Learning Cryptol: the Basics

Iavor S. Diatchki

Goal

- At the end of this tutorial you should have gained:
  - A basic understanding of the structure of the Cryptol language
  - Experience of creating and modifying programs in Cryptol
  - Experience with using the Cryptol REPL

Cryptol REPL Basics

- Typing an expression will evaluate it
- REPL commands all start with a colon
  - example: `:help`
- Any command may be abbreviated
  - example: type `:h` instead of `:help`
- Pressing `TAB` will complete a command, or show alternatives

`:help`	show a list of commands
`:quit`	leave the REPL

Loading a File

`:load AES.cry`	load a file
`:module AES`	load a module (AES.cry)
`:reload`	reload currently loaded file

Query the REPL

`:browse`	show definitions in scope
`:type 1+2`	show the type of an expression
`:set`	show the REPL settings
`:set ascii=on`	change the value of a REPL setting

Exercises

- Start a Cryptol REPL
- What is the type of `True`?
- Evaluate the expression: `0x2 + 0b1100`
- What is the value of the `base` setting?
- Set `base` to 2
- Use "up-arrow" to evaluate `0x2 + 0b1100` again.
- Type: `:help toInteger`
- Type: `:help /`

Bits and Words

- Booleans are of type `Bit`
  - `True, False`
- n-bit words have type `[n]`:
  - `16, 0xF0, 0b00001111, 0o20`
- Unbounded mathematical integers have type `Integer`:
  - `16`

Literal Overloading

- Decimal literals are overloaded:
  - `16 : [8]`
  - `16 : [32]`
  - `16 : Integer`
- Other literals have a fixed type:
  - `0b1 : [1]`
  - `0o7 : [3]`
  - `0xF : [4]`
  - `'a' : [8]`

Standard Operations

- Logical operators:
  - conjunction, disjunction, xor, negation
  - `&& || ^ ~`
- Arithmetic operators:
  - add, subtract, multiply, divide, modulus, exponentiate
  - `+ - * / % ^^`
- Comparisons:
  - the result is a `Bit`
  - `== != < <= > >=`
- Conditional expressions:
  - expression-level *if-then-else*
  - `if (x % 2) == 0 then 0 else 1`

Exercises

- Chapter 1, exercise: 1.1–1.4

Sequences

- Homogeneous *sequences* are written in brackets:
  - `[ False, True, True, False ]`
  - `[ [True,False], [False,False] ]`
  - Sequence type `[n]a`: `n` elements, each of type `a`
- *Words* are simply sequences of bits:
  - big-endian: `0b110 == [True,True,False]`
  - `[n]Bit`, usually abbreviated to `[n]`
- Quoted strings are simply sequences of 8-bit words:
  - `"abc" == [ 0x61, 0x62, 0x63 ]`

Tuples and Records

- Heterogeneous data may be grouped in *tuples*:
  - `(13, "hello", True)`
- *Records* are like tuples but with named fields:
  - `{ x = 0x08, y = 0xFFFF }`

Accessing Elements

- Accessing sequence elements:
  - 0-indexed, from the front: `[1,2,3] @ 0 == 1`
  - 0-indexed, from the back: `[1,2,3] ! 0 == 3`
- To access fields of tuples and records use `.`:
  - `('a', 0xFF).0 == 'a'`
  - `{ x = 'a', y = 0xFF }.x == 'a'`

Lifting Operations

- Many operations on the basic types also work on structured types pointwise:
  - `[1,2] + [3,4] == [4,6]`
- Comparisons use lexicographic ordering:
  - Sequences and tuples: left to right
  - Records: use alphabetic order on the fields
  - `[1,2] < [1,3]  &&  (1,2) < (1,3)`

Exercises

- Chapter 1: 1.5–1.8

Sequence Enumerations

- Finite: `[1 .. 10]`
  - Note: `1` and `10` are types!
  - Required to compute sequence length.
- Infinite: `[1 ...]`
  - `1` is an arbitrary expression
- Other variants:
  - Arithmetic progressions: `[ 1, 3, .. 11 ]`
  - Decreasing sequences: `[ 11, 10 .. 0 ]`

More Sequence Operations

- Concatenation:
  - `[1..5] # [3,6,8] == [1,2,3,4,5,3,6,8]`
- Shifts and rotations
  - Shifts: `<<`, `>>`; rotations: `<<<`, `>>>`
  - `[0,1,2,3] <<  2 == [2,3,0,0]`
  - `[0,1,2,3] <<< 2 == [2,3,0,1]`

Sequence Comprehensions

- Similar to comprehensions in set theory
  - `[` *operation* `x | x <-` *sequence* `]`
- Apply an operation to each element in a sequence

```
[ 2 * x + 3 | x <- [1, 2, 3, 4] ]
  == [5, 7, 9, 11]
```

- Commonly used as a *control* structure

Traversals

- Cartesian traversal:

```
[ (x,y) | x <- [0,1,2]
        , y <- [3,4]
]
== [ (0,3), (0,4)
   , (1,3), (1,4)
   , (2,3), (2,4) ]
```

- Parallel traversal:

```
[ (x,y) | x <- [1,2,3]
        | y <- [3,4,7,11,18]
]
== [ (1,3), (2,4), (3,7) ]
```

Exercises

- Chapter 1, exercises: 1.9–1.30

Zero

- `zero` is a constant that inhabits all types:
  - `(zero : Bit) == False`
  - `(zero : [4]) == [False,False,False,False]` (i.e. `0x0`)
- Consider `xs`, a sequence of bits:
  - What does `xs == ~zero` say?
  - What does `xs !=  zero` say?

Exercises

- Chapter 1, exercises: 1.31–1.42

Types

- All Cryptol expressions have types
- Cryptol can automatically infer types for expressions
- Simple (*monomorphic*) types:
  - `Bit`, `Integer`, `Z n`
  - sequence: `[`*len*`]` *ty*, the length may be infinite, `inf`
  - tuples: `(` *ty1*, *ty2*, *ty3* `)`
  - functions: *input* `->` *output*
- Examples:
  - a 32-bit word: `[32] Bit`, equivalently `[32]`
  - sequences and matrices of words: `[8][32]`, `[10][8][32]`
  - an infinite sequence: `[inf] Integer`
  - functions: `([16],[16]) -> [16]`

Polymorphic Types

- A *polymorphic* type is a family of types
- `[2, 4, 5, 3] : {a} (fin a, a >= 3) => [4][a]`
  - a sequence of 4 elements
  - each element is a word of size `a`
  - `a` must be at least 3-bits long, and not `inf`
- `tail: {n,a} [1+n]a -> [n]a`
  - a function
  - expects a sequence with `1 + n` elements of any type as argument
  - it returns a sequence of `n` elements of the same type as result

Explicit Type Application

```
take : {front, back, a} (fin front) =>
                [front + back]a -> [front]a
```

- Positional: `` take`{20} ``
- Named: `` take`{front=20} ``
- Take all but the last 10: `` take`{back=10} ``

Exercises

- Chapter 1, exercises: 1.43–1.46

Functions

- Functions are *mathematical functions*
  - not procedures that return values
  - no mutable state, IO operations, etc.
- Functions can have multiple arguments, and can return multiple results in a tuple:

```
someFun : [8] -> [8] -> ([8],[8])
someFun x y = (xy, x + y)
  where xy = x * y
```

- Functions don't have to be named:

```
\x y -> 2 + x * y
```

Patterns

- *Patters* are convenient notation for accessing fields of value.
- They can be used in function arguments or when defining variables.
- At present, all Cryptol patterns are *irrefutable*.
- Example:

```
f (x, y # z) = z # [b,b,a] # y # x
  where
  [a,b] = y
```

Exercises

Chapter 1, exercises 1.47–1.54

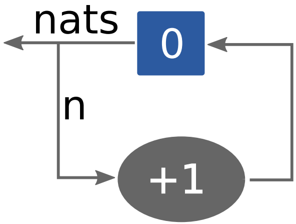
Other Useful Types and Functions

- Type `Z n`: integers modulo `n`
- Types `/^` and `%^`: division and modulus rounding *up*
  - Useful when working with padding
- Operators `<$` and `/$`: signed comparisons and division

Recurrences

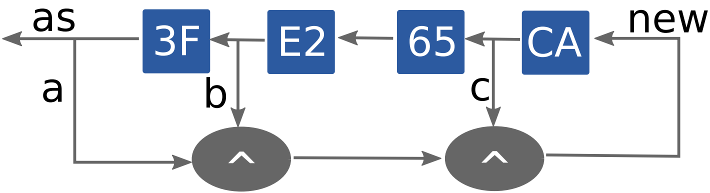
- Textual description of shift circuits
  - Follow mathematics: use stream-equations
  - Stream-definitions can be *recursive*
    - and can define infinite streams

```
nats = [0] # [ n + 1 | n <- nats ]
```



Stream Equations

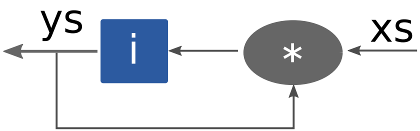
```
as  = [0x3F, 0xE2, 0x65, 0xCA] # new
new = [ a ^ b ^ c | a <- as
                  | b <- drop`{1} as
                  | c <- drop`{3} as ]
```



Computing Over a Stream

- Stream definitions may have parameters.
- The parameters may be other streams, *stream transformers*.
- Compute: `i`, `i*x0`, `i*x0*x1`, `i*x0*x1*x2`, ...

```
g i xs = ys
  where
  ys = [i] # [ x * y | x <- xs
                     | y <- ys ]
```

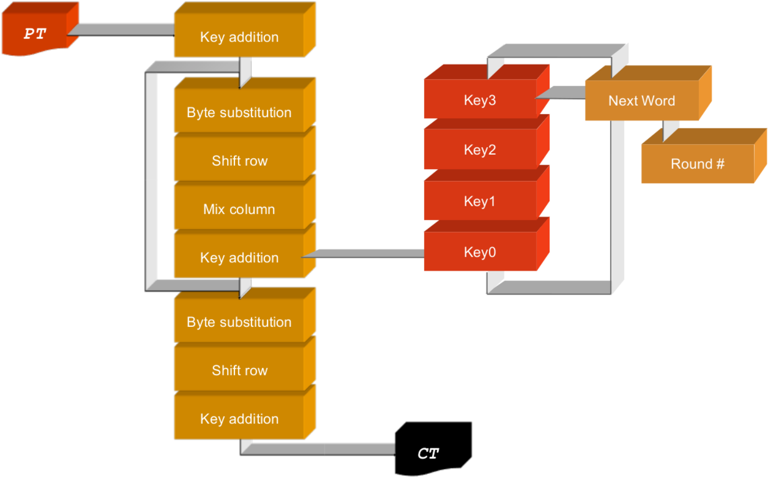


Exercises

- Chapter 1, exercises: 1.55–1.62

Constructing Cryptol Specifications

AES Structure



AES 256 API

Nb	Number of columns (32-bit words) comprising the State. For this standard, ***Nb*** = 4. (Also see Sec. 6.3.)
Nk	Number of 32-bit words comprising the Cipher Key. For this standard, ***Nk*** = 4, 6, or 8. (Also see Sec. 6.3.)
Nr	Number of rounds, which is a function of ***Nk*** and ***Nb*** (which is fixed). For this standard, ***Nr*** = 10, 12, or 14. (Also see Sec. 6.3.)

```
type Nb = 4          // Columns in the state
type Nk = 8          // Columns in the key
type Nr = 14         // Number of rounds
```

The State

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the **State**. The State consists of four rows of bytes, each containing ***Nb*** bytes, where ***Nb*** is the block length divided by 32. In the State array denoted by the symbol *s*, each individual byte has two indices, with its row number r in the range $0 \quad r < 4$ and its column number c in the range $0 \quad c <$ ***Nb***. This allows an individual byte of the State to be referred to as either $s_{r,c}$ or $s[r,c]$. For this standard, ***Nb***=4, i.e., $0 \quad c < 4$ (also see Sec. 6.3).

type State = [4][Nb][8]

Row traversals: `ShiftRows`

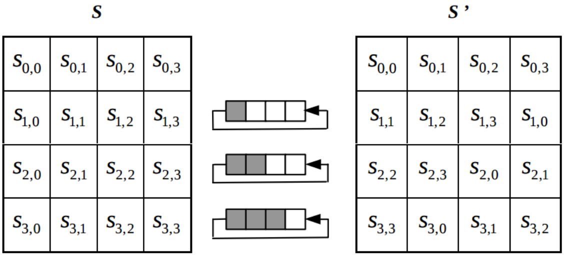


Figure 8. `ShiftRows()` cyclically shifts the last three rows in the State.

```
ShiftRows : State -> State
ShiftRows s = [ r <<< i | r <- s
                        | i <- [0,1,2,3] ]
```

Column traversals: `MixColumns`

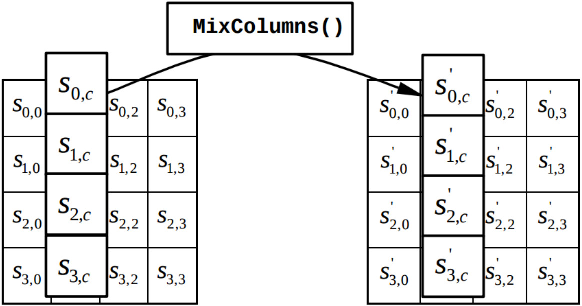


Figure 9. `MixColumns()` operates on the State column-by-column.

```
MixColumns : State -> State
MixColumns s =
  transpose [ ptimes col cx | col <- transpose s ]
```

Element traversals: SubBytes

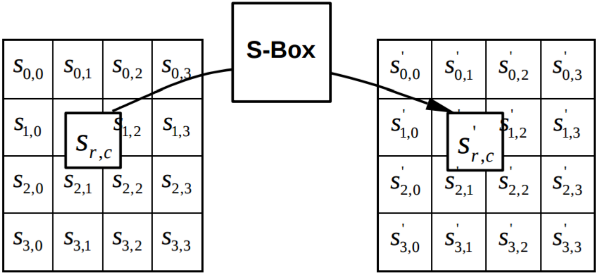


Figure 6. `SubBytes()` applies the S-box to each byte of the State.

```
SubBytes : State -> State
SubBytes s = split [ sbox @ a | a <- join s ]
```

AES Rounds

At the start of the Cipher, the input is copied to the State array using the conventions described in Sec. 3.4. After an initial Round Key addition, the State array is transformed by implementing a round function 10, 12, or 14 times (depending on the key length), with the final round differing slightly from the first ***Nr*** -1 rounds. The final State is then copied to the output as described in Sec. 3.4.

```
Rounds : [Nk + 1] RoundKey -> State -> State
Rounds ([k0] # ks # [kN]) i =
  FinalRound kN (rounds ! 0)
  where
  initS  = i ^ k0
  rounds = [ initS ] # [ Round k s | k <- ks
                                   | s <- rnd ]

Round      : RoundKey -> State -> State
FinalRound : RoundKey -> State -> State
```

Summary

- At the end of this tutorial you should have gained:
  - A basic understanding of the structure of the Cryptol language
  - Experience of creating and modifying programs in Cryptol
  - Experience with using the Cryptol REPL

Further Exercises

- Chapter 3, modeling the Enigma