| galois |

Data Types

Consider data types in the C language. Here is an example related to SHA 512:

```
struct sha512_state_st {
   uint64_t h[8];
   uint64_t Nl, Nh;
   uint8_t p[128];
   unsigned num, md_len;
};
```

This structure consists of an 8 element 64 bit unsigned integer array named `h`, 64 bit unsigned integer numbers `Nl` and `Nh`, a 128 element byte array named `p` and 32 bit unsigned integers `num` and `md_len`. Creation and initialization of a variable of this type might look like this:

```
SHA512_CTX stateInit = {
         {1,2,3,4,5,6,7,8},
         23, 44,
         {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
         66, 43 };
```

A simple function to manipulate a SHA512 state and return a fresh copy might look like this (please forgive the lack of protections – this is just for illustration):

```
SHA512_CTX *stateManip (SHA512_CTX state) {
   int i;
   SHA512_CTX *s = (SHA512_CTX*)malloc(sizeof(SHA512_CTX));
   for (i=0 ; i < 8 ; i++) s->h[i] = state.h[i];
   s->Nl = state.Nl;
   s->Nh = state.Nh;
   for (i=0 ; i < 128 ; i++) s->p[i] = state.p[i];
   s->num = state.num;
   s->md_len = 12;
   return s;
}
```

A function to print a SHA512 state might look like this:

```
void printState (SHA512_CTX state) {
   int i;
   printf("h=");
   for (i=0 ; i < 8 ; i++) printf("%ld,",state.h[i]);
   printf("\nNl=%ld, Nh=%ld\n", state.Nl, state.Nh);
   printf("p=");
   for (i=0 ; i < 128 ; i++) printf("%d,",state.p[i]);
```

```
   printf("\nnum=%d, md_len=%d\n", state.num, state.md_len);
}
```

Cryptol versions of these would be the following. For the data structure `SHAState`:

```
type SHAState = { h : [8][64],
                  block : [128][8],
                  Nl : [64],
                  Nh : [64],
                  num : [32],
                  md_len : [32]
                }
```

For an initialization of a variable of type `SHAState`:

```
stateInit : SHAState
stateInit = { h = [1,2,3,4,5,6,7,8],
              block = zero,
              Nl = 11,
              Nh = 16,
              num = 1,
              md_len = 128 }
```

A function that manipulates a state might look like this:

```
stateManip : SHAState -> SHAState
stateManip s = { h = s.h,
                 block = s.block,
                 Nl = s.Nl,
                 Nh = s.Nh,
                 num = s.num,
                 md_len = 42 }
```

Observe that in later lessons there will be attempts to show that C function are functionally equivalent to Cryptol specifications. But Cryptol has no pointers to manipuate as C does. Hence something needs to be added to make that happen.

Cryptol allows parameterization of types. This is done by creating a section called parameter and then declaring type parameters with constraints. For example:

```
parameter
   type w : #
   type constraint (fin w, w <= 64, 32 <= w)
H : [2][w]
H = [100,101]
```

results in a variable H which is an array of size 2 with width between 32 and 64. Note that

```
v1 : [2][32]
```

and

```
type v1 = [2][32]
```

express the same type.

Example:
Consider the Chicken, Fox, Grain riddle. A farmer has to get a fox, a chicken, and a sack of corn across a river, all intact. To do this the farmer has only a rowboat that is big enough to carry just the farmer and one other thing. If the fox and the chicken are left together, the fox

will eat the chicken. If the chicken and the corn are left together, the chicken will eat the corn. The fox does not like corn. What sequence of river crossings will allow the farmer to arrive at the opposite bank without anything eating anything?

Begin setting up a Cryptol solution by considering problem state. There is a left bank and a right bank. Use a field of 4 bits to identify who or what, called a player, is on each bank. Let the farmer's status be represented by the least significant bit, the chicken's status by the 2nd bit, the corn's status by the 3rd bit, and the fox's status but the most significant bit. If a player is on the left bank then it's bit is 1 on the 4 bit field representing the left bank and it's bit is 0 on the 4 bit field representing the right bank and vice versa. So, in Cryptol, one can declare a type such as the following to represent bank state:

```
type OneBank = [4]
```

For the state of the problem, two banks can be combined like this:

```
type BankState = { left : OneBank, right : OneBank }
```

Say the farmer starts on the left bank. The the initial state is

```
startState = { left = 0b1111, right = 0b0000 }
```

the end state is

```
endState = { left = 0b0000, right = 0b1111 }
```

A `OneBank` state is not *safe* if it is `0b1010`, or `0b0110`. One of those states means something has eaten something on the corresponding bank. The following function `safeAndValidState` returns `True` given `BankState s` if and only if both left and right `OneBank` states of `s` are safe and the BankState is valid which means a player with bit of value 1 on one bank has value 0 on the opposite bank.

```
safeAndValidState : BankState -> Bit
safeAndValidState s = ~(s.left == 10 \/ s.left == 6 \/ s.right == 10 \/
                        s.right == 6) /\ s.left ^ s.right == 0xF
```

The next function takes as input a (safe and valid) state `s` plus a position (0,1,2) `idx` and outputs a possible next state. If `s.left` is even then `s.right` is odd and represents the move to opposite bank. Then the next state `left` will be `s.left || s.right` masked with `0b0011` if `idx` = 0, `0b0101` if `idx` = 1 and `0b1001` if `idx` = 2. The next state `right` will be `s.right && 0b0001 && 0b1100` if `idx` = 0, `0b1010` if `idx` = 1, and `0b0110` if `idx` = 2. Similarly for the case that `s.left` is odd and `s.right` is even. Here is the function:

```
nextState : BankState -> [8] -> BankState
nextState s idx =
   if s.right % 2 == 1 then
       { left = s.left || s.right && (0b0001 || (1 << (idx+1))),
         right = s.right && 0b1110 && (~ (1 << (idx+1):[4])) }
   else
       { left = s.left && 0b1110 && (~ (1 << (idx+1):[4])),
         right = s.right || s.left && (0b0001 || (1 << (idx+1))) }
```

For example:

```
Main> let t = { left=0b1001, right=0b0110 }
Main> nextState t 0
{left = 0b1000, right = 0b0111}
Main> nextState t 1
```

```
{left = 0b1000, right = 0b0111}
Main> nextState t 2
{left = 0b0000, right = 0b1111}
```

If `states` represents the sequence of `BankStates` that is the solution, there are a few checks that must be made on `states`. First, all successive `BankStates` in the sequence `states` must be a legitimate `nextState` of the previous `BankState`. That check is done by the following:

```
neighborsConsistent : {n} (fin n) => [1 + n]BankState -> Bit
neighborsConsistent states = z ! 0
   where
   z = [True]#[ ((y == nextState x 0) \/
                 (y == nextState x 1) \/
                 (y == nextState x 2)) /\ p
              | x <- states | y <- drop `{1} states | p <- z]
```

Second, all states must be safe and valid. That may be taken care of by the following

```
allStatesSafeAndValid : {n} (fin n) => [n]BankState -> Bit
allStatesSafeAndValid states = z ! 0
   where
      z = [True]#[ safeAndValidState y /\ x | y <- states | x <- z ]
```

The first `BankState` should be the `startState` and the last `BankState` should be the `endState`. These may be encoded like this:

```
(states ! 0 == endState) /\ (states @ 0 == startState)
```

All `BankStates` in the solution sequence should be unique. The following can be used for that:

```
allStatesUnique states = ~(z ! 0)
   where
      z = [False]#[ (states@i == states@j /\ ~(i == j)) \/ k
                  | k <- z | i <- [0..7] , j <- [0..7] ]
```

The 7 shows up in the above because a minimum solution contains 8 `BankStates` (a well known number). Putting all of the above together gives:

```
solution : [8]BankState -> Bit
property solution states =
   (neighborsConsistent states) /\
   (allStatesSafeAndValid states) /\
   (allStatesUnique states) /\
   (states ! 0 == endState) /\
   (states @ 0 == startState)
```

Notice the number 8 again indicating the expected solution size. Run it like this:

```
Main> :s base=2
Main> :s satNum=all  // print all solutions
Main> :sat solution
Satisfiable
solution
  [{left = 0b1111, right = 0b0000}, {left = 0b1100, right = 0b0011},
   {left = 0b1101, right = 0b0010}, {left = 0b1000, right = 0b0111},
   {left = 0b1011, right = 0b0100}, {left = 0b0010, right = 0b1101},
   {left = 0b0011, right = 0b1100}, {left = 0b0000, right = 0b1111}]
  = True
```

```
solution
  [{left = 0b1111, right = 0b0000}, {left = 0b1100, right = 0b0011},
   {left = 0b1101, right = 0b0010}, {left = 0b0100, right = 0b1011},
   {left = 0b0111, right = 0b1000}, {left = 0b0010, right = 0b1101},
   {left = 0b0011, right = 0b1100}, {left = 0b0000, right = 0b1111}]
  = True
Models found: 2
(Total Elapsed Time: 0.108s, using "Z3")
```

The use of list comprehensions as seen above in `allStatesNotUnique`, `allStatesSafeAndValid`, and `neighborsConsistent` will be discussed with plenty of examples in the next lesson.