Static Contract Checking for Haskell

Dana N. Xu
University of Cambridge

Ph.D. Supervisor:
Simon Peyton Jones
Microsoft Research Cambridge
From Types to Contracts

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

head :: [a] -> a

...(head 1)... 

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

...(head [])... 

BAD means “should not happen: crash”

Type

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

Bug!

Contract
(arbitrary Haskell boolean expression)
What we want

• Adapt Findler-Felleisen’s ideas for dynamic (high-order) contract checking.
• Do static contract checking for a lazy language.

Contract

Haskell function

Glasgow Haskell Compiler (GHC)

Where the bug is

Why it is a bug
Three Outcomes

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

(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}

bubbleHelper -> {r | not (snd r) ==> sorted (fst r)}

insertsort, mergesort, bubblesort ∈ {xs | True}

insertsort, mergesort, bubblesort -> {r | sorted r}
AVL Tree

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}
The Contract Idea for Higher-Order Function 
[Findler/Felleisen]

\[ f_1 :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \]
\[ f_1 \in (\{x \mid x > 0\} \rightarrow \{y \mid y \geq 0\}) \rightarrow \{r \mid r \geq 0\} \]
\[ f_1 \ g = (g \ 0) - 1 \]

Blame \( f_1 \): \( f_1 \) calls \( g \) with wrong argument

\[ f_2 :: \{r \mid \text{True}\} \]
\[ f_2 = f_1 \ (\lambda x \rightarrow x - 5) \]

\( f_1 \) does not satisfy its post-condition

\[ f_3 :: \{r \mid \text{True}\} \]
\[ f_3 = f_1 \ (\lambda x \rightarrow x - 1) \]

\( f_3 \) is Ok.

Can't tell at run-time

Blame \( f_2 \): \( f_2 \) calls \( f_1 \) with wrong argument
What is a Contract?

| Contract | t ::= {x | p} | Predicate Contract |
|----------|--------------|---------------------|
|          | x:t₁ → t₂    | Dependent Function Contract |
|          | (t₁, t₂)    | Tuple Contract |
|          | Any          | Polymorphic Any Contract |

Ok = \{x | True\}

| 3 ∈ \{x | x > 0\} | (3, []) ∈ Any |
| 3 ∈ \{x | True\} | (3, []) ∈ (Ok, {ys | null ys}) |
| inc ∈ x:{x | x>0} → {y | y == x + 1} |

Precondition | Postcondition | Postcondition can mention argument
What we want?

Check \( f \in <\text{contract\_of\_f}> \)

- If \( \text{main} \in \text{Ok} \), then the whole program cannot crash.
- If not, show which function to blame and why.
Define $e \in t$

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

(related to Blume & McAllester: JFP’06)

Construct
$e \triangleright t$
(e “ensures” t)

[POPL’10]

ESC/Haskell
[Haskell’06]

Symbolically simplify
$(e \triangleright t)$

See if BAD is syntactically in $e'$. If yes, DONE; else give BLAME

some $e'$
Wrappers \(\triangleright\) and \(\triangleleft\)

\(\triangleright\) pronounced ensures \(\triangleleft\) pronounced requires

\[e \triangleright \{x \mid p\} = \text{case } p[e/x] \text{ of}\]
\[\quad \text{True } \rightarrow e\]
\[\quad \text{False } \rightarrow \text{BAD}\]

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

\[e \triangleright (t_1, t_2) = \text{case } e \text{ of}\]
\[\quad (e_1, e_2) \rightarrow (e_1 \triangleright t_1, e_2 \triangleright t_2)\]

\[e \triangleright \text{Any } = \text{UNR}\]

related to [Findler-Felleisen:ICFP02]
Wrappers \(\triangleright\) and \(\triangleleft\)

(\(\triangleright\) pronounced \textit{ensures} \quad \triangleleft\) pronounced \textit{requires})

\[
e \triangleleft \{x \mid p\} = \text{case } p[e/x] \text{ of}
\]

\[
\begin{align*}
\text{True} & \rightarrow e \\
\text{False} & \rightarrow \text{UNR}
\end{align*}
\]

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

\[
e \triangleleft (t_1, t_2) = \text{case e of}
\]

\[
(e_1, e_2) \rightarrow (e_1 \triangleleft t_1, e_2 \triangleleft t_2)
\]

\[
e \triangleleft \text{Any} = \text{BAD}
\]

related to [Findler-Felleisen:ICFP02]
Some Interesting Details

Theory

- Contracts that loop
- Contracts that crash
- Lovely Lemmas

Practice

- Adding tags, e.g. BAD “f”
  - Achieves precise blaming
- More tags to trace functions to blame
  - Achieves the same goal of [Meunier:POPL06]
- Using a theorem prover
- Counter-example guided unrolling
## Lovely Lemmas 😊

<table>
<thead>
<tr>
<th>Lemma</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Congruence</td>
<td>$\forall e_1, e_2. e_1 \leq e_2 \iff \forall C. C[e_1] \leq C[e_2]$</td>
</tr>
<tr>
<td>Conditional Projection (w.r.t. $\leq, \succeq$)</td>
<td>For all $e$ and crash-free $t$, if $e \in t$, then (a) $e \triangleleft t \leq e$; (b) $e \triangleright t \geq e$.</td>
</tr>
<tr>
<td>Key Lemma</td>
<td>For all crash-free $e$, crash-free $t$, $e \triangleleft t \in t$.</td>
</tr>
<tr>
<td>Monotonicity of $\in$</td>
<td>If $e_1 \in t$ and $e_1 \leq e_2$, then $e_2 \in t$</td>
</tr>
</tbody>
</table>
| Idempotence                               | $\forall e, t. (a) (e \triangleright t) \triangleright t \equiv e \triangleright t$  
                                           | (b) $(e \triangleleft t) \triangleleft t \equiv e \triangleleft t$        |
| Projection Pair                           | $\forall e, t. (e \triangleright t) \triangleleft t \leq e$             |
| Closure Pair                              | $\forall e, t. e \leq (e \triangleleft t) \triangleright t$             |
| Telescoping Property                      | For all $e$, crash-free $t$. $(e \overset{r_1}{\underset{r_2}{\bowtie}} t) \overset{r_3}{\underset{r_4}{\bowtie}} t = e \overset{r_1}{\underset{r_4}{\bowtie}} t$ |


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
After Ph.D.

• Postdoc project in 2009: probabilistic contract for component base design [ATVA’2010]
• Current project at Xavier Leroy’s team (INRIA)
  - a verifying compiler:
    1. Apply the idea to OCaml compiler by allowing both static and dynamic contract checking
    2. Connect with Coq to verify more programs statically.
    3. Use Coq to prove the correctness of the framework.
    4. Apply new ideas back to Haskell (e.g. GHC).
Static and Dynamic

Program with Specifications

Compile time error attributes blame to the right place

Dynamic checking

Run time error attributes blame to the right place

Static checking

No blaming means Program cannot crash

Or, more plausibly:
If you guarantee that $f \in t$, then the program cannot crash
What exactly does it mean to say that

\[ e \text{ "satisfies" contract } t \]
When does $e$ satisfy a contract?

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

- **Brief, declarative...**

$$\text{inc} \in x : \{x \mid x > 0\} \rightarrow \{y \mid y = x + 1\}$$

**Precondition**  **Postcondition**  **Postcondition can mention argument**
When does \( e \) satisfy a contract?

\[
e \in \{x \mid p\} \iff e \uparrow \quad \text{or} \quad (e \text{ is crash-free and } p[e/x] \not\twoheadrightarrow \{\text{BAD, False}\})
\]

\[
e \in x : t_1 \rightarrow t_2 \iff e \uparrow \quad \text{or} \quad (e \rightarrow^* \lambda x.e_2 \text{ and } \forall e_1 \in t_1. (e \ e_1) \in t_2[e_1/x])
\]

\[
e \in (t_1, t_2) \iff e \uparrow \quad \text{or} \quad (e \rightarrow^* (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2)
\]

\[
e \in \text{Any} \iff \text{True}
\]

- The delicate one is the predicate contract.
- Our decision:

\[
e \in \{x \mid p\} \iff e \text{ is crash-free}
\]

- Question: What expression is crash-free?
  
  e is crash-free iff no blameless context can make e crash

\[
e \text{ is crash-free iff } \forall C. \text{BAD} \notin_s C. \ C[e] \not\twoheadrightarrow^* \text{BAD}
\]
Crash-free Examples

<table>
<thead>
<tr>
<th>Crash-free?</th>
<th>( \text{x} \rightarrow \text{x} )</th>
<th>YES</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>(1, True)</td>
<td>YES</td>
</tr>
<tr>
<td>NO</td>
<td>(1, BAD)</td>
<td>NO</td>
</tr>
<tr>
<td>NO</td>
<td>( \text{x} \rightarrow \text{if } \text{x} &gt; 0 \text{ then } \text{x} \text{ else } (\text{BAD}, \text{x}) )</td>
<td>NO</td>
</tr>
<tr>
<td>Hmm.. YES</td>
<td>( \text{x} \rightarrow \text{if } \text{x} \times \text{x} \geq 0 \text{ then } \text{x} + 1 \text{ else } \text{BAD} )</td>
<td></td>
</tr>
</tbody>
</table>

Lemma:

\( \text{e is syntactically safe} \implies \text{e is crash-free.} \)
When does $e$ satisfy a contract?

$$e \in \{x \mid p\} \iff e \uparrow \text{ or } (e \text{ is crash-free and } p[e/x] \not\Rightarrow^* \{\text{BAD, False}\})$$

$$e \in x : t_1 \to t_2 \iff e \uparrow \text{ or } (e \Rightarrow^* \lambda x.e_2 \text{ and } \forall e_1 \in t_1. (e.e_1) \in t_2[e_1/x])$$

$$e \in (t_1, t_2) \iff e \uparrow \text{ or } (e \Rightarrow^* (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2)$$

$$e \in \text{Any} \iff \text{True}$$

See the paper for ...

• Why $e$ must be crash-free to satisfy predicate contract?
• Why divergent expression satisfies all contract?
• What if contract diverges (i.e. $p$ diverges)?
• What if contract crashes (i.e. $p$ crashes)?
How can we mechanically check that

\[ e \in t \quad ??? \]
Example

\begin{aligned}
\text{head} :: [a] & \to a \\
\text{head} [\,] &= \text{BAD} \\
\text{head} (x:xs) &= x
\end{aligned}

\text{head} \in \{ \, xs \mid \text{not (null xs)} \, \} \to \text{Ok}

\begin{aligned}
\text{head} \triangleright \{ xs \mid \text{not (null xs)} \} & \to \text{Ok} \\
&= \forall v. \text{head} (v \triangleleft \{ xs \mid \text{not (null xs)} \}) \triangleright \text{Ok}
\end{aligned}

\begin{aligned}
\text{e} \triangleright \text{Ok} &= e \\
&= \forall v. \text{head} (v \triangleleft \{ xs \mid \text{not (null xs)} \})
\end{aligned}

\begin{aligned}
&= \forall v. \text{head} (\text{case \text{not (null v)}} \text{of} \text{ True} \to v \text{ False} \to \text{UNR})
\end{aligned}
\(v. \text{head (case not (null } v) \text{ of True } \rightarrow v\)
\(\text{False } \rightarrow \text{UNR})\)

Now inline ‘not’ and ‘null’

\(= \backslash v. \text{head (case } v \text{ of} \)
\(\text{[ ] } \rightarrow \text{UNR}\)
\((p:ps) \rightarrow p)\)

Now inline ‘head’

\(= \backslash v. \text{case } v \text{ of}\)
\(\text{[ ] } \rightarrow \text{UNR}\)
\((p:ps) \rightarrow p)\)

\(\text{null :: } [a] \rightarrow \text{Bool}\)
\(\text{null } [] = \text{True}\)
\(\text{null } (x:xs) = \text{False}\)

\(\text{not :: } \text{Bool} \rightarrow \text{Bool}\)
\(\text{not } \text{True} = \text{False}\)
\(\text{not } \text{False} = \text{True}\)

\(\text{head :: } [a] \rightarrow a\)
\(\text{head } [] = \text{BAD}\)
\(\text{head } (x:xs) = x\)

So head [] fails with UNR, not BAD, blaming the caller.
Static and Dynamic Program with Specifications

Compile time error attributes blame to the right place

Dynamic checking

Static checking

Run time error attributes blame to the right place

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

[Flanagan, Mitchell, Pottier]