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,

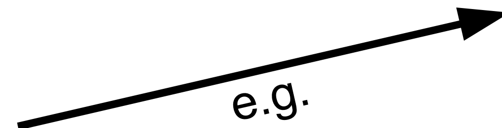
TRY TO FIND

a set of **inputs** that **falsifies** the property.

A problem instance (1)

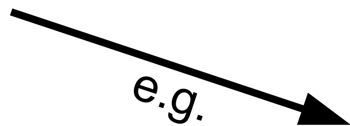
A **program**:

```
merge [1,4] [2,3] → [1,2,3,4]
```




```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y     = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys

ord [] = True
ord [x] = True
ord (x:y:ys) = x <= y && ord (y:ys)
```




```
ord [1,2,3,4] → True
```

A problem instance (2)

A **property**:

```
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 `xs` and a `ys` such that `prop_ordMerge xs ys` is `False`.

Solution 1: SmallCheck*

Tests the property on all inputs of tree-depth <= d.

depth 7

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

only 16384 tests were relevant

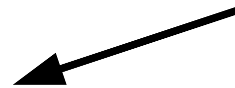
took 147 seconds

* After QuickCheck by Claessen and Hughes

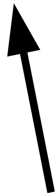
Solution 2: Lazy SmallCheck

Tests all inputs up to tree-depth **d**, **as before**.

depth **7**, **as before**



```
GHCi> depthCheck 7 prop_ordMerge
OK, passed 3749737 tests.
```



now took **0.6** seconds

This talk

1. **How** Lazy SmallCheck works.

2. **Experiences** with Lazy SmallCheck.

3. Relationship with **lazy narrowing**.

~~**4**. **Extensions**: parallel evaluation & residuation.~~

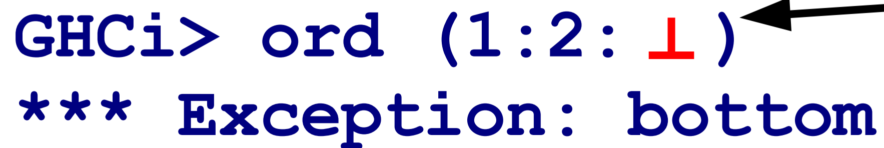
1. How Lazy SmallCheck works

The intuiton and the algorithm

Intuition (1)

First, observe the behaviour of `ord` on partially defined inputs:

`⊥ = error "bottom"`




```
GHCi> ord (1:0:⊥)
False
```

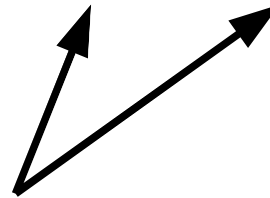
For all *x*, `ord (1:0:`*x*`)` is `False`

Intuition (2)

Likewise for `prop_ordMerge`:

```
GHCi> prop_ordMerge (1:0:⊥) ⊥
True
```

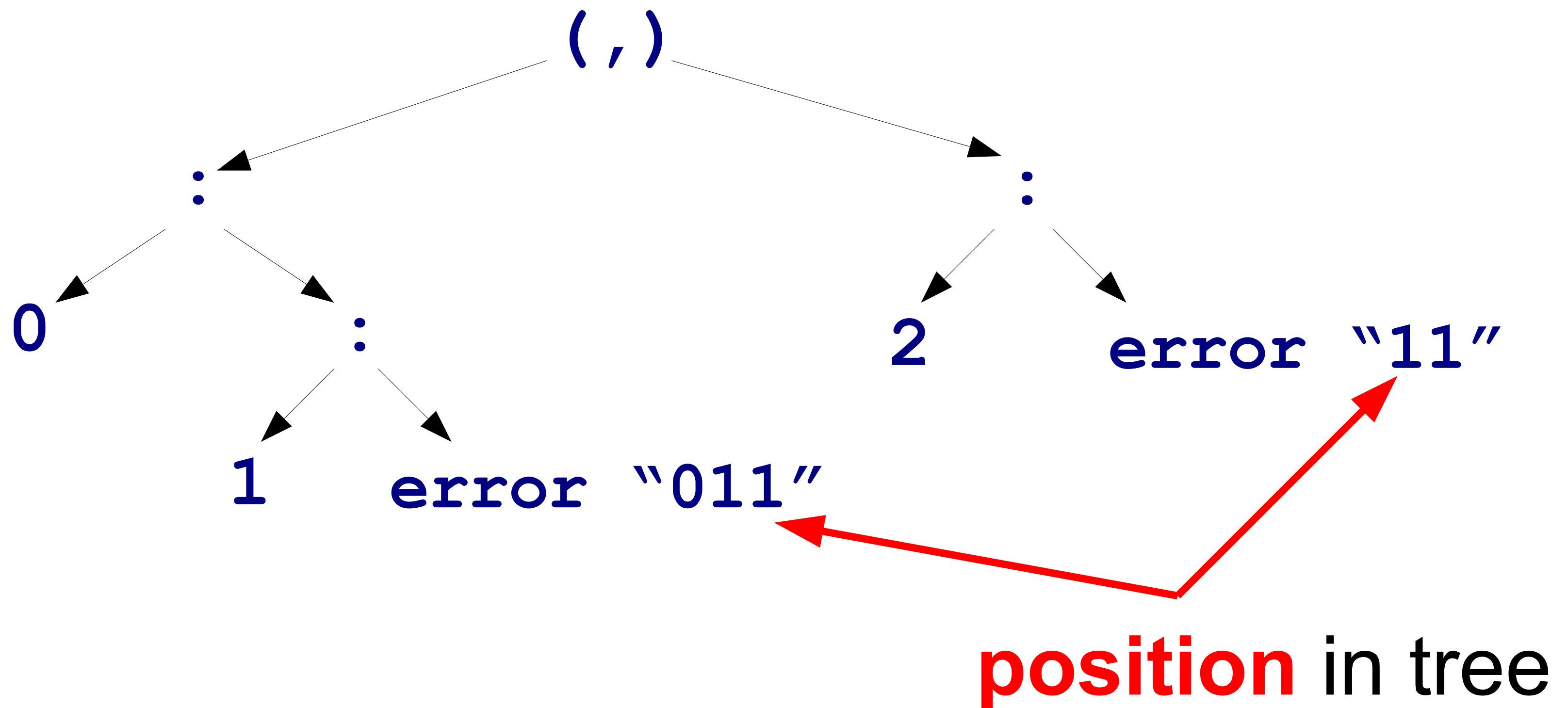
For all *x*, *y*, `prop_ordMerge (1:0:`*x*`)` *y* is `True`.



No need to generate more-refined inputs of this form

Algorithm (1)

How are partially-defined inputs represented?
e.g. `(0:1:⊥, 2:⊥)`



Algorithm (2)

Haskell-like psuedocode:

initially, **`error ""`**

```
refute prop input =
  case prop input of
    True      -> return ()
    False     -> display input >> exit
    Error pos -> mapM_ (refute prop) (refine input pos)
```

the argument passed to **`error`**
i.e. "position of demand"

possible using **Control.Exception** module

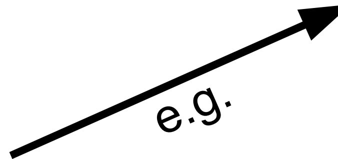
2. Experiences

Applying Lazy SmallCheck to Circuit Design

Circuit 1

`encode [False, False, True]` ⟶ `[False, True]`

A binary encoder:




```
encode :: [Bool] -> [Bool]
```

A bit-vector to integer converter:

```
num :: [Bool] -> Int
```

`num [False, True]` ⟶ `2`

A predicate for "one-hot" checking:

```
oneHot [] = False
oneHot (x:xs)
  | x = not (or xs)
  | otherwise = oneHot xs
```

`oneHot [False, False, True]` ⟶ `True`

Property 1

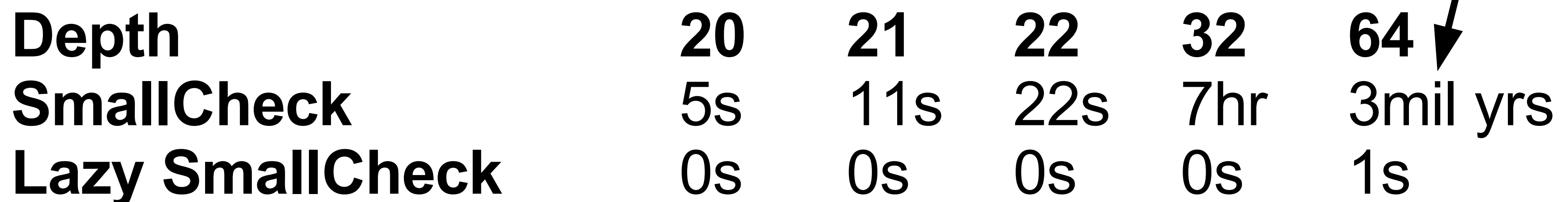
A property:

Can't be checked with a SAT solver

```
prop_encode xs =
    oneHot xs ==> (num (encode xs) == n)
  where
    n = length (takeWhile not xs)
```

Timings:

predicted!



Depth	20	21	22	32	64
SmallCheck	5s	11s	22s	7hr	3mil yrs
Lazy SmallCheck	0s	0s	0s	0s	1s

exponential v. linear

Circuit 2

A binary multiplexor:

```
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:

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

Property 2

A property:

property is spine strict

```
prop_binMux sel xs =
     length xs == 2 ^ length sel
  && isMatrix xs
 ==> binMux sel xs = xs !! num sel
```

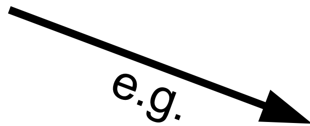
Timings:

Depth	6	7	8	16
SmallCheck	1s	135s	-	-
Lazy SmallCheck	0s	0s	0s	3s

Circuit 3

A binary decoder:

```
decode [] = [True]
decode (x:xs) =
  concatMap (\y -> [not x && y, x && y]) (decode xs)
```




```
decode [True, True] ⟶ [False, False, False, True]
```

Property 3

A property:

property is hyper strict!

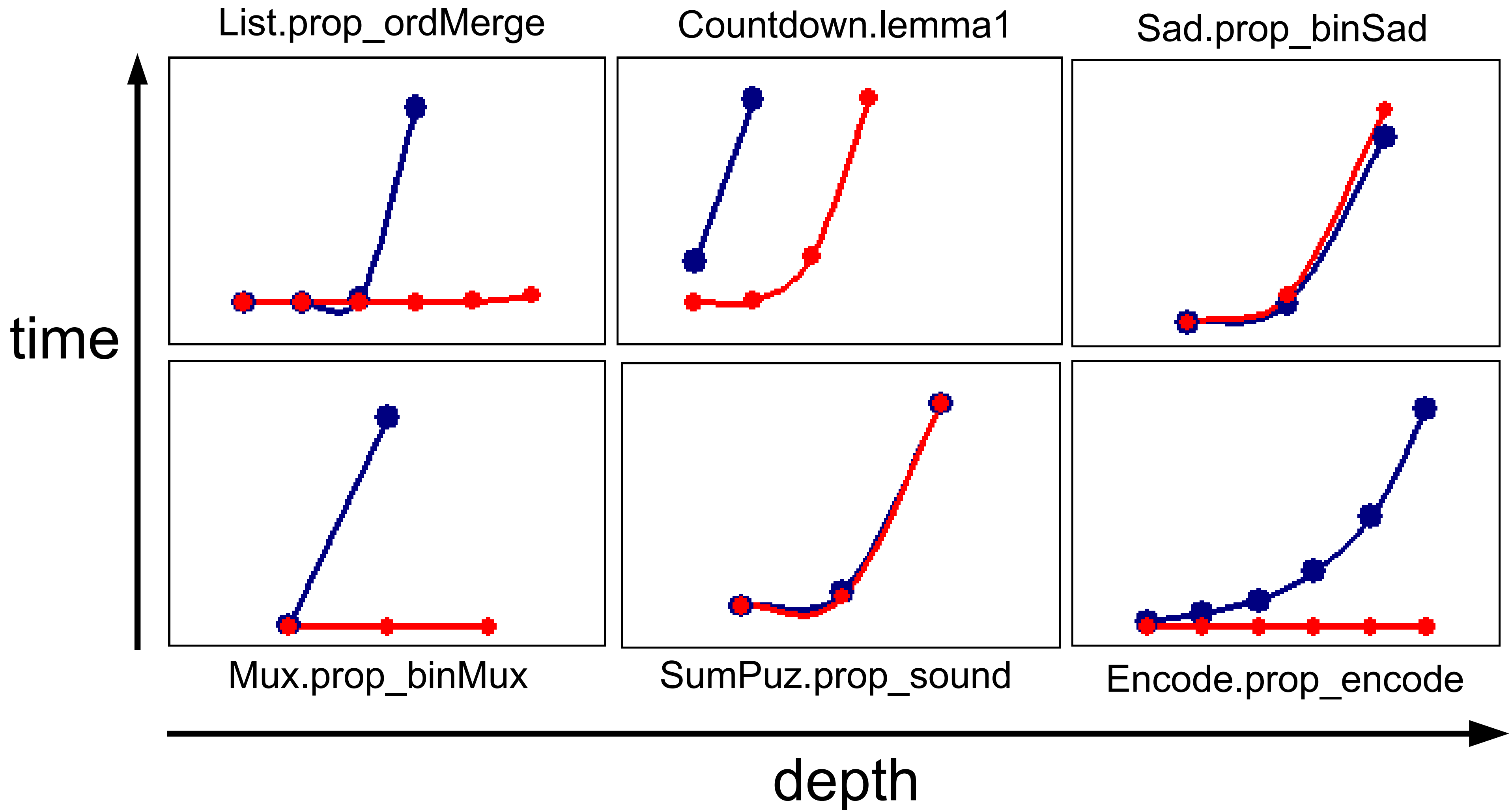
```
prop_encDec xs = encode (decode xs) == xs
```

Timings:

Depth	**9**	**10**	**11**
SmallCheck	2s	10s	76s
Lazy SmallCheck	2s	13s	87s

overhead: applying a hyper strict property to partial inputs!

Graphical comparison



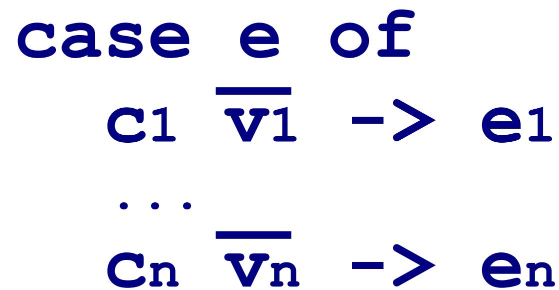
3. Lazy Narrowing

Similar to Lazy SmallCheck

What is lazy narrowing?

A lazy evaluation strategy that permits unbound variables in inputs.

if HNF(**e**) is an unbound variable ***v***,



then, for each **i** in **1**..**n**, bind ***v*** to $c_i\ \overline{v_i}$ and evaluate e_i

Example of lazy narrowing

unbound variables of type `Bool`

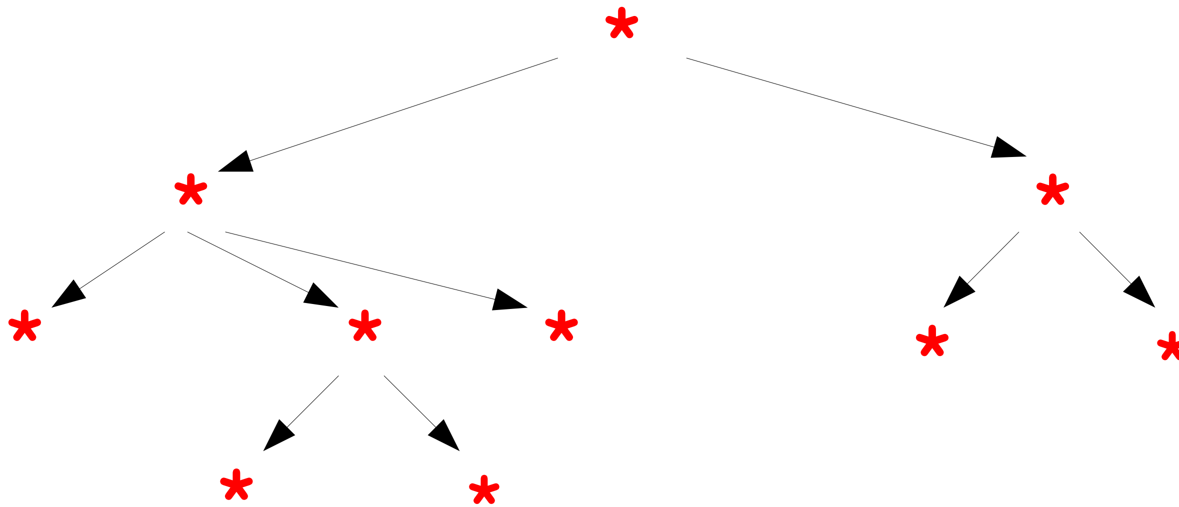
```
ord [x,y,z]
  →  True      if x = False and y = False
     ...
  →  False     if x = True and y = False
     ...
```

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 represents 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

Conclusion

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.