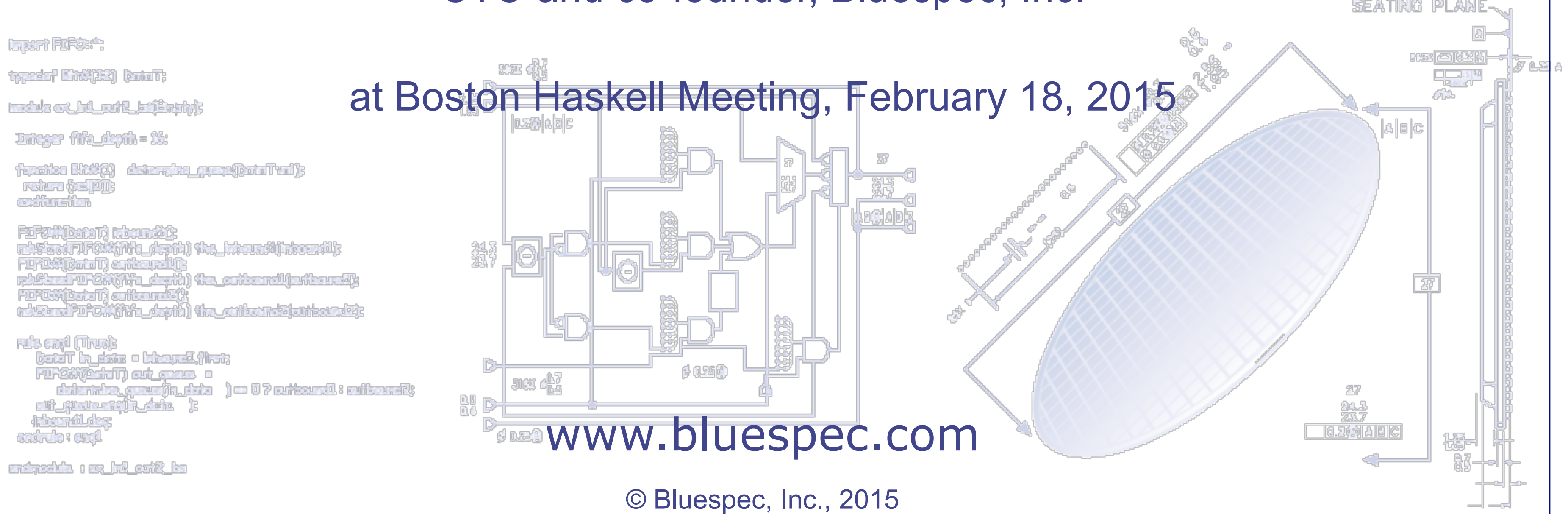
Enigmatic Haskell, Haskellish Enigma

Rishiyur Nikhil
CTO and co-founder, Bluespec, Inc.

at Boston Haskell Meeting, February 18, 2015



www.bluespec.com

And now for something completely useless …

… i.e., there's probably no practical value in coding up Enigma,
nor in obtaining an electronic implementation (using Bluespec BSV),
other than as a fun educational programming exercise.

Introduction

The Enigma was a German cryptography machine used in World War 2.

- It was famously the subject of intense code-breaking efforts, initially by cryptographers in Poland, and later by Alan Turing and his colleagues at Bletchley Park in England, which led to great advances in Computer Science and Engineering.

Cryptol (www.cryptol.net) is a Haskell-based DSL for formal specification, verification and development of existing and new cryptographic algorithms.

Bluespec BSV (www.bluespec.com) is a Haskell-based DSL for designing complex digital circuits, typically implemented in FPGAs and ASICs.

Today we will study and run Cryptol and BSV code that models the Enigma.

Enigma

Enigma: a brief history

- Completely electro-mechanical (no electronics!)
- Patented 1918 by Arthur Scherbius, Germany
- Many models, 1923 – 1940s; steadily increasing complexity (e.g., number of rotors)

Cracking it:

- 1930s: Polish cryptographers Marian Rejewski and colleagues
  - Constructed Polish “bomba” machines to assist in cracking of daily operational settings
- Increasing complexity (and imminent invasion!) outran Polish ability to keep up
- July 1939: Polish cryptographers share knowledge with French and British cryptographers
- Oct 1939: Polish cryptographers evacuated to “PC Bruno”, French cryptography facility near Paris.
  - Collaborate with Bletchley Park, England (incl. Alan Turing)
- June 1940 fall of France: work shifts entirely to Bletchley Park
  - Construction of “bombe” machines to assist in cracking of daily operational settings

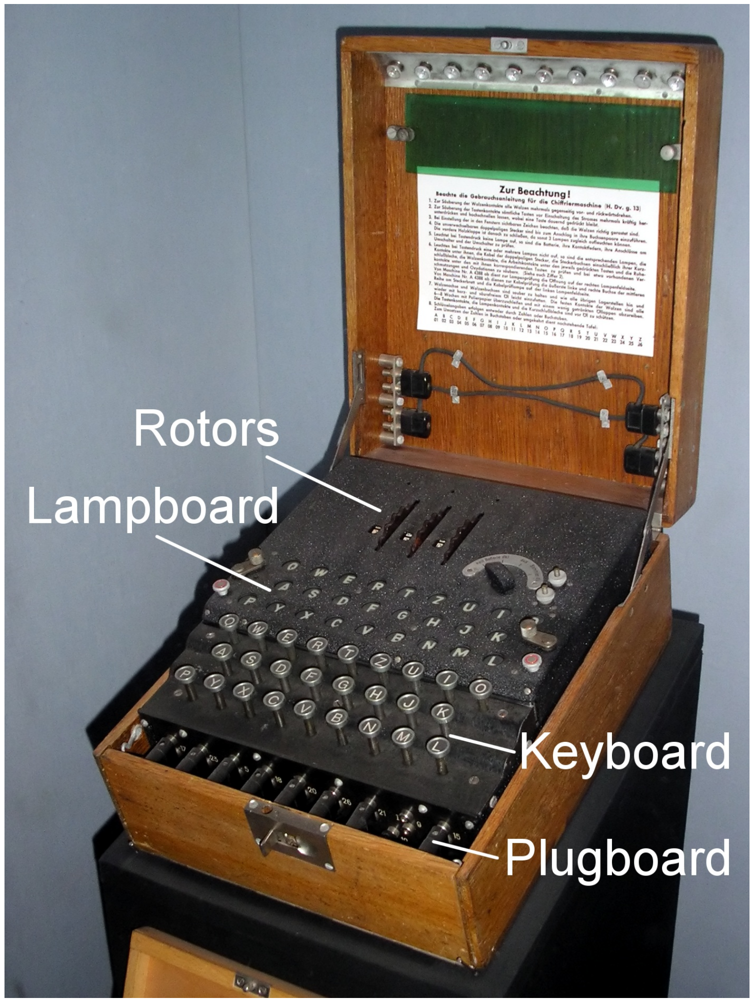
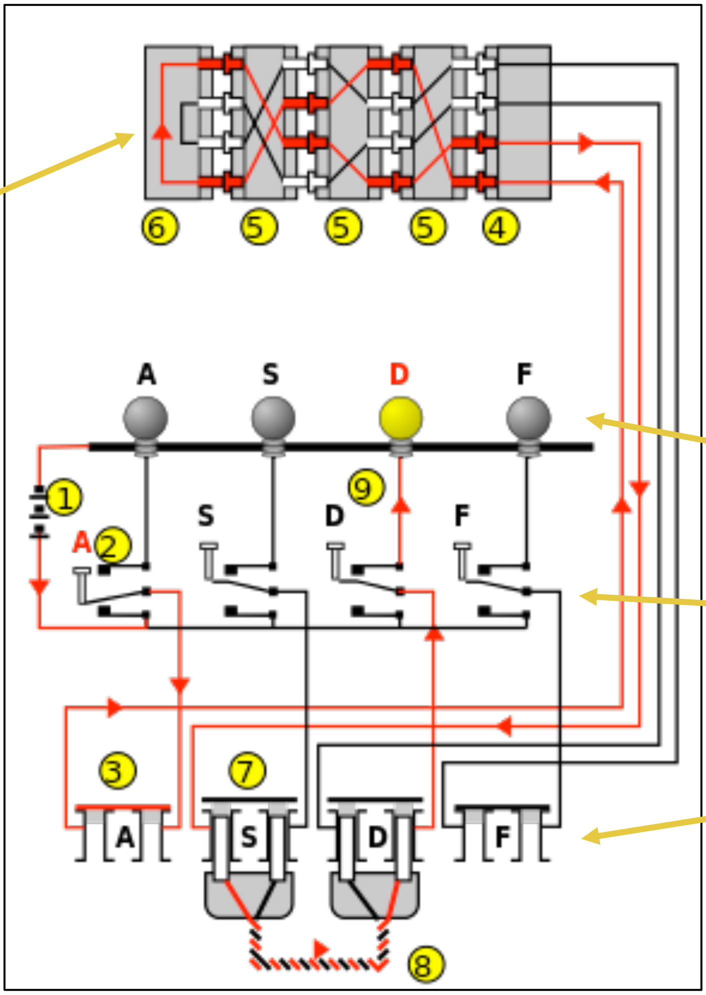


Image: Wikipedia

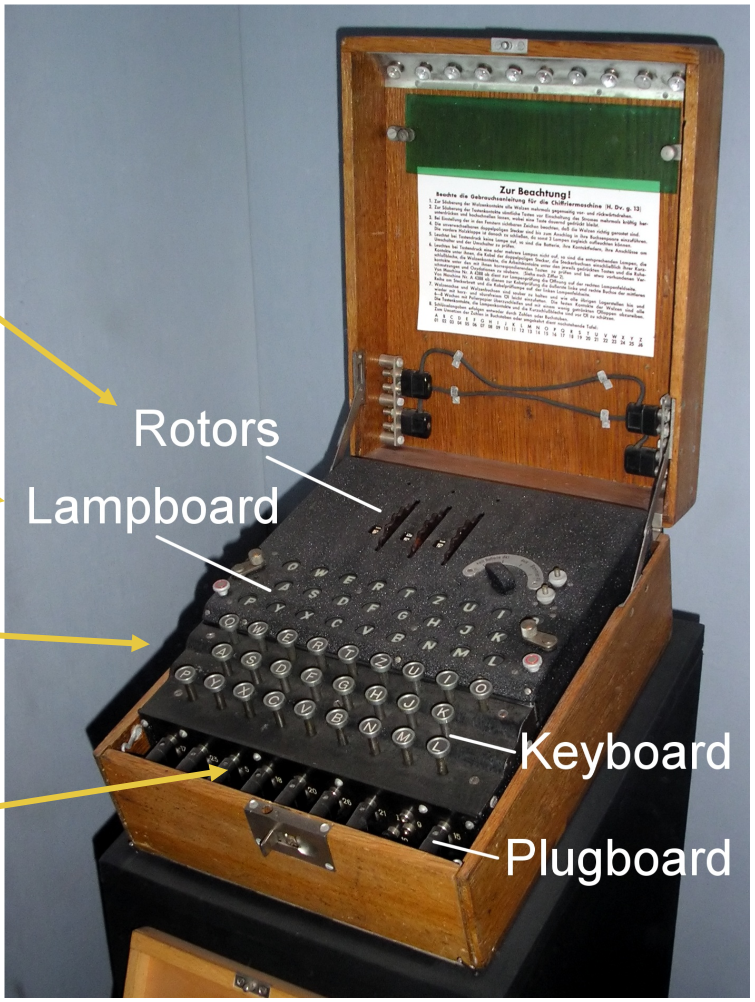
Key contributor to WW II effort!

But total secrecy under British Official Secrets Act ➔ accomplishments emerge only decades later

Enigma: components and operation



Reflector

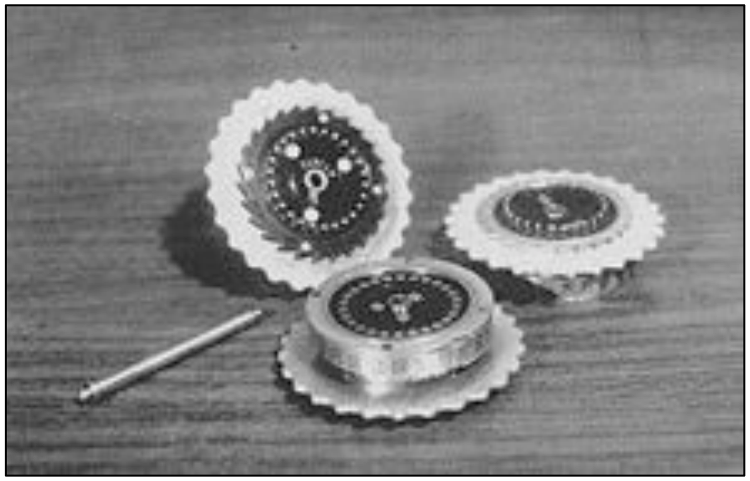


Images: Wikipedia

Polyalphabetic substitution cipher

- Substitution (“scrambling”, “permutation”): cleartext letter → ciphertext letter
- Initialization per message: plugboard settings, choice of rotors, rotor ordering, reflector
- Substitution function changes after each letter

Enigma: rotors



Images: Wikipedia

- Each rotor: 26 contacts on right connect internally (via permutation) to 26 contacts on left
- Brushes on one rotor connect to contact plates on next rotor
- Used in both directions (due to reflector on far left)
- Like old-style car odometer: each revolution (26 clicks), nudges rotor to its left by 1 click

Initializations varied every day, every message:

- From standard set of rotors, pick rotors to install in machine, and in which slot
- Initial offset (# of clicks) for each rotor

Enigma: resources

- Wikipedia (of course!)
- Books:
  - Andrew Hodges, *Alan Turing: The Enigma*, Vintage, 1992
  - Simon Singh. *The code book: the evolution of secrecy from Mary, Queen of Scots, to quantum cryptography*. Doubleday, 1999
- Many movies, including:
  - The Imitation Game (2014)
  - Codebreaker (2011)
  - Decoding Alan Turing (2009)
  - Breaking the Code (1996)
- Many museums around the world with Enigma machines, including:
  - Bletchley Park, England
  - Deutsches Museum, Munich, Germany
  - National Cryptologic Museum, Fort Meade, Maryland, USA
  - Computer History Museum, Mountain View, California, USA

And, a special treat for Boston-area people: Museum of World War II Boston

- www.museumofworldwarII.org
- Natick (20 mins west of Boston)
- Special exhibition of Enigma machines, Feb 14 to May 1, 2015
  - 9 machines, including 4-rotor Naval Enigma
  - U-boat code books, etc.
- Contact them in advance for a visit (can't just walk in)

Cryptol for Enigma

The Enigma Cryptol code shown here is is from the book “Programming Cryptol”

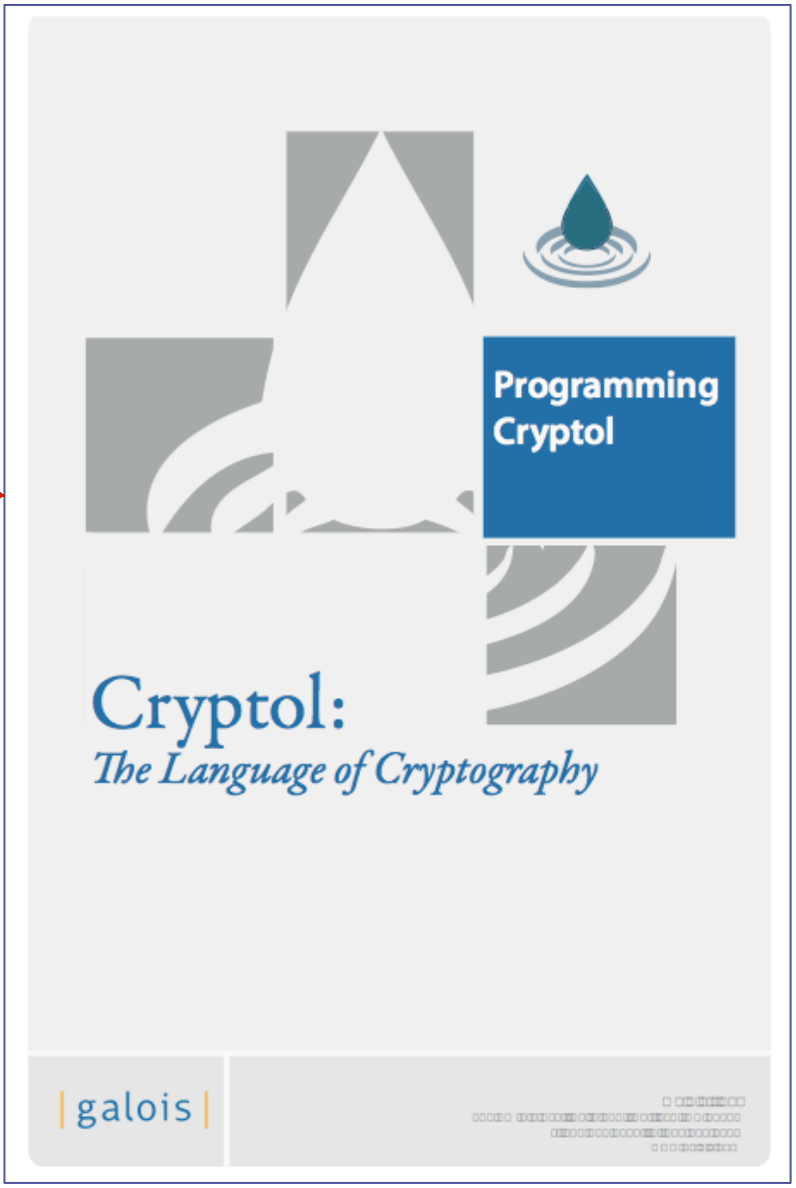
Caveats:

- I’m new to Cryptol, with a total of only a few days of study so far
- Any errors in commentary here are mine, not Galois’

Cryptol

- Developed by Galois, Inc., Portland, Oregon (www.galois.com)
- DSL for cryptography
- Cryptol language and implementation released open-source (www.cryptol.net)
  - Download implementation
  - and PDF book …
  - (including Enigma code)

- Heavily Haskell-influenced
  - Pure, lazy functional language
  - Central concepts: bit sequences, bit field manipulation, Galois field arithmetic
  - Sequences of sequences, comprehensions, parallel comprehensions
  - Polymorphic types, typeclasses
  - Typeclasses for numeric types and size constraints
  - Verfication facilities: Quickcheck-like, plus SMT solvers, etc.



Cryptol

We will now:

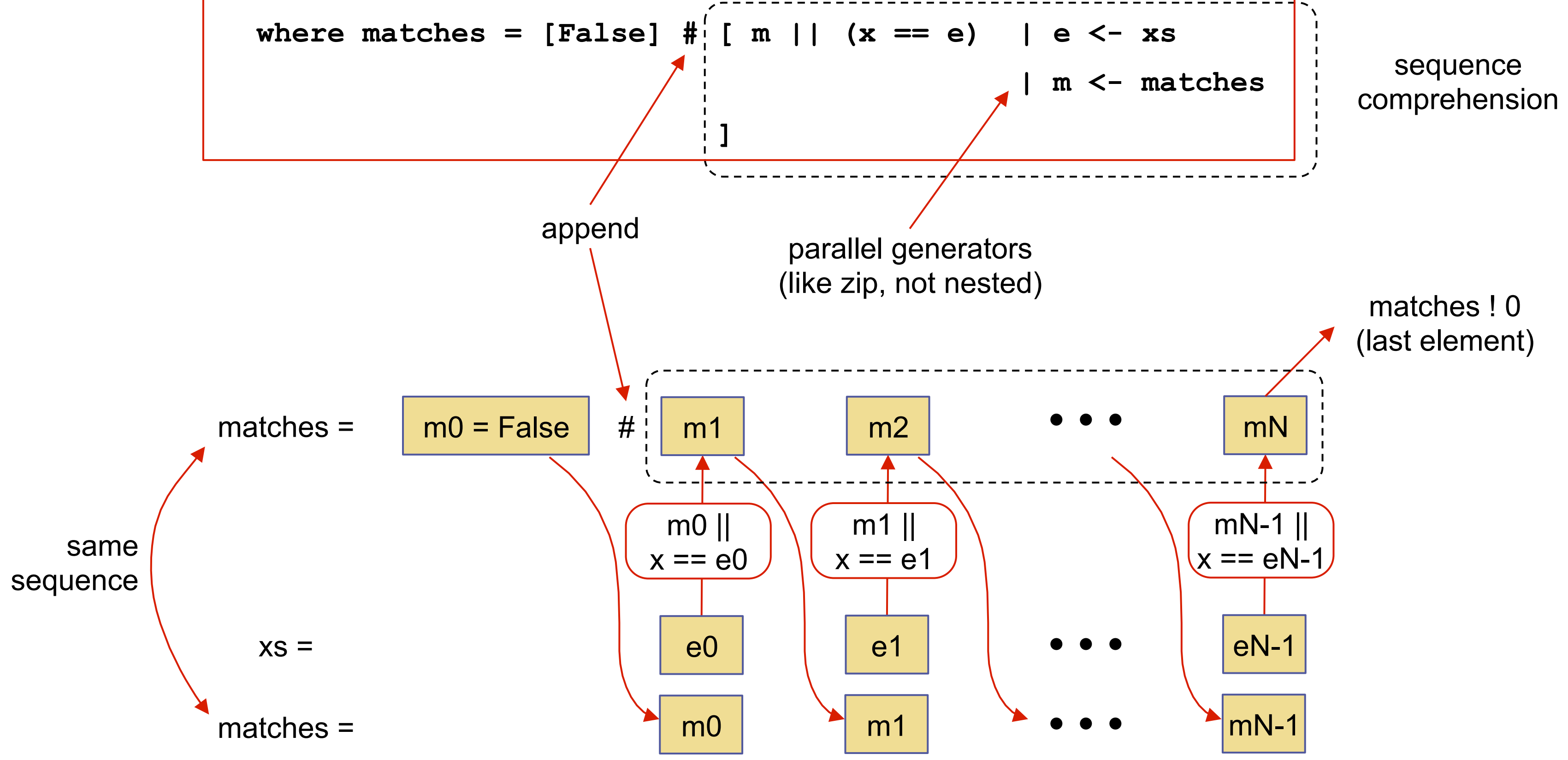
- Switch to an Emacs buffer and study the Cryptol source file Enigma.cry
    - (actually, ‘Enigma_annotated.cry’, where I’ve reorganized it a bit and added a few comments).
- Switch to a terminal window and run the Cryptol code

Cryptol code to model Enigma (file: “Enigma.cry”)

A very common idiom is the recurrence relation using sequence comprehensions, selecting the last element (essentially a fold operation).

Example:

```
// Check membership in a sequence:
elem : {a, b} (fin 0, fin a, Cmp b) => (b, [a]b) -> Bit
elem (x, xs) = matches ! 0
  where matches = [False] # [ m || (x == e)   | e <- xs
                                              | m <- matches
                            ]
```



Cryptol code to model Enigma (file: “Enigma.cry”)

Cryptol codes seem to use these comprehension-based recurrence relations for both

- folds (only interested in final folded value), and
- foldmaps (also interested in the sequence itself)

It is not clear to me why one doesn’t just use a ‘fold’ function for the former.
Example: the ‘elem’ function on the previous slide is just foldl.

Is the comprehension-based style more suitable for formal proofs?

Running the Cryptol code

(demo in terminal window)

```
$ cryptol Enigma.cry
                        _        _
   ___ _ __ _   _ _ __ | |_ ___ | |
  / __| '__| | | | '_ \| __/ _ \| |
 | (__| |  | |_| | |_) | || (_) | |
  \___|_|   \__, | .__/ \__\___/|_|
            |___/|_| version 2.1.0 (b394f1b)

Loading module Cryptol
Loading module Enigma
Enigma>
Enigma> :set ascii=on
Enigma>
Enigma> enigma (modelEnigma, "ENIGMAWASAREALLYCOOLMACHINE")
"UPEKTBSDROBVTUJGNCEHHGBXGTF"
Enigma>
Enigma> dEnigma (modelEnigma, "UPEKTBSDROBVTUJGNCEHHGBXGTF")
"ENIGMAWASAREALLYCOOLMACHINE"
Enigma>
Enigma> :q
$
```

Cryptol standard prelude

Enigma model

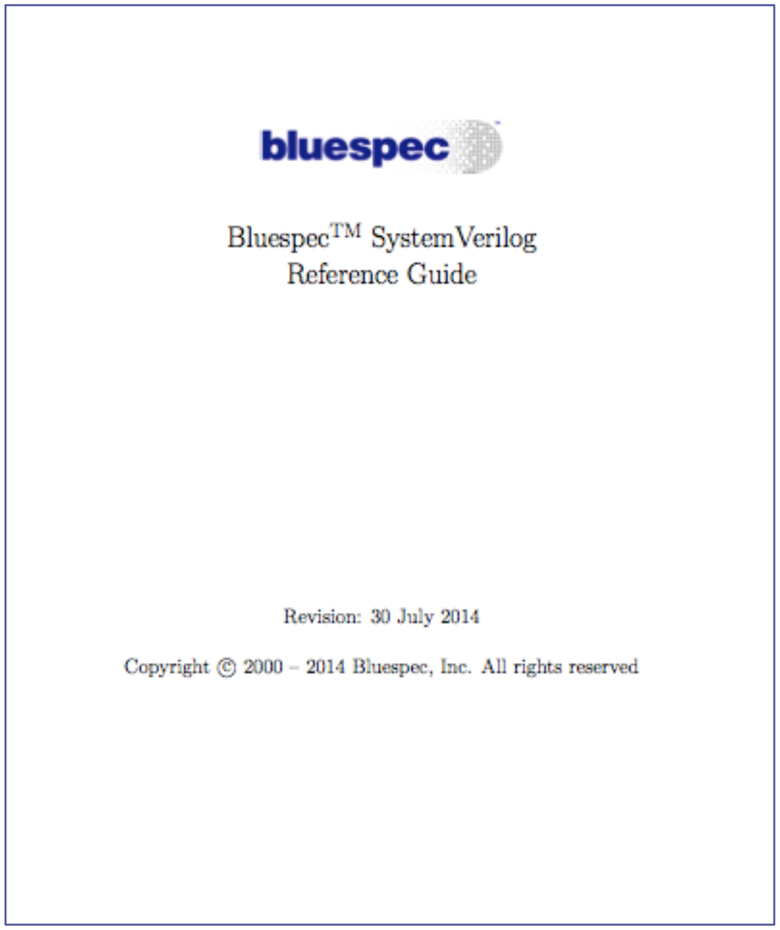
encrypt

decrypt

Bluespec BSV code for Enigma

Bluespec BSV

- Developed by Bluespec, Inc., Framingham, Massachusetts (www.bluespec.com)
- DSL for digital circuit design, from very small to very large (CPUs, complete Systems on a Chip)
- BSV language and implementation is a licensed product from Bluespec
  - Free to universities



bluespec

Bluespec^{TM} SystemVerilog
Reference Guide

Revision: 30 July 2014

Copyright © 2000 – 2014 Bluespec, Inc. All rights reserved

- "Static elaboration" (circuit structure): heavily Haskell-influenced
  - Pure, lazy functional language
  - Polymorphic types, typeclasses
  - Typeclasses for numeric types and size constraints
- Dynamic semantics (circuit behavior):
  - Guarded Atomic Actions ("atomic rules")

bluespec

Bluespec BSV

We will now:

- Switch to an Emacs buffer and study the BSV source files:

• Enigma.bsv	(transcription of Enigma.cry)
• HW_Enigma.bsv	(HW module encapsulation)
• Tb.bsv	(testbench driver)

- Switch to a terminal window and
  - Compile and run the BSV code in Bluesim (native code simulator)
  - Compile the BSV code to Verilog and run it in a Verilog simulator
  - Look at some waveforms

Note:

- BSV-generated Verilog is always "synthesizable" further to gates (using tools such as Xilinx Vivado, Altera Quartus, or Synopsys Design Compiler).
- This does not automatically mean we'll get a "good" hardware implementation (silicon area, # of gates, clock speed, latency, throughput, power, …). For that, we must also pay attention to *architecture* (e.g., pipelining) which, so far, we have not.

Cryptol recurrence relation idiom ➔ BSV vector idiom

Cryptol:

```
// Check membership in a sequence:
elem : {a, b} (fin 0, fin a, Cmp b) => (b, [a]b) -> Bit
elem (x, xs) = matches ! 0
  where matches = [False] # [ m || (x == e)   | e <- xs
                                              | m <- matches
                             ]
```

BSV:

```
// Check membership in a sequence:
function Bool elem (b x, Vector #(a, b) xs) provisos (Eq (b));
   Vector #(TAdd #(a, 1), Bool) matches;
   Integer ia = valueOf (a);
   matches [0] = False;
   for (Integer j = 1; j <= ia; j = j + 1)
      matches [j] = matches [j-1] || (x == xs [j-1]);
   return matches [ia];
endfunction
```

In BSV, loops are pure functions, just syntactic shorthand for linear recursions. This loop defines a sequence of increasingly-defined immutable vectors 'matches'.

Cryptol recurrence relation idiom ➔ BSV vector idiom

Of course, if you're just doing a fold (not a foldmap), we don't need the vector to retain the intermediate values:

BSV:

```
// Check membership in a sequence:
function Bool elem (b x, Vector #(a, b) xs) provisos (Eq (b));
   Vector #(TAdd #(a, 1), Bool) matches;
   Integer ia = valueOf (a);
   matches [0] = False;
   for (Integer j = 1; j <= ia; j = j + 1)
      matches [j] = matches [j-1] || (x == xs [j-1]);
   return matches [ia];
endfunction
```

In BSV, loops are pure functions, just syntactic shorthand for linear recursions. This loop defines a sequence of increasingly-defined immutable vectors 'matches'.

```
// Check membership in a sequence:
function Bool elem (b x, Vector #(a, b) xs) provisos (Eq (b));
   Integer ia = valueOf (a);
   match = False;
   for (Integer j = 1; j <= ia; j = j + 1)
      match = match || (x == xs [j-1]);
   return match;
endfunction
```

This loop defines a sequence of immutable values 'match'.

(Like "SSA form" in compiler IRs.)

bluespec

Cryptol recurrence relation idiom ➔ BSV vector idiom

Or, just use 'foldl/foldr/fold' from the BSV library:

BSV:

```
// Check membership in a sequence:
function Bool elem (b x, Vector #(a, b) xs) provisos (Eq (b));
   Vector #(TAdd #(a, 1), Bool) matches;
   Integer ia = valueOf (a);
   matches [0] = False;
   for (Integer j = 1; j <= ia; j = j + 1)
      matches [j] = matches [j-1] || (x == xs [j-1]);
   return matches [ia];
endfunction
```

In BSV, loops are pure functions, just syntactic shorthand for linear recursions. This loop defines a sequence of increasingly-defined immutable vectors 'matches'.

```
// Check membership in a sequence:
function Bool elem (b x, Vector #(a, b) xs) provisos (Eq (b));
   function Bool f (Bool match, b xsJ) = match || (x == xsJ);
   return foldl (f, False, xs);
endfunction
```

Just use 'foldl'

bluespec

Compiling & linking BSV for Bluesim (native simulator)

(demo in terminal window)

```
$ make compile
Compiling for Bluesim ...
bsc -u -sim -simdir build_bsim -bdir build -info-dir build -keep-fires -aggressive-
conditions -no-warn-action-shadowing -steps-warn-interval 250000 -p .:./src_BSV:%/
Prelude:%/Libraries src_BSV/Tb.bsv
checking package dependencies
compiling ./src_BSV/Enigma.bsv
compiling ./src_BSV/HW_Enigma.bsv
code generation for mkModelEnigma starts
Elaborated module file created: build/mkModelEnigma.ba
compiling src_BSV/Tb.bsv
code generation for mkTb starts
Elaborated module file created: build/mkTb.ba
All packages are up to date.
Compilation for Bluesim finished
$
```

```
$ make link
Linking Bluesim...
bsc -e mkTb -keep-fires -sim  -o ./bsim.exe  -simdir build_bsim -bdir build -info-
dir build  -p .:./src_BSV:%/Prelude:%/Libraries
Bluesim object created: build_bsim/mkTb.{h,o}
Bluesim object created: build_bsim/mkModelEnigma.{h,o}
Bluesim object created: build_bsim/model_mkTb.{h,o}
Simulation shared library created: bsim.exe.so
Simulation executable created: ./bsim.exe
Bluesim linking finished
$
```

Running Bluesim (native simulator)

(demo in terminal window)

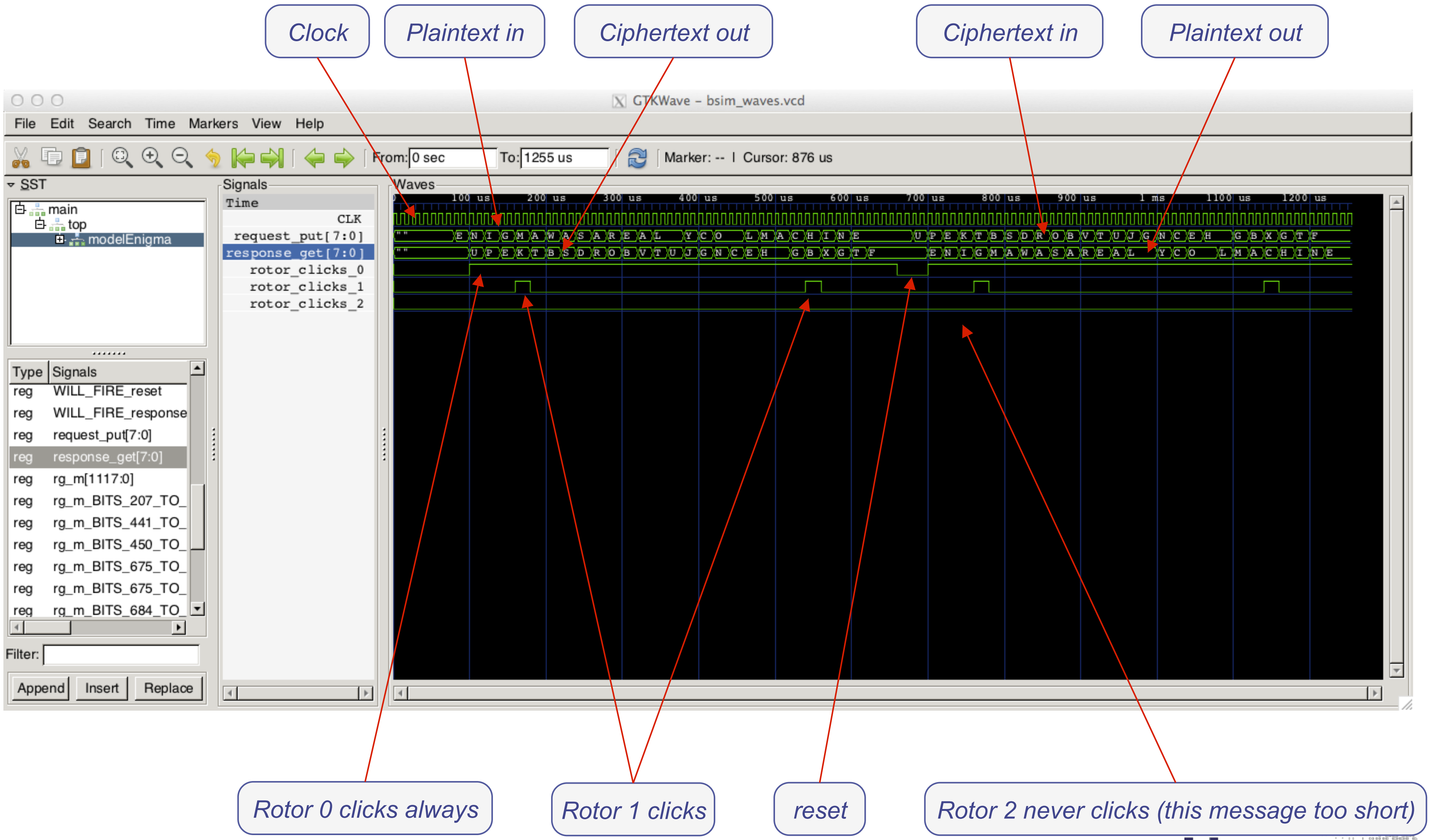
```
$ make bsim
Simulating (Bluesim) and generating VCD waveform file ...
./bsim.exe  -V bsim_waves.vcd
Plaintext input:      ENIGMAWASAREALLYCOOLMACHINE

Direct Cryptol-derived version
Cipher text output:   UPEKTBSDROBVTUJGNCEHHGBXGTF
Plaintext output:     ENIGMAWASAREALLYCOOLMACHINE

HW version (sequential input/output of text)
Cipher text output:   UPEKTBSDROBVTUJGNCEHHGBXGTF
Plaintext output:     ENIGMAWASAREALLYCOOLMACHINE
Bluesim simulation finished
$


$ ls -als bsim_waves.vcd
552 -rw-r--r--  1 nikhil  staff  282215 Feb 17 13:15 bsim_waves.vcd
$
```

Generates waveform file

Waveforms (viewed in gtkwave waveform viewer)

```
$ gtkwave bsim_waves.vcd
$
```

(Please view gtkwave_screenshot.tiff if this is too small to read)



Generating Verilog from BSV, and Verilog linking

(demo in terminal window)

```
$ make rtl
Compiling to Verilog ...
bsc -u -elab -verilog -vdir verilog  -bdir build_v  -info-dir build_v -keep-fires -
aggressive-conditions -no-warn-action-shadowing -steps-warn-interval 250000 -p .:./
src_BSV:%/Prelude:%/Libraries src_BSV/Tb.bsv
checking package dependencies
compiling ./src_BSV/Enigma.bsv
compiling ./src_BSV/HW_Enigma.bsv
code generation for mkModelEnigma starts
Verilog file created: verilog/mkModelEnigma.v
Elaborated module file created: build_v/mkModelEnigma.ba
compiling src_BSV/Tb.bsv
code generation for mkTb starts
Verilog file created: verilog/mkTb.v
Elaborated module file created: build_v/mkTb.ba
All packages are up to date.
Compilation to Verilog finished
$
```

Verilog files created

```
$ make vlink
Linking Verilog ...
bsc -e mkTb -verilog -o vsim.exe -vdir verilog -vsim iverilog -keep-fires \
                    verilog/mkTb.v
Verilog binary file created: vsim.exe
Verilog linking finished
$
```

iverilog simulator

bluespec

Running Verilog simulation

(demo in terminal window)

```
$ make vsim
Simulating Verilog and generating VCD waveform file ...
./vsim.exe  +bscvcd
VCD info: dumpfile dump.vcd opened for output.
Plaintext input:      ENIGMAWASAREALLYCOOLMACHINE

Direct Cryptol-derived version
Cipher text output:   UPEKTBSDROBVTUJGNCEHHGBXGTF
Plaintext output:     ENIGMAWASAREALLYCOOLMACHINE

HW version (sequential input/output of text)
Cipher text output:   UPEKTBSDROBVTUJGNCEHHGBXGTF
Plaintext output:     ENIGMAWASAREALLYCOOLMACHINE
Verilog simulation finished
$


$ ls -als dump.vcd
544 -rw-r--r--  1 nikhil  staff  276459 Feb 17 13:11 dump.vcd
$
```

Generates waveform file

End

Thank you!