Lazy SmallCheck

Matthew Naylor

(joint work with Fredrik Lindblad and Colin Runciman)
The problem

**GIVEN**

a program and a property of that program which is expected to hold,

**TRY TO FIND**

a set of inputs that falsifies the property.
A problem instance (1)

A program:

\[
\begin{align*}
\text{merge } & [1,4] \ [2,3] \rightarrow [1,2,3,4] \\
\text{merge } & [] \ ys = ys \\
\text{merge } & xs [] = xs \\
\text{merge } & (x:xs) \ (y:ys) \\
& | x \leq y = x : \text{merge } xs \ (y:ys) \\
& | \text{otherwise} = y : \text{merge } (x:xs) \ ys \\
\text{ord } & [] = \text{True} \\
\text{ord } & [x] = \text{True} \\
\text{ord } & (x:y:ys) = x \leq y \land \text{ord } (y:ys)
\end{align*}
\]

\[
\begin{align*}
\text{e.g. } & \text{ord } [1,2,3,4] \rightarrow \text{True}
\end{align*}
\]
A problem instance (2)

A property:

```haskell
prop_ordMerge :: [Char] -> [Char] -> Bool
prop_ordMerge xs ys =
    ord xs && ord ys ==> ord (merge xs ys)
```

The problem is to try to find an \( \text{xs} \) and a \( \text{ys} \) such that \( \text{prop\_ordMerge}\ \text{xs} \ \text{ys} \) is False.
Solution 1: SmallCheck

Tests the property on all inputs of tree-depth $\leq d$.

 GHCi> depthCheck 7 prop_ordMerge
 Completed 187690000 test(s) without failure.
 But 187673616 did not meet $\Rightarrow$ condition.

 only 16384 tests were relevant

 took 147 seconds

* After QuickCheck by Claessen and Hughes
Solution 2: Lazy SmallCheck

Tests all inputs up to tree-depth \( d \), as before.

\[
\text{GHCi}\> \text{depthCheck 7 prop\_ordMerge}
\]

OK, passed 3749737 tests.

depth 7, as before

now took 0.6 seconds

245 times faster!
This talk

1. **How** Lazy SmallCheck works.

2. **Experiences** with Lazy SmallCheck.

3. Relationship with **lazy narrowing**.

4. **Extensions**: parallel evaluation & residuation.
1. How Lazy SmallCheck works

The intuition and the algorithm
Intuition (1)

First, observe the behaviour of \texttt{ord} on partially defined inputs:

\begin{itemize}
  \item \texttt{GHCi> ord (1:2: \perp)}
  
  \texttt{*** Exception: bottom}

  \item \texttt{GHCi> ord (1:0: \perp)}

  \texttt{False}
\end{itemize}

For all \texttt{x}, \texttt{ord (1:0:x)} is \texttt{False}
Likewise for \texttt{prop\_ordMerge}: \\

\texttt{GHCi> prop\_ordMerge (1:0: \bot) \bot} \\
\texttt{True} \\

For all \(x, y\), \texttt{prop\_ordMerge (1:0:x) y} is True.

No need to generate more-refined inputs of this form.
Algorithm (1)

How are partially-defined inputs represented?
e.g. \((0:1: \bot, 2: \bot)\)
Algorithm (2)

Haskell-like psuedocode: initially, \texttt{error \"\"}

\begin{verbatim}
refute prop input =
case prop input of
  True     -> return ()
  False    -> display input >> exit
  Error pos -> mapM_ (refute prop) (refine input pos)
\end{verbatim}

the argument passed to \texttt{error}

i.e. “position of demand”

possible using \texttt{Control.Exception} module
2. Experiences

Applying Lazy SmallCheck to Circuit Design
A binary encoder:

\[
\text{encode} :: \text{[Bool]} \rightarrow \text{[Bool]}
\]

\[
\text{encode} \ [\text{False, False, True}] \rightarrow \ [\text{False, True}]
\]

e.g.

A bit-vector to integer converter:

\[
\text{num} :: \text{[Bool]} \rightarrow \text{Int}
\]

\[
\text{num} \ [\text{False, True}] \rightarrow 2
\]

e.g.

A predicate for “one-hot” checking:

\[
\text{oneHot} \ [\] = \text{False}
\]

\[
\text{oneHot} \ (x:x) \rightarrow
\]

| \[ x = \text{not} \ (\text{or} \ xs) \]
| otherwise = \text{oneHot} \ xs

\[
\text{oneHot} \ [\text{False, False, True}] \rightarrow \text{True}
\]
**Property 1**

A property:

\[
\text{prop\_encode} \; \text{xs} = \begin{cases}
\text{oneHot} \; \text{xs} \implies (\text{num} \; (\text{encode} \; \text{xs}) = n) \\
\text{where} \\
\quad n = \text{length} \; (\text{takeWhile not} \; \text{xs})
\end{cases}
\]

Timings:

<table>
<thead>
<tr>
<th>Depth</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallCheck</td>
<td>5s</td>
<td>11s</td>
<td>22s</td>
<td>7hr</td>
<td>3mil yrs</td>
</tr>
<tr>
<td>Lazy SmallCheck</td>
<td>0s</td>
<td>0s</td>
<td>0s</td>
<td>0s</td>
<td>1s</td>
</tr>
</tbody>
</table>

Can't be checked with a SAT solver

predicted! exponential v. linear
Circuit 2

A binary multiplexor:

```haskell
binMux :: [Bool] -> [[Bool]] -> [Bool]

e.g. binMux [False,True] [[False,False], [True,False], [False,True], [True,True]] → [False,True]
```

A predicate to check that a list of lists is a matrix:

```haskell
isMatrix xs = all (=== head ns) ns
where
    ns = map length xs
```
Property 2

A property:

\[
\text{prop\_binMux\_sel\_xs} = \\
\text{length\_xs == 2 ^ length\_sel} \\
\text{&& isMatrix\_xs} \\
\text{==> binMux\_sel\_xs = xs !! num\_sel}
\]

Timings:

<table>
<thead>
<tr>
<th>Depth</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallCheck</td>
<td>1s</td>
<td>135s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lazy SmallCheck</td>
<td>0s</td>
<td>0s</td>
<td>0s</td>
<td>3s</td>
</tr>
</tbody>
</table>
A binary decoder:

\[
\begin{align*}
    \text{decode } [] &= [\text{True}] \\
    \text{decode } (x:xs) &= \text{concatMap } (\lambda y \rightarrow [\text{not } x \land y, x \land y]) \ (\text{decode } xs)
\end{align*}
\]

\text{e.g.}

\[
\text{decode } [\text{True, True}] \rightarrow [\text{False, False, False, False, True}]
\]
Property 3

A property:

\[
\text{prop\_encDec } xs = \text{encode } (\text{decode } xs) == xs
\]

Timings:

<table>
<thead>
<tr>
<th>Depth</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallCheck</td>
<td>2s</td>
<td>10s</td>
<td>76s</td>
</tr>
<tr>
<td>Lazy SmallCheck</td>
<td>2s</td>
<td>13s</td>
<td>87s</td>
</tr>
</tbody>
</table>

overhead: applying a hyper strict property to partial inputs!
Graphical comparison

- List.prop_ordMerge
- Countdown.lemma1
- Sad.prop_binSad
- Mux.prop_binMux
- SumPuz.prop_sound
- Encode.prop_encode

Lazy SmallCheck vs. SmallCheck
3. Lazy Narrowing

Similar to Lazy SmallCheck
What is lazy narrowing?

A lazy evaluation strategy that permits unbound variables in inputs.

If \( \text{HNF}(e) \) is an unbound variable \( v \),

\[
\text{case } e \text{ of } \\
c_1 \ v_1 \rightarrow e_1 \\
\cdots \\
c_n \ v_n \rightarrow e_n
\]

Then, for each \( i \) in \( 1..n \), bind \( v \) to \( c_i \ v_i \) and evaluate \( e_i \).
Example of lazy narrowing

unbound variables of type `Bool`

ord \([x, y, z]\)

\[\rightarrow \text{True} \quad \text{if} \quad x = \text{False} \quad \text{and} \quad y = \text{False}\]

\[\ldots\]

\[\rightarrow \text{False} \quad \text{if} \quad x = \text{True} \quad \text{and} \quad y = \text{False}\]

\[\ldots\]

result can sometimes be computed without knowing all the variables

just like Lazy SmallCheck!
The difference (1)

View an inlined program as a tree whose nodes represent case expressions and edges represent case alternatives:

If cases evaluated by Lazy Narrowing = \( n \)
then those by Lazy SmallCheck = \( O(n^2) \)

warning: very crude measure!
The difference (2)

Lazy narrowing is **more efficient**.

But Lazy SmallCheck is **effective** nevertheless
- And it works in standard Haskell (with Control.Exception).

How big is the performance difference **in practice**?

Would be interesting to **compare** Lazy SmallCheck with:
1. Fredrik's property-directed test generator [TFP'07]
2. Reach, a target-directed generator by Colin and I [SCAM'07]
3. MCC, the Munster Curry Compiler
Lazy evaluation aids automatic testing.

The demand of the property can guide automatic generation of relevant test cases.

Lazy SmallCheck exploits this in a standard Haskell (plus Control.Exception) library.

Future plan: merge with SmallCheck.

available from Hackage!