STATIC CONTRACT CHECKING FOR HASKELL

Simon Peyton Jones
Microsoft Research

Joint work with
Dana Xu, University of Cambridge

Dec 2007
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.
Contract checking for non-PhD programmers

Make it like type checking

What we want

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

Glasgow Haskell Compiler (GHC)

Where the bug is

Why it is a bug
The contract idea [Findler/Felleisen]

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

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

\[
\text{head} \in \{xs \mid \text{not (null xs)}\} \rightarrow \text{Ok}
\]

BAD means “Should not happen: crash”

head “satisfies” this contract

Ordinary Haskell

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

\[ f : : ([a] \rightarrow a) \rightarrow [a] \rightarrow a \rightarrow a \]
\[ f \ g \ [\ ] \ x = x \]
\[ f \ g \ x s \ x = g \ x s \]
\[ f \in \{xs \mid \text{not (null xs)}\} \rightarrow \text{Ok} \rightarrow \text{Ok} \]
\[ \ldots(f \ \text{head}) \ldots \]

- \( f \)'s contract allows a function that crashes on empty lists
- This call is ok \( f \)'s contract explains why it is ok
- \( f \) only applies \( g \) to a non-empty list
Program with contracts

**Static checking**

Compile time error attributes blame to the right place

**Dynamic checking**

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
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 $e_1 \rightarrow e_2$

\[
\begin{align*}
a, e, p & ::= n \mid v \mid \lambda(x :: \tau).e \mid e_1 e_2 \mid K \overrightarrow{e} \\
& \quad \mid \text{case } e_0 \text{ of } alt_1 \ldots alt_n \mid \text{BAD} \mid \text{UNR} \\
alt & ::= pt \rightarrow e \\
pt & ::= K(x :: \tau) \mid \text{DEFAULT}
\end{align*}
\]
What is a contract?

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

3 ∈ {x | x>0}
3 ∈ {x | True}
True ∈ {x | x}

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

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

Ok = {x | True}
What is a contract?

| Contract | \( t ::= \{ x | p \} \) | Predicate Contract |
|----------|----------------|-------------------|
|          | \( x : t_1 \rightarrow t_2 \) | Dependent Function Contract |
|          | \( (t_1,t_2) \) | Tuple Contract |
|          | \( \text{Any} \) | Polymorphic Any Contract |

\( \text{abs} \in \text{Ok} \rightarrow \{ x | x \geq 0 \} \)

\( \text{prop_rev} \in \text{Ok} \rightarrow \{ x | x \} \)

\( \text{sqrt} \in x:{x | x \geq 0} \rightarrow \{ y | y^*y = x \} \)

Precondition

Postcondition

\( \text{Ok} = \{ x | \text{True} \} \)

Guarantees to return True

Postcondition can mention argument
data \( T = T_1 \) \( \text{Bool} \) | \( T_2 \) \( \text{Int} \) | \( T_3 \) \( T \) \( T \)

\[
\text{sumT} :: T \rightarrow \text{Int}
\]

\[
\text{sumT} \in \{ x \mid \text{noT1} \ x \} \rightarrow \text{Ok}
\]

\[
\text{sumT} (T_2 \ a) = a
\]

\[
\text{sumT} (T_3 \ t_1 \ t_2) = \text{sumT} \ t_1 + \text{sumT} \ t_2
\]

\[
\text{noT1} :: T \rightarrow \text{Bool}
\]

\[
\text{noT1} (T_1 \ _) = \text{False}
\]

\[
\text{noT1} (T_2 \ _) = \text{True}
\]

\[
\text{noT1} (T_3 \ t_1 \ t_2) = \text{noT1} \ t_1 \&\& \text{noT1} \ t_2
\]

No case for \( T_1 \)
Examples

\[
\text{sumT} :: T \rightarrow \text{Int}
\]
\[
\text{sumT} \in \{x \mid \text{noT1 } x\} \rightarrow \text{Ok}
\]
\[
\text{sumT} (T2 \ a) = a
\]
\[
\text{sumT} (T3 \ t1 \ t2) = \text{sumT} \ t1 + \text{sumT} \ t2
\]

\[
\text{rmT1} :: T \rightarrow T
\]
\[
\text{rmT1} \in \text{Ok} \rightarrow \{r \mid \text{noT1 } r\}
\]
\[
\text{rmT1} (T1 \ a) = \text{if } a \text{ then } T2 \ 1 \text{ else } T2 \ 0
\]
\[
\text{rmT1} (T2 \ a) = T2 \ a
\]
\[
\text{rmT1} (T3 \ t1 \ t2) = T3 (\text{rmT1} \ t1) (\text{rmT1} \ t2)
\]

\[
\text{f} :: T \rightarrow \text{Int}
\]
\[
\text{rmT1} \in \text{Ok} \rightarrow \text{Ok}
\]
\[
f \ t = \text{sumT} (\text{rmT1} \ t)
\]
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 $f \text{ “satisfies” a contract } t$?

$f \in t$
When does $f$ “satisfy” a contract?

\[
\begin{align*}
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 \rightarrow t_2 & \iff 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) & \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}
\end{align*}
\]

- Brief, intuitive, declarative...
When does f “satisfy” a contract?

- The delicate one is the predicate contract
- **Question 1**: what if e diverges?
- **Our current decision**: if e diverges then e ∈ {x:p}
When does f “satisfy” a contract?

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

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

\[ e \in \{ x \mid p \} \implies e \text{ is crash-free} \]
regardless of \( p \)

\[ e \text{ is crash-free} \iff \text{no blameless context can make } e \text{ crash} \]

\[ \forall C. \quad C[e] \xrightarrow{*} \text{BAD} \implies \text{BAD} \in C \]
## Crash free

<table>
<thead>
<tr>
<th></th>
<th>Crash free?</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD</td>
<td>NO</td>
</tr>
<tr>
<td>(1, BAD)</td>
<td>NO</td>
</tr>
<tr>
<td>( \forall x. \text{BAD} )</td>
<td>NO</td>
</tr>
<tr>
<td>( \forall x. \text{case x of } { \text{[]} \rightarrow \text{BAD}; (p:ps) \rightarrow p } )</td>
<td>NO</td>
</tr>
<tr>
<td>(1, True)</td>
<td>YES</td>
</tr>
<tr>
<td>( \forall x. x+1 )</td>
<td>YES</td>
</tr>
<tr>
<td>( \forall x. \text{if } (x\times x \geq 0) \text{ then True else BAD} )</td>
<td>Umm.. YES</td>
</tr>
</tbody>
</table>

**Conclusion:** BAD ∈ e is not enough! It is undecidable whether or not e is crash-free.
When does f “satisfy” a contract?

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

- Hence:  \( e \text{ crash free } \iff e \in \text{Ok} \)

- This is why we need Any; e.g. \( \text{fst } \in (\text{Ok, Any}) \rightarrow \text{Ok} \)
Question 3: what if \( p \) diverges?

e.g. \( \text{True} \in \{ x | \text{loop} \} \) ???

Our decision: yes. (Same reason as for \( e \).)
# When does f “satisfy” a contract?

<table>
<thead>
<tr>
<th>Condition</th>
<th>Equivalent Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e \in { x \mid p } )</td>
<td>( e \uparrow ) or ((e \text{ is crash-free and } p[e/x] \not\rightarrow^* {\text{BAD, False}}))</td>
</tr>
<tr>
<td>( e \in x : t_1 \rightarrow t_2 )</td>
<td>( e \uparrow ) or ( \forall e_1 \in t_1. (e e_1) \in t_2[e_1/x] )</td>
</tr>
<tr>
<td>( e \in (t_1, t_2) )</td>
<td>( e \uparrow ) or ((e \rightarrow^* (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2))</td>
</tr>
<tr>
<td>( e \in \text{Any} )</td>
<td>( e \uparrow ) or ( (e \rightarrow^* (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2))</td>
</tr>
</tbody>
</table>

- **Question 4:** what if \( p \) crashes?
  - e.g. \( \text{True} \in \{ x \mid \text{BAD} \} \) ???

- Our decision: no. Treat \text{BAD} like \text{False}. 

---
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 \( t \)?
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 \(t\)

\[e \triangleright \{x \mid p\} = \begin{cases} \text{case } p[e/x] \text{ of} \\ True & \to e \\ False & \to \text{BAD} \end{cases}\]

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

\[e \triangleright \text{Any} = \text{UNR}\]
What is \( (e \triangleright t) \)?

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

\[
e \triangleright \{x \mid p\} = \text{case } p[e/x] \text{ of }
\begin{align*}
\text{True} & \rightarrow e \\
\text{False} & \rightarrow \text{UNR}
\end{align*}
\]

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

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

\[ \text{\(\forall v. \text{head} \ (\text{case \ not \ (null \ v) \ of} \ \text{True} \to v \ \text{False} \to \text{UNR})\)} \]

Now inline 'not' and 'null'

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

Now inline 'head'

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

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

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

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

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 $t$

**Grand Theorem**

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

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

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

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

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

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

Crashes more often
$e_1 \leq e_2$ iff
$\forall C. \ C[e_2] \Rightarrow * BAD$
$\Rightarrow C[e_1] \Rightarrow * BAD$
Contracts that loop

\( x.BAD \in \{x \mid \text{loop} \} \) ?

- **NO:** \( x.BAD \) is not cf

- **BUT:** \( (\forall x. B) \triangleright \{x \mid \text{loop} \} \)
  
  = case loop of ... = loop, which is cf

- **Solution:**

  \[
  e \triangleright_{N} \{x \mid p\} = \text{case } \text{fin}_{N} \, p[e/x] \text{ of True } \rightarrow e \\
  \text{False } \rightarrow \text{BAD}
  \]

  - Reduction under \( \text{fin}_{N} \) decrements \( N \)
  - \( \text{fin}_{0} \, p \rightarrow \text{True} \)
  - 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 
  \[ e \in t \iff e \triangleright t \text{ is crash free} \]

\{x|p\} well-formed if \( p \) is crash free
\( x: t_1 \rightarrow t_2 \) well-formed if \( \forall e \in t_1, t_2[e/x] \) well-formed
Practical aspects
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
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]
Suppose we compute that $e \triangleleft t = \text{\textbackslash xs. not (null xs)}$

Should we inline null, not?

**No:** null, not can’t crash ($\text{null} \in \text{Ok}, \text{not} \in \text{OK}$)
And hence $e \triangleleft t$ is crash-free

But suppose $e \triangleleft t = \text{\textbackslash xs. if not (null xs) then BAD else 1}$
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 \( \text{BAD} \in e_s \)
3. If so, inline each function called in \( e_s \), and go to (1)
4. Stop when tired
Using a theorem prover

\[ f \in x: \text{Ok} \implies \{ y \mid y > x \} \implies \text{Ok} \]

\[ g \in \text{Ok} \]

\[ g(i) = f(i)(i+8) \]

Feed this theorem of arithmetic to an external theorem prover.

\[ g \triangleright \text{Ok} = \text{case } (i+8 > i) \text{ of } \]

\[ \text{True } \implies f(i)(i+8) \]

\[ \text{False } \implies \text{BAD} \]

Remember, we replace \( f \) by \( (f \triangleleft tf) \)
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 ($t_1 \land t_2$, $t_1 \lor t_2$, recursive contracts...)
What is \((e \triangleright t)\)?

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

\[
\begin{align*}
\text{\(r \in \{\text{BAD, UNR}\}\)} & \quad \text{BAD} & \quad \text{UNR} \\
\neg \text{BAD} = \text{UNR} & \quad \neg \text{UNR} = \text{BAD} & \quad e \triangleright t = e \times t & \quad e \triangleright t = e \times t \\
\end{align*}
\]

\[
\begin{align*}
e \times \{x \mid p\} & = e \text{ `seq` case } \text{fin } p[e/x] \text{ of } \{\text{True } \rightarrow e; \text{ False } \rightarrow r\}
\end{align*}
\]

\[
\begin{align*}
e \times x: t_1 \rightarrow t_2 & = e \text{ `seq` } \lambda v. \text{ let } \{x = (v \times t_1)\} \text{ in } (e x) \times t_2
\end{align*}
\]

\[
\begin{align*}
e \times (t_1, t_2) & = \text{case } e \text{ of } \{(e_1, e_2) \rightarrow (e_1 \times t_1, e_2 \times t_2)\}
\end{align*}
\]

\[
\begin{align*}
e \times \text{Any} & = \neg r
\end{align*}
\]