Splittable Random Number Generators

Michał Pałka   Koen Claessen

Chalmers University of Technology
Gothenburg, Sweden

Haskell Implementors' Workshop, 2012

```haskell
import Test.QuickCheck

newtype Int14 = Int14 Int
  deriving Show

instance Arbitrary Int14 where
  arbitrary = fmap Int14 $ choose (0, 13)

prop_shouldFail (_, Int14 a) (Int14 b) = a /= b
```

```
import Test.QuickCheck

newtype Int14 = Int14 Int
  deriving Show

instance Arbitrary Int14 where
  arbitrary = fmap Int14 $ choose (0, 13)

prop_shouldFail (_, Int14 a) (Int14 b) = a /= b

*Flop> quickCheckWith stdArgs { maxSuccess = 10000 }
   prop_shouldFail
+++ OK, passed 10000 tests.
```

From System.Random

```
stdSplit :: StdGen -> (StdGen, StdGen)
stdSplit g = ...
-- no statistical foundation for this!
```

Splittable RNG

```
class RandomGen g where
   next      :: g -> (Int, g)
   split     :: g -> (g, g)
```

Needed for random lazy data!

Plan so far

1. Take linear RNG
2. Add splitting

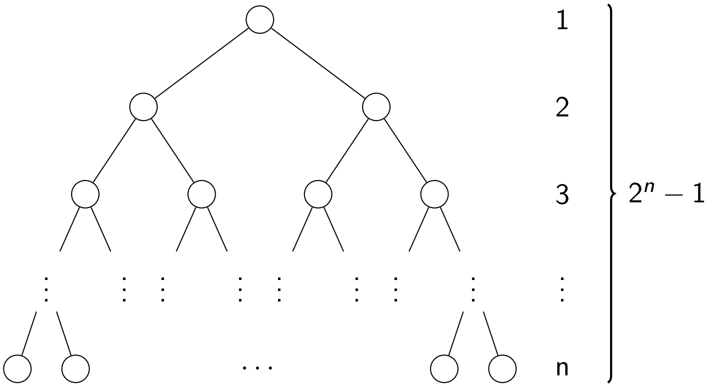
What do linear RNGs have?

- Period
- Seed size
- Generator passes statistical tests

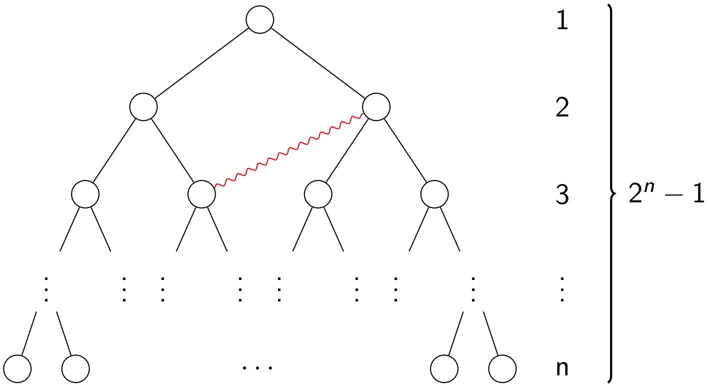
What do linear RNGs have?

- Period ($2^{\textbf{seed size}}$)
- Seed size
- Generator passes statistical tests

Splitting tree



Splitting tree



Goodness criterion (crypto)

Generator `g :: IO Rand`
Program `p :: Rand -> Bool` (discriminator)
Perfect generator `rand_org :: IO Rand`

If $P(\texttt{liftM p g} \to \texttt{True}) > P(\texttt{liftM p rand\_org} \to \texttt{True}) + \epsilon$
then `p` is a discriminator.

Block ciphers

$$\begin{tikzcd} 0 \arrow[d, "\mathbf{enc}\ k"'] & 1 \arrow[d, "\mathbf{enc}\ k"'] & \cdots \\ a_0 & a_1 & \end{tikzcd}$$

Block ciphers

$$
\begin{tikzcd}
0 \arrow[d, "\textbf{enc}\ k"'] & 1 \arrow[d, "\textbf{enc}\ k"'] & \cdots \\
a_0 \arrow[r, dashed, no head, "\not\approx"] & a_1 &
\end{tikzcd}
$$

Conventional RNGs

- Period length
- "Works for me"
- Statistical tests

Block ciphers

- Bits of security
- Concrete definition of randomness
- Peer review, proofs

$$
\begin{tikzcd}
 & k, i \arrow[dl] \arrow[dr] \arrow[r, "\textbf{next}"] & (\textbf{enc}\; k\; i, 0), (k, i) \\
\textbf{enc}\; k\; i, 0 & & k, i+1
\end{tikzcd}
$$

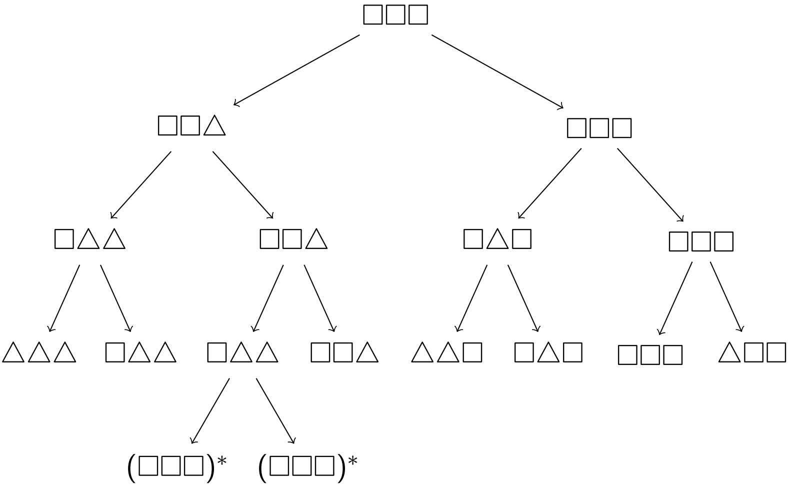
$$\textbf{enc}\; k \not\approx \textbf{enc}\; k' \text{ when } k \neq k'$$

How fast?

QuickCheck is split-intensive

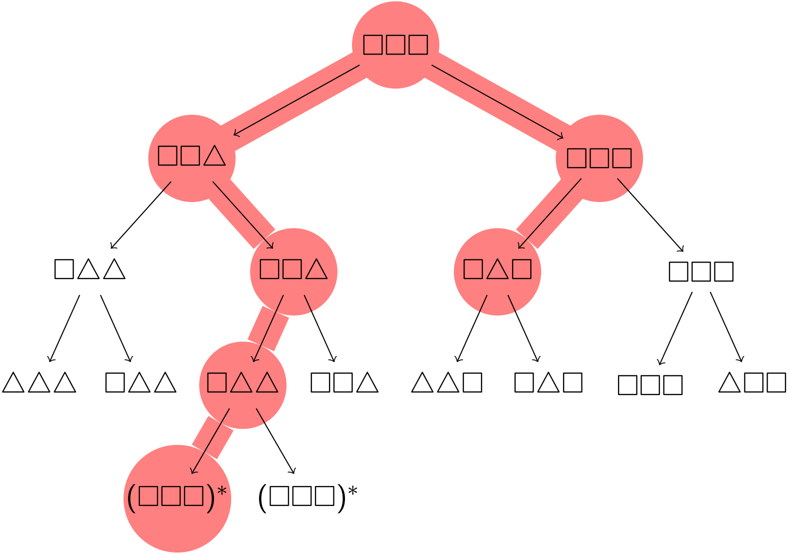
Additional counter

$$\square\triangle\square = k, \square\triangle\square, i$$



Additional counter

$\square\triangle\square = k, \square\triangle\square, i$



Implementation

- Uses 256-bit ThreeFish cipher (part of Skein, SHA-3 candidate)
- Reference C implementation (FFI)

The numbers

- QuickCheck properties: 24% slower
- Only 6% spent in ThreeFish
- Linear generation: 1M `Word32`s in 68 ms (20% for TF)
- 4x faster than StdGen
- 5x slower than mwc-random?

(for x86-64)

Other important factors

- Allocation
- Rest of the Random API
- Low-level optimisation
- We need to fix the `Random` instances!

Weaknesses

- May loop in *left*
- How serious?
- Possible to aleviate it

Conclusions

- Need stronger guarantees than what regular RNGs are promising
- Hard to split without strong guarantees
- Cryptographic block ciphers give acceptable performance
- Haskell users deserve a good default RNG!