| galois |

Solutions to the exercises

Exercise 1:

First choose a way to represent the problem. The most natural is to use a sequence of integers to represent the pigeon holes: a sequence of length `n` represents `n` pigeon holes. Each number in the sequence is the number of pigeons in the corresponding hole. To represent `n+1` pigeons sum the numbers in the sequence over all holes in the sequence.

```
npigeons : {n} (fin n) => [n][12] -> [12]
npigeons holes = z ! 0
  where
    z = [0]#[ x+y | x <- holes | y <- z ]
```

The above uses a 12 bit number to represent the total number of pigeons which is adequate. As an example, with all code in file `phf.cry`:

```
Main> :l phf.cry
Loading module Cryptol
Loading module Main
Main> :s base=10
Main> npigeons [0,1,2,0,3,1]
7
```

The following enforces the hypothesis that the capacity of all holes is one pigeon:

```
pcap : {n} (fin n) => [n][12] -> Bit
pcap holes = z ! 0
  where
    z = [True]#[ (x == 1 \/ x == 0) /\ y | x <- holes | y <- z ]
```

The following property will prove if it is not possible to distribute pigeons one per hole:

```
some_hole_overloaded: [2048][12] -> Bit
property some_hole_overloaded holes =
   if npigeons holes > length holes then ~(pcap holes) else True
```

This property proves the hypothesis for `2048` holes. This is notable because it is well known that SAT solvers are particularly bad at proving this property for CNF representations of this problem which is one of the reasons this is considered a benchmark problem. The situation for CNF improves with extended resolution but the solution is difficult for most to look at because the CNF formula is written to express a proof by induction.

```
Main> :l phf.cry
Loading module Cryptol
Loading module Main
Main> :s prover=z3
Main> :prove some_hole_overloaded
Q.E.D.
(Total Elapsed Time: 4.102s, using "Z3")
```

Alternatively,

```
Main> :l phf.cry
Loading module Cryptol
Loading module Main
Main> :sat ~some_hole_overloaded
Unsatisfiable
(Total Elapsed Time: 3.956s, using "Z3")
```

Exercise 2:

For fm1 returns a member of 100 element lst:

```
fm1_returns_member : [100][32] -> Bit
property fm1_returns_member lst = member (fm1 lst) lst
```

For fm2 returns a member of 100 element lst:

```
fm2_returns_member : [100][32] -> Bit
property fm2_returns_member lst = member (fm2 lst) lst
```

For fm1 returns the same number as fm2 for 100 element lst:

```
fm1_same_as_fm2 : [100][32] -> Bit
property fm1_same_as_fm2 lst = fm1 lst == fm2 lst
```

Proofs (above code in file `findmax.cry`):

```
Main> :l findmax.cry
Loading module Cryptol
Loading module Main
Main> :s prover=cvc4
Main> :prove fm1_returns_member
Q.E.D.
(Total Elapsed Time: 0.084s, using "CVC4")
Main> :prove fm2_returns_member
Q.E.D.
(Total Elapsed Time: 0.088s, using "CVC4")
Main> :s prover=z3
Main> :prove fm1_same_as_fm2
Q.E.D.
(Total Elapsed Time: 0.222s, using "Z3")
```

Observe `fm1` does not always return the same number as `fm2` since `fm1 []` `is [0]` `and fm2 []` is not allowed.

Exercise 3:

```
adder_netlist U V W = (xpos X, xpos Y)
  where
    u = xneg U;
    v = xneg V;
    w = xneg W
    a = ~u
    h = ~w
    f = u /\ v
    c = a /\ v
    b = ~v
    d = b /\ u
    e = c \/ d
    g = ~e
    l = e /\ w
    i = g /\ w
    j = h /\ e
    X = i \/ j
    Y = f \/ l

adder_spec U V W = (xpos (u ^ v ^ w), xpos ((u /\ v) \/ (u /\ w) \/ (v /\ w)))
  where
    u = xneg U;
    v = xneg V;
    w = xneg W
```

```
netlistOK : Bit -> Bit -> Bit -> Bit
property netlistOK U V W = adder_netlist U V W == adder_spec U V W
```

Proves like this:

```
Main> :l adder.cry
Loading module Cryptol
Loading module Main
Main> :prove netlistOK
Q.E.D.
(Total Elapsed Time: 0.034s, using "Z3")
```

Exercise 4:

```
xpos d = if d == True then 1 else 0
xneg d = if d == 1 then True else False

// lst is a sequence of {0,1} representing a number in binary.
// Output of xlate is the decimal number
xlate lst = z ! 0
  where
    z = [0]#[ 2*x + y  | x <- z | y <- lst ]

circuit a4 a3 a2 a1 b4 b3 b2 b1 c = out
  where
    (o1,c1) = adder_netlist a1 b1 c;
    (o2,c2) = adder_netlist a2 b2 c1;
    (o3,c3) = adder_netlist a3 b3 c2;
    (o4,c4) = adder_netlist a4 b4 c3;
    out = xlate [xpos o4,xpos o3,xpos o2,xpos o1]

spec a4 a3 a2 a1 b4 b3 b2 b1 c =
   (xlate [a4, a3, a2, a1] + xlate [b4, b3, b2, b1] + xlate [c]) % 16

addWorks : Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> Bit
property addWorks A4 A3 A2 A1 B4 B3 B2 B1 C =
    circuit a4 a3 a2 a1 b4 b3 b2 b1 c == spec a4 a3 a2 a1 b4 b3 b2 b1 c
  where
    a4 = xpos A4;
    a3 = xpos A3;
    a2 = xpos A2;
    a1 = xpos A1;
    b4 = xpos B4;
    b3 = xpos B3;
    b2 = xpos B2;
    b1 = xpos B1;
     c = xpos C
```

Note: most of this is Boolean logic but the specification of `adder` is numeric. In order to mix logic with numbers the functions `xneg` and `xpos` are provided.

Proves like this:

```
Main> :l adder.cry
Loading module Cryptol
Loading module Main
Main> :prove addWorks
Q.E.D.
(Total Elapsed Time: 0.131s, using "Z3")
```