t1' t2' -> (OK noT1) t1 &&
                                  (OK noT1) t2 )) of
 True \rightarrow case x of
           T1 a \rightarrow BAD "sumT"
           T3 t1 t2 -> case ((OK noT1) t1) of
                         False -> BAD "sumT"
                         True ->/case ((OK noT1) t2) of
case (NoInline ((OK noT1) t1)) of False -> BAD/"sumT"
   True -> ...
           case (NoInline ((OK noT1) t2)) of
               True -> ...
```

Counter-Example Guided Unrolling – The Algorithm

```
escH rhs 0 = "Counter-example :" ++ report rhs
escH rhs n =
let rhs' = simplifier rhs
    b = \text{noBAD} rhs'
in case b of
  True \rightarrow "No Bug."
  False \rightarrow let s = slice rhs'
            in case noFunCall s of
               True \rightarrow let eg = oneEg s
                         in "Definite Bug :" ++ report eg
               False \rightarrow let s' = unrollCalls s
                         in escH s' (n-1)
```

Tracing

Counter-Example Generation

f1 x z @ requires { x < z } f2 x z = 1 + f1 x z f2 x z = 1 + f1 x z f3 [] z = 0 f3 (x:xs) z = case x > z of

```
f3<sub>chk</sub> xs z = True -> f2 x z
f3<sub>chk</sub> xs z = False -> ...
case xs of
[] -> 0
(x:y) -> case x > z of
True -> Inside "f2" <12>
(Inside "f1" <11> (BAD "f1"))
False -> ...
```

```
Warning <13>: f3 (x:y) z where x>z
calls f2
which calls f1
which may fail f1's precondition!
```

Contributions

- Checks each program in a modular fashion on a per function basis. The checking is sound.
- Pre/postcondition annotations are in Haskell.
 - Allow recursive function and higher-order function
- □ Unlike VC generation, we treat pre/postcondition as booleanvalued functions and use symbolic simplification.
 - Handle user-defined data types
 - Better control of the verification process
- □ First time that Counter-Example Guided approach is applied to unrolling.
- □ Produce a trace of calls that may lead to crash at compile-time.
- Our prototype works on small but significant examples
 - Sorting: insertion-sort, merge-sort, bubble-sort
 - Nested recursion

Future Work

□ Allowing pre/post declaration for data types.

```
data A where
```

```
A1 :: Int -> A
```

```
A2 :: [Int] \rightarrow [Bool] \rightarrow A
```

```
A2 x y @ requires { length x == length y}
```

□ Allowing pre/post declaration for parameters in higherorder function.

```
map f xs @ requires {all f.pre xs}
map f [] = []
map f (x:xs') = f x : map f xs'
```

□ Allowing polymorphism and support full Haskell.