Cryptol: A Comprehensive Guide

- Mastering the Art of Cryptol Programming -

Ji Yong-Hyeon

Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

March 27, 2024

Cryptol vs EasyCrypt

- **Cryptol**: Cryptol is a domain-specific language designed specifically for specifying cryptographic algorithms. A creation by Galois, Inc., it's a tool used primarily for creating high-assurance cryptographic software. Cryptol allows developers to write cryptographic algorithms in a way that directly reflects the mathematical specifications, which makes it easier to analyze and verify for correctness and security.

- **EasyCrypt**: On the other hand, EasyCrypt is a toolset designed for the formal verification of cryptographic proofs. It provides a framework for developing and verifying mathematical proofs of the security of cryptographic constructions, such as encryption schemes, signature schemes, and hash functions. EasyCrypt operates at a higher level of abstraction compared to Cryptol and is used for proving the security properties of cryptographic protocols mathematically.

If you're comparing them from a user perspective, Cryptol is more about the implementation and specification of cryptographic algorithms, making sure they are implemented correctly according to their mathematical definitions. EasyCrypt is more about proving the theoretical security properties of cryptographic protocols and systems.

Contents

Chapter 1

A Crash Course in Cryptol

1.1 Basic Data Types

1.1.1 Bit: Booleans

"The most basic data type in Cryptol, representing a single binary digit (0 or 1)."

Basic Bit Operations: True or False

Data Type: `Bit`

In Cryptol, a '`Bit`' can either be '`True`' or '`False`'.

```
a : Bit
a = True

b : Bit
b = False
```

```
$ Main> a
True
$ Main> b
False
```

Similar to C, Cryptol supports logical operations like **AND**, **OR**, and **NOT**.

```
result_and : Bit
result_and = a && b

result_or : Bit
result_or = a || b

result_not : Bit
result_not = ~a
```

```
$ Main> result_and
False
$ Main> result_or
True
$ Main> result_not
False
```

Intermediate Bit Operations

Conditional Expressions: Cryptol uses the '**if...then...else**' syntax, which is similar to the ternary operator '**? :** ' in C.

```
a : Bit
a = True

b : Bit
b = False

// Conditional expression
result_cond : Bit
result_cond = if a then b else ~b
```

```
$ Main> result_cond
False
```

XOR Operation: This is commonly used in cryptography for things like one-time pads.

```
a : Bit
a = True

c : Bit
c = True

result_xor : Bit
result_xor = a ^ c
```

```
$ Main> result_xor
False
```

Advanced Cryptographic Applications with Bit

Simple Cryptographic Operation: Here's a very basic example of a cryptographic operation in Cryptol, a one-time pad encryption of a single bit. In practical applications, you'll work with sequences or arrays of bits.

```
// One-time pad encryption
plaintext_bit : Bit
plaintext_bit = True // Your plaintext

key_bit : Bit
key_bit = False // Your secret key

// XOR for encryption
encrypted_bit : Bit
encrypted_bit = plaintext_bit ^ key_bit

// XOR for decryption (should equal plaintext_bit)
decrypted_bit : Bit
decrypted_bit = encrypted_bit ^ key_bit
```

```
$ Main> encrypted_bit
True
$ Main> decrypted_bit
True
```

```
module XOR where

// Defines an XOR function for two bits
xor : Bit -> Bit -> Bit
xor a b = (a || b) && ~(a && b)

// Property: XOR is its own inverse
property xor_inverse a b = xor (xor a b) b == a
```

```
$ cryptol XOR.cry
version 2.8.0

Loading module Cryptol
Loading module XOR
$ XOR> :prove xor_inverse
Q.E.D.
(Total Elapsed Time: 0.008s, using Z3)
```

1.1.2 Sequences: Homogeneous Collections

"Fixed-length, ordered collections of elements, which can be bits or other types."

Basic Sequence Operations

Sequence

In Cryptol, a **sequence** is defined with square brackets. `[, , ,]`.

Remark 1.1. Unlike C, where you typically define the array size and type, Cryptol infers these from the context.

```
seq : [4]Bit // A sequence of 4 bits
seq = [True, False, True, False] // Equivalent to 0b1010
```

```
$ Main> seq
0xa
```

Accessing Elements: Accessing sequence elements is similar to accessing array elements in C.

```
element1 : Bit
element1 = seq @ 0 // Accessing the first element, similar to seq[0] in C

element3 : Bit
element3 = seq @ 3
```

```
$ Main> element1
True
$ Main> element3
False
```

Intermediate Sequence Operations

Sequence Concatenation: You can concatenate sequences to form longer sequences.

```
seq1 : [3]Bit
seq1 = [True, True, False] // Similar to {1, 1, 0} in C

seq2 : [5]Bit
seq2 = [False, True, True, False, True] // Similar to {0, 1, 1, 0, 1} in C

// Concatenating two sequences 0b 110 01101 = 0b 1100 1101
combined_seq : [8]Bit
combined_seq = seq1 # seq2
```

```
$ Main> seq1
0x6
$ Main> seq2
0x0d
$ Main> combined_seq
0xcd
```

```
seq1 : [3][4]
seq1 = [0xf, 0x1, 0x8]

seq2 : [5][8]
seq2 = [0xff, 0xcc, 0x11, 0x22, 0x99]

ele1 : [4]
ele1 = seq1 @ 1

ele2 : [8]
ele2 = seq2 @ 3
```

```
$ Main> seq1
[0xf, 0x1, 0x8]
$ Main> ele1
0x1
$Main> seq2
[0xff, 0xcc, 0x11, 0x22, 0x99]
$ Main> ele2
0x22
```

Advanced Cryptographic Applications with Sequence

Block Operations: In cryptography, you often work with blocks of data. In Cryptol, you can easily split sequences into blocks or combine blocks into a single sequence.

```
// Splitting a sequence into blocks
long_seq : [16]Bit // Similar to a block of data in cryptography
long_seq = 0xAAAA

blocks : [2][8]Bit // Splitting into two blocks of 8 bits
blocks = split long_seq // There's no direct C equivalent

joined_seq : [16]Bit
joined_seq = join blocks
```

```
$ Main> long_seq
0xaaaa
$ Main> blocks
[0xaa, 0xaa]
$ Main> joined_seq
0xaaaa
```

Applying Functions to Sequences: In Cryptol, you can apply functions to entire sequences, which is useful for cryptographic transformations.

```
// A simple bitwise NOT operation applied to a sequence
negate_seq : [8]Bit -> [8]Bit
negate_seq input = ~input // Bitwise NOT each element

input_seq : [8]Bit
input_seq = 0xCD // ~ 0b11001101 = 0b00110010 = 0x32

negated_seq : [8]Bit
negated_seq = negate_seq input_seq
```

```
$ Main> negate_seq
<function>
$ Main> input_seq
0xcd
$ Main> negated_seq
0x32
```

Advanced Cryptographic Patterns: Sequences are used in various cryptographic algorithms, such as block ciphers and hash functions, to handle data in structured formats.

```
// Example of XORing two sequences (common in many cryptographic algorithms
   )
xor_sequences : [8]Bit -> [8]Bit -> [8]Bit
xor_sequences seq1 seq2 = zipWith (^) seq1 seq2 // Element-wise XOR

seqA : [8]Bit
seqA = 0xAA

seqB : [8]Bit
seqB = 0x55

//   0b 1010 1010
// ^ 0b 0101 0101
//   0b 1111 1111
result_seq : [8]Bit
result_seq = xor_sequences seqA seqB // Result of XORing seqA and seqB
```

```
$ Main> seqA
0xaa
$ Main> seqB
0x55
$ Main> result_seq
0xff
```

Note ('zipWith' in Haskell).

```
// [f(x1, y1), f(x2, y2), ..., f(x_n, y_n)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f list1 list2
```

1.1.3 Tuples: Heterogeneous Collections

"Collections of possibly different types grouped together."

Basic Tuple Operations

Tuple

A **tuple** can hold a fixed number of elements, which can be of different types.

Remark 1.2. Unlike C, where you define a struct with named fields, Cryptol tuples are indexed by position.

```
// A tuple containing a Bit, a 3-Bit sequence, and a 4-Bit sequence
myTuple : (Bit, [3]Bit, [4]Bit)
myTuple = (True, [False, True, False], [True, True, False, False])
```

```
$ Main> myTuple
(True, 0x2, 0xc)
```

Accessing Tuple Elements: Elements in a tuple are accessed by their positions, starting from zero.

```
first_element : Bit
first_element = myTuple.0 // Accessing the first element of the tuple

second_element : [3]Bit
second_element = myTuple.1 // Accessing the second element of the tuple
```

```
$ Main> first_element
True
$ Main> second_element
0x2
```

Updating Tuple Elements: Unlike in C, where you might update an element directly, in Cryptol, you usually create a new tuple with the updated value due to its immutable nature.

```
// Updating the second element
updatedTuple : (Bit, [3]Bit, [4]Bit)
updatedTuple = (myTuple.0, [True, True, True], myTuple.2)
```

```
$ Main> updatedTuple
(True, 0x7, 0xc)
```

Intermediate Tuple Operations

Tuple Decomposition: You can decompose a tuple into its components, similar to unpacking a struct in C.

```
// Decomposing the tuple into variables
(bitVal, threeBitSeq, fourBitSeq) = myTuple
```

```
$ Main> bitVal
True
$ Main> threeBitSeq
0x2
$ Main> fourBitSeq
0xc
```

Nested Tuples: Tuples can contain other tuples, providing a way to structure complex data.

```
// A tuple where the first element is also a tuple
nestedTuple : ((Bit, Bit), [4]Bit)
nestedTuple = ((True, False), [True, True, False, False])
```

```
$ Main> nestedTuple
((True, False), 0xc)
```

Advanced Cryptographic Applications with Tuple

Using Tuples in Cryptographic Algorithms: Tuples can represent structured data in cryptographic algorithms, like keys and states.

```
// An example with a simple encryption scheme
type Key = [8]Bit
type State = [8]Bit
type CipherText = [8]Bit

encrypt : (Key, State) -> CipherText
encrypt (key, state) = zipWith (^) key state // A simple XOR-based
   encryption

property EncryptDecrypt =
  \(key : Key) (state : State) ->
    (encrypt(key, encrypt(key, state))) == state
```

```
$ Main> :prove
:prove EncryptDecrypt
        Q.E.D.
(Total Elapsed Time: 0.012s, using Z3)
```

Advanced Data Structuring: In more complex cryptographic algorithms, you might use tuples to represent different parts of the algorithm's state, input, or output.

```
// An example using tuples for algorithm state management
type RoundKey = [8]Bit;
type RoundState = ([8]Bit, [8]Bit); // Tuple representing state for two
   rounds

// Update both states
updateState : (RoundKey, RoundState) -> RoundState;
updateState (key, (state1, state2)) =
  (zipWith (^) key state1, zipWith (^) key state2);

property DoubleUpdate =
  \(key : RoundKey) (initialState : RoundState) ->
    updateState(key, updateState(key, initialState)) == initialState
```

```
$ Main> :prove
:prove DoubleUpdate
        Q.E.D.
(Total Elapsed Time: 0.013s, using Z3)
```

1.1.4 Records: Named Collections

"Similar to tuples, but each element is identified by a name."

Basic Record Operations

Record

We define a record with braces '{}' and specify the names and types of its fields.

```
type MyRecord = {
  field1 : Bit,
  field2 : [3]Bit,
  field3 : [4]Bit
}
myRecord : MyRecord
myRecord = {
  field1 = True,
  field2 = [False, True, False],
  field3 = [True, True, False, False]
}
first_field : Bit
first_field = myRecord.field1 // Accessing the first field of the record
second_field : [3]Bit
second_field = myRecord.field2 // Accessing the second field of the record
```

```
$ Main> myRecord
{field1 = True, field2 = 0x2, field3 = 0xc}
$ Main> first_field
True
$ Main> second_field
0x2
```

Intermediate Record Operations

Nested Records: Records can contain other records, which allows you to structure complex data hierarchically.

```
type NestedRecord = {nested : MyRecord, anotherField : [5]Bit}
nestedRecord : NestedRecord
nestedRecord = {
  nested = {
    field1 = False,
    field2 = [True, False, True],
    field3 = [False, False, False, True]
  },
  anotherField = [True, True, True, False, False]
}
```

```
$ Main> nestedRecord
{nested = {field1 = False, field2 = 0x5, field3 = 0x1},
        anotherField = 0x1c}
```

Advanced Cryptographic Applications with Record

Cryptographic Keys and Configurations: Records are excellent for representing complex keys and configuration options in cryptographic algorithms.

```
// A record type for encryption configuration
type CryptoConfig = {key : [128], iv : [128]}

// An example configuration using all zeros (False)
defaultConfig : CryptoConfig
defaultConfig = {key = zero, iv = ~zero}
```

```
$ Main> defaultConfig
{key = 0x00000000000000000000000000000000,
 iv = 0xffffffffffffffffffffffffffffffff}
```

State Management in Cryptographic Algorithms: Use records to manage the state of an algorithm, making it easier to handle multiple pieces of related data.

```
type AlgorithmState = {counter : [32], buffer : [64], flags : [8]} // A
   record type for the algorithm state

initialState : AlgorithmState
initialState = {counter = zero, buffer = zero, flags = zero} // Initializes
    all to zero

updateState : AlgorithmState -> AlgorithmState
updateState state =  { state |
  counter = state.counter + 1,
  buffer = ~state.buffer,
  flags = zipWith (^) state.flags 0xf0
}
```

```
property CounterIncrement state =
  (updateState state).counter == state.counter + 1

property BufferToggle state =
  (updateState (updateState state)).buffer == state.buffer

property FlagsXOR state =
  (updateState state).flags == zipWith (^) state.flags 0xf0
```

```
$ Main> :prove
:prove CounterIncrement
Q.E.D.
(Total Elapsed Time: 0.013s, using Z3)
:prove BufferToggle
Q.E.D.
(Total Elapsed Time: 0.008s, using Z3)
:prove FlagsXOR
Q.E.D.
(Total Elapsed Time: 0.009s, using Z3)
```

1.1.5 Function

"Types representing functions from inputs to outputs."

Basic Function Usage

Simple Function: Let's start with a basic function that adds two numbers. In Cryptol, the syntax for function definition is concise.

```
add : Integer -> Integer -> Integer;
add x y = x + y;
```

```
$ Main> add 4294967296 4294967296
8589934592
```

Function with Bitwise Operation: Functions in Cryptol commonly operate on bits and sequences of bits. Here's a function that performs a bitwise AND on two 8-bit sequences.

```
andBits : [8] -> [8] -> [8];
andBits a b = zipWith (&&) a b;
```

```
$ Main> andBits 0x0f 0xf0
0x00
```

Intermediate Function Usage

Functions with Conditional Logic: Cryptol functions can implement more complex logic, including conditionals.

```
max : Integer -> Integer -> Integer
max x y = if x > y then x else y
```

```
$ Main> max 51 94
94
```

```
property max_ge_x (x : Integer, y : Integer) = max x y >= x

property max_ge_y (x : Integer, y : Integer) = max x y >= y

property max_is_either (x : Integer, y : Integer) =
  (max x y == x) || (max x y == y)
```

Recursive Functions: Cryptol supports recursive functions, which can be powerful for certain types of algorithms.

```
factorial : Integer -> Integer;
factorial n = if n == 0 then 1 else n * factorial (n - 1);
```

```
Main> factorial 10
3628800
```

Advanced Cryptographic Function Usage

Encrypting with XOR (One-Time Pad): Here's how you might define a simple encryption function using XOR, which is a foundational operation in many cryptographic systems..

```
xorEncrypt : [8] -> [8] -> [8];
xorEncrypt key plaintext = zipWith (^) plaintext key;

property Reversibility (key : [8], plaintext : [8]) =
  xorEncrypt key (xorEncrypt key plaintext) == plaintext
```

```
$ Main> :prove
:prove Reversibility
       Q.E.D.
(Total Elapsed Time: 0.012s, using Z3)
```

1.1.6 Type Variables

Basic Usage of Type Variables

Generic Identity Function: A simple example of a type variable in use is the identity function, which returns whatever value is passed to it, regardless of type.

```
id : {a} a -> a;
id x = x;
```

```
$ Main> id 0xff
0xff
$ Main> id 0xffff
0xffff
```

Generic Swap Function: This function swaps the elements of a 2-tuple, and works for any types contained in the tuple.

```
swap : {a, b} (a, b) -> (b, a)
swap (x, y) = (y, x)

mixedPair : (Integer, [8])
mixedPair = (42, 0xaa)

swappedMixedPair : ([8], Integer)
swappedMixedPair = swap mixedPair
```

```
$ Main> mixedPair
(42, 0xaa)
$ Main> swappedMixedPairnewp
(0xaa, 42)
```

Intermediate Usage of Type Variables

Generic Arrays: You can define functions that operate on arrays of any type. This is useful for functions that perform operations like reversing an array.

```
id : {a} a -> a
id x = x
```

```
$ Main> id 0xff
0xff
$ Main> id 0xffff
0xffff
```