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') = x

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

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

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

Postconditions

```
rev xs @ ensures { null $res ==> null xs }
rev [] = []
rev (x:xs') = rev xs' ++ [x]
```

A postcondition (ordinary Haskell)

```
… case (rev xs) of
      [] -> head xs
      (x:xs') -> …
```

Crash!

```
(==>) :: Bool -> Bool -> Bool
(==>) True x = x
(==>) False x = True
```

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 -> 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
```

```
f91 n @ requires { n <= 101 }
f91 n @ ensures { $res == 91 }
f91 n = case (n <= 100) of
             True -> f91 (f91 (n + 11))
         False -> n - 10
```

Sorting

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

```
g1 x @ requires {True}
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

Language Syntax

following Haskell's lazy semantics

pgm	$\in$	**Program**	
pgm	$:=$	$def_1, \ldots, def_n$	
def	$\in$	**Definition**	
def	$:=$	$f\ \vec{x} = e$	
	$\mid$	$f\ \vec{x}$ @ `requires` $\{\ e\ \}$	
	$\mid$	$f\ \vec{x}$ @ `ensures` $\{\ e\ \}$	
a, e	$\in$	**Expression**	
a, e	$::=$	`BAD` lbl	A crash
	$\mid$	`OK` e	Safe expression
	$\mid$	`UNR`	Unreachable
	$\mid$	`NoInline` e	No inlining
	$\mid$	`Inside` $lbl\ loc\ e$	A call trace
	$\mid$	$\lambda x.e$	
	$\mid$	$e_1\ e_2$	An application
	$\mid$	`case` e_0 `of` $alts$	
	$\mid$	`let` x=e_1 `in` e_2	
	$\mid$	$C\ e_1 \ldots e_n$	Constructor
	$\mid$	x	Variable
	$\mid$	n	Constant
$alts$	$::=$	$alt_1 \ldots alt_n$	
alt	$::=$	$p \rightarrow e$	Case alternative
p	$::=$	$C\ x_1 \ldots x_n$	Pattern
val	$\in$	**Value**	
val	$::=$	$n \mid C\ e_1 \ldots e_n \mid \lambda x.e$	

Preprocessing

1. **Filling in missing pattern matchings.**

```
head (x:xs) = x
```

→

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

2. **Type checking the pre/postconditions.**

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

✗

```
head :: Eq a => [a] -> a
```

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

$$
\begin{array}{l}
f_{\text{Chk}}\ \vec{x} = \texttt{case f.pre}\ \vec{x}\ \texttt{of} \\
\qquad\qquad \texttt{True} \longrightarrow \texttt{let \$res} = e[f_1\#/f_1, \ldots, f_n\#/f_n] \\
\qquad\qquad\qquad\qquad \texttt{in case f.post}\ \vec{x}\ \texttt{\$res} \\
\qquad\qquad\qquad\qquad\quad \texttt{True} \longrightarrow \texttt{\$res} \\
\qquad\qquad\qquad\qquad\quad \texttt{False} \longrightarrow \texttt{BAD "}\mathit{post}\texttt{"}
\end{array}
$$

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

No need to look inside OK calls

All crashes in *f* are exposed in *f#*

$$
\begin{aligned}
f\#\ \vec{x} \quad = \quad & \texttt{case f.pre}\ \vec{x}\ \texttt{of} \\
& \quad \texttt{False} \longrightarrow \texttt{BAD "}f\texttt{"} \\
& \quad \texttt{True} \longrightarrow \texttt{let \$res} = (\texttt{OK}\ f)\ \vec{x} \\
& \qquad\qquad \texttt{in case f.post}\ \vec{x}\ \texttt{\$res of} \\
& \qquad\qquad\quad \texttt{True} \longrightarrow \texttt{\$res}
\end{aligned}
$$

Simplifier

$\text{let } x = r \text{ in } b$	$\Longrightarrow$	$b[r/x]$		(INLINE)
$(\lambda x.e_1)\ e_2$	$\Longrightarrow$	$e_1[e_2/x]$		(BETA)
$(\mathsf{case}\ e_0\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow e_i\})\ a$	$\Longrightarrow$	$\mathsf{case}\ e_0\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow (e_i\ a)\}$	$fv(a) \cap \vec{x_i} = \emptyset$	(CASEOUT)
$\mathsf{case}\ (\mathsf{case}\ e_0\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow e_i\})\ \mathsf{of}\ alts$	$\Longrightarrow$	$\mathsf{case}\ e_o\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow \mathsf{case}\ e_i\ \mathsf{of}\ alts\}$	$fv(alts) \cap \vec{x_i} = \emptyset$	(CASECASE)
$\mathsf{case}\ C_j \vec{e_j}\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow e_i\}$	$\Longrightarrow$	$\mathtt{UNR}$	$\forall i. C_j \neq C_i$	(NOMATCH)
$\mathsf{case}\ e_0\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow e_i; C_j\ \vec{x_j} \rightarrow \mathtt{UNR}\}$	$\Longrightarrow$	$\mathsf{case}\ e_0\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow e_i\}$		(UNREACHABLE)
$\mathsf{case}\ e_0\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow e_i\}$	$\Longrightarrow$	e_1	patterns are exhaustive and e_0 is crash-free and for all $i, fv(e_i) \cap \vec{x_i} = \emptyset$ and $e_1 = e_i$	(SAMEBRANCH)
$\mathsf{case}\ e_0\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow e\}$	$\Longrightarrow$	e_0	$e_0 \in \{\mathtt{BAD}\ lbl, \mathtt{UNR}\}$	(STOP)
$\mathsf{case}\ C_i\ \vec{y_i}\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow e_i\}$	$\Longrightarrow$	$e_i[y_i/x_i]$		(MATCH)
$\mathsf{case}\ e_0\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow \ldots \mathsf{case}\ e_0\ \mathsf{of}\{C_i\ \vec{x_i} \rightarrow e_i\} \ldots\}$	$\Longrightarrow$	$\mathsf{case}\ e_0\ \mathsf{of}\ \{C_i\ \vec{x_i} \rightarrow \ldots e_i \ldots\}$		(SCRUT)

Expressive specification does not increase the complication of checking

```
filter f xs @ ensures { all f $res }

filter_chk 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

```
foo :: Int -> Int -> Int
foo i j @ requires { i > j }
foo# i j = case i > j of
             False -> BAD "foo"
             True -> …
```

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

```
goo i = foo (i+8) i
```

goo_{chk}
```
goo_chk i = case (i+8 > i) of
             False -> BAD "foo"
             True -> …
```

```
>>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 @ requires { noT1 x }
sumT (T2 a) = a
sumT (T3 t1 t2) = sumT t1 + sumT t2
```

After simplifying $\texttt{sumT}_{\texttt{chk}}$, we may have:

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

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

Keeping Known Information

```
case (case (NoInline ((OK noT1) x)) of
      True -> case x of
                T1 a'  -> False
                T2 a'  -> True
                T3 t1' t2' -> (OK noT1) t1  &&
                              (OK noT1) t2 )) 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"
```

```
case (NoInline ((OK noT1) t1)) of
    True -> ...
```

```
case (NoInline ((OK noT1) t2)) of
    True -> ...
```

Counter-Example Guided Unrolling – The Algorithm

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

Tracing

$$
\begin{array}{lcl}
f\#\ \vec{x} & = & \texttt{case f.pre}\ \vec{x}\ \texttt{of} \\
 & & \quad \texttt{False} \longrightarrow \texttt{BAD "}f\texttt{"} \\
 & & \quad \texttt{True} \longrightarrow \texttt{let \$res} = (\texttt{OK}\ f)\ \vec{x} \\
 & & \qquad \texttt{in case f.post}\ \vec{x}\ \texttt{\$res of} \\
 & & \qquad\quad \texttt{True} \longrightarrow \texttt{\$res}
\end{array}
$$

$$
\begin{array}{lcl}
f\#\ \vec{x} & = & \texttt{Inside "}f\texttt{"}\ loc \\
 & & \quad (\texttt{case f.pre}\ \vec{x}\ \texttt{of} \\
 & & \quad \texttt{False} \longrightarrow \texttt{BAD "}f\texttt{"} \\
 & & \quad \texttt{True} \longrightarrow \texttt{let \$res} = (\texttt{OK}\ f)\ \vec{x} \\
 & & \qquad \texttt{in case f.post}\ \vec{x}\ \texttt{\$res of} \\
 & & \qquad\quad \texttt{True} \longrightarrow \texttt{\$res})
\end{array}
$$

Counter-Example Generation

```
f1 x z @ requires { x < z }
f2 x z = 1 + f1 x z
```

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

```
f3chk 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!
```

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 boolean-valued 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] -> [Bool] -> A
    A2 x y @ requires { length x == length y}
```

- Allowing pre/post declaration for parameters in higher-order 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.