Splittable Random Number Generators

Michał Pałka       Koen Claessen

Chalmers University of Technology
Gothenburg, Sweden

Haskell Implementors’ Workshop, 2012
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!
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 ($2^{\text{seed size}}$)
- Seed size
- Generator passes statistical tests
Splitting tree

\[ 2^n - 1 \]
Splitting tree
Goodness criterion (crypto)

Generator \( g :: \text{IO Rand} \)
Program \( p :: \text{Rand} \rightarrow \text{Bool} \) (discriminator)
Perfect generator \( \text{rand\_org} :: \text{IO Rand} \)

If \( P(\text{liftM} \ p \ g \rightarrow \text{True}) > P(\text{liftM} \ p \ \text{rand\_org} \rightarrow \text{True}) + \epsilon \)
then \( p \) is a discriminator.
Block ciphers

0
\downarrow
\text{enc } \mathit{k}
\hspace{2em}
1
\downarrow
\text{enc } \mathit{k}
\hspace{2em} \cdots
\hspace{2em}
\downarrow
\hspace{2em}
a_0
\hspace{2em}
a_1
Block ciphers

\[
\begin{array}{c}
0 \\
\text{enc } k \\
\downarrow \\
a_0
\end{array}
\quad
\begin{array}{c}
1 \\
\text{enc } k \\
\downarrow \\
a_1
\end{array}
\quad \cdots
\]

\[\not\approx\]
Conventional RNGs

- Period length
- “Works for me”
- Statistical tests

Block ciphers

- Bits of security
- Concrete definition of randomness
- Peer review, proofs
\[ k, i \xrightarrow{\text{next}} (\text{enc } k, i, 0), (k, i) \]

- \text{enc } k, i, 0
- \( k, i + 1 \)

\textbf{enc } k \not\simeq \textbf{enc } k' \text{ when } k \neq k'
How fast?

QuickCheck is split-intensive
Additional counter

\[ □△□ = k, □△□, i \]
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 Word32s 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 alleviate 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!