A Brief Introduction to Cryptol

Galois Connections Inc.

15th March 2001

1 Introduction

Cryptol is a Domain Specific Language for expressing cryptographic algorithms. Galois Connections, Inc. is currently designing Cryptol, implementing an interactive development environment for Cryptol, and building a compiler that generates highly efficient C code implementations of cryptographic algorithms.

This document, and Cryptol itself, are very much a work-in-progress. View this document as a snapshot in the design process, not a final document.

The main features of Cryptol are:

- A uniform view of data, from bits all the way up to streams of blocks
- A uniform view of control, again from bits to streams of blocks
- A flexible, but strong type system
- Easy interoperability with C, Java, and most standard languages
- Ability to generate highly-efficient implementations

In Cryptol, data is composed of bits, and arbitrarily nested matricies. A computer word in Cryptol is just a matrix of bits. But Cryptol gives you very flexible views of this word. For example, this word might be a matrix of 4 bytes, and each byte might be viewed as a matrix of 2 4-bit nibbles. Cryptol allows the programmer to view data as a flat vector of bits, or as a structured hierarchy. Cryptol also allows the programmer to express infinite-sized matricies in order to easily express streams of data.

This very simple, but flexible, view of data has a corresponding simple and flexible notion of control. Control at all levels, from bits to streams is expressed using a matrix comprehension notation, akin to set comprehensions. Combining matrix comprehensions with recursive definitions allows us to directly express sophisticated recurrence relations that appear often in cryptographic algorithms.

2 Elements of Cryptol

Cryptol is a declarative language, thus there are no assignment statements or imperative loops. It is also a first-order language. A Cryptol program consists of a collection of definitions, either of values or of functions.

```
x = 13;
incr x = x + 1;
foo (x, y) = 2 * x + 3 * y - 1;
```

The building blocks of Cryptol are bits, which are written `True` for a one bit, and `False` for a zero bit. The more common building block of Cryptol programs, however, is *words*, i.e. vectors of bits. A numeric literal in a Cryptol program is represented as a word. The type of a word is written by enclosing the size in bits in square brackets: e.g. the type of a 32-bit word is written `[32]`.

Numeric literals can either be written in the usual base 10, or expressed as hexidecimal constants, by prefixing with `0x`, as in `0x1fc3`.

2.1 Matricies

Data is structured in Cryptol using matricies. These are written as square brackets around space separated elements. The elements of a matrix may be bits, or other matricies. However, the elements at any given level in a matrix must all be the same size. A word is just a single-dimension matrix.

Matricies whose elements are enumerations can be expressed using `..`. For example, the matrix consisting of the elements from 1 to 10 can be written `[ 1 .. 10 ]`. If two starting elements are provided, the difference between the two provides the delta for subsequent values. Thus:

```
[ 1 3 .. 9 ] ⟼ [ 1 3 5 7 9 ]
```

The elements of a matrix are indexed from left-to-right, starting with 0. Matricies are indexed using the operator `@`. In the following, `y` will have value `9`.

```
xs = [13 27 9 34];
y  = xs @ 2;
```

The `@@` operator can be used to construct a new matrix out of elements of another matrix. In the following, ys will have value `[27 34]`.

```
ys = xs @@ [1 3];
```

The bits of a word are indexed in little-endian fashion. In the following, `y` has value `True`, and `z` has value `5`.

```
x = 0x85
y = 0x80 @ 7
z = x @@ [0 .. 3];
```

Copyright © 2001

2.2 Arithmetic

Many of the basic arithmetic operators are defined over words. Arithmetic in Cryptol is modulo the size of the word. The operators are `+`, `-`, `*`, `/`, `%` (modulo) and `**` (power).

2.3 Boolean operations

Cryptol has the standard boolean operations of *and*, *or*, *exclusive-or* and *complement*, written as `&`, `|`, `^` and `~`. These operations in Cryptol are bulk operators - they work on everything from Bit to arbitrarily-nested matricies. The operations on Bit are standard, and on matricies, they are defined element-wise.

2.4 Equality

The equality operator in Cryptol is written `==`. It compares two like-typed values, and returns a result of type `Bit`. Matricies are compared element-wise, to arbitrary depth.

```
1 == 2 ⟼ False
[ [13 34] [14 9] ] == [ [13 34] [14 9] ] ⟼ True
```

2.5 Comparison

The standard comparison operators are available: `<`, `>`, `<=`, `>=`. These are only defined over words, and have a result of type `Bit`.

2.6 Conditional Expressions

The standard if-then-else construct is available. The first expression is expected to be of type `Bit`, and the `then` and `else` expressions are expected to be the same type.

```
min (x, y) = if x <= y then x else y
```

2.7 Shifts and Rotates

The shift and rotate operators in Cryptol allow shifting/rotating at the outer level of any matrix. Shifts are written using the operator `<<` or `>>`. The left-hand argument is the matrix to be shifted, and the right-hand argument is a word describing how much to shift. The matrix is filled in with zeros, where zero is understood to be an appropriately typed element that is all zeros.

```
xs = [0 1 2 3 4 5 6 7];
xs << 4 ⟼ [4 5 6 7 0 0 0 0]
yss = [ [0 1] [2 3] [4 5] [6 7] ];
yss >> 2 ⟼ [ [0 0] [0 0] [0 1] [2 3] ]
```

Copyright © 2001

```
x = 0x1234;
x >> 8 ⟼ 0x12
```

The final example should raise an eyebrow. In the second example, the shift right filled in zeros in the lower-numbered elements of the matrix. However, in the third example, the zero bits were filled into the higher-numbered bit-positions. Shifts and rotates on words are treated differently from shifts/rotates on larger-dimensions matricies. For words, shift left moves elements from lower-indexed positions to higher-indexed positions. This is reversed for higher-dimension matricies: shift left moves elements from higher-indexed positions to lower. This inconsistency is due to the ambiguity of the term "left", and is somewhat forced on us by historical precedent: shift-left traditionally means making a value larger. But it is also consistent with how data is presented, since literals are printed big-endian (highest bit position on the left), while higher-dimensioned matricies are printed little-endian (least element position on the left).

2.8 Joining matricies

Matricies of differing widths (but the same element types) may be joined together into one matrix using the join operator.

```
xs = [0 1 2 3];
ys = [4 5 6];
xs # ys ⟼ [0 1 2 3 4 5 6];
```

2.9 Matrix Comprehensions

A very convenient way of describing matricies is the matrix comprehension notation, analogous to set comprehension notation. A matrix is described by drawing and combining elements from other matricies, referred to as *generators*. When there are multiple generators, the elements produced form the cartesian product of the two matricies.

```
[ [x y] || x <- [0 1], y <- [4 .. 7] ]
⟼
[[0 4] [0 5] [0 6] [0 7] [1 4] [1 5] [1 6] [1 7]]
```

Groups of generators may also be combined in parallel. In this case, the elements produced will be only as long as the elements of the shortest group.

```
[ [x y z] || x <- [0 1], y <- [4 .. 7]
          || z <- [0 .. 4] ]
⟼
[[0 4 0] [0 5 1] [0 6 2] [0 7 3] [1 4 4]]
```

Parallel generators are very useful for doing element-wise operations between matricies. For example, if we didn't have exclusive-or built-in, we could have defined it as follows:

```
xor (xs, ys) = [ (x & ~y) | (~x & y)
               || x <- xs || y <- ys ];
```

2.10 Size Polymorphism

Cryptol has a very flexible notion of the *size* of data. When a numeric literal is used in a program, it will take on whatever size (i.e. number of bits) is demanded by its surrounding context. For example, if we have a function `twizzle`, which expects a 32-bit word argument, then the parameter in the call `twizzle 14` will be interpreted as a 32-bit literal.

But things get interesting when the context *doesn't* immediately constrain the size. A value or function with unconstrained sizes is *size polymorphic*. However, this polymorphism isn't completely unbounded - a numeric literal is constrained to be at least as large as the number of bits necessary to represent the literal itself. E.g. the literal `32` would require at least 6 bits. This constraint is reflected in the type given, with size constaints given to the left of a `=>`. E.g. the literal `32` has type `(a >= 6) => [a]`.

The builtin function `sizeOf` can be used to determine the size of the outermost dimension of an expression. For example, `sizeOf([1 2 3 4])` is equal to 4.

2.11 Shape Polymorphism

In addition to size polymorphism, Cryptol functions can also be polymorphic about the number of dimensions a value has, i.e., its shape. This is expressed in the type of a matrix by putting a type variable after the sequence of outer dimensions. For example, the type of a matrix that has four elements, but we don't know or care what those elements are, would be `[4]a`.

A nice example of this is a function that re-arranges the elements of a matrix:

```
swab [a b c d] = [d c b a];
```

This function has type `[4]a -> [4]a`.

2.12 Subtyping

A common step in a crypto algorithm is to divide up words into smaller pieces, such as dividing a 128-bit word into 4 32-bit words. Cryptol provides a simple form of subtyping that make this particularly convenient. Any function that expects, for example, a matrix of 4 32-bit words may also be passed a 128-bit word. Consider the following function.

```
frotz [a b c d] = a + b - c + d
```

We can pass a matrix consisting of four words to it, but we can also pass a lower dimension matrix whose outer dimension (in this case, single dimension) is a multiple of four. If we assume that we're doing 8-bit arithmetic (i.e., `a b c d`

are all 8-bits wide), then we can either pass `frotz` a value of type `[4][8]`, or a value of type `[32]`.

```
frotz [1 2 3 4] ⟼ 4
frotz 0x04030201 ⟼ 4
```

2.13 Controlling Polymorphism

Definitions in Cryptol can be given a type signature. The use of the signature is optional, but is often useful, and even necessary. The flexibility of the type system can lead to programs that are more polymorphic than the programmer expects or needs, and type signatures can be used to restrict polymorphism. E.g., the following defines a constant, which is constrained to be 32-bits wide:

```
x : [32];
x = 13;
```

Consider the definition of `xor` given in Section 2.9. If we didn't constraint its type, it would be assigned a type more polymorphic than the user might want. The assigned type would be:

```
xor : {a b c} ([a]b,[c]b) -> [min(a,c)]b;
```

If the user was intending to define a function only over words, then this defintion is both more size polymorphic (works over matricies of differing sizes) and shape polymorphic than desired (works over arbitrarily-dimensioned matricies). While this flexibility might be useful in some cases, often it will lead to ambiguities later on. A more useful type might be to constrain it only to work on arbitrary-sized words of the same length.

```
xor : {a} ([a], [a]) -> [a];
```

2.14 Streams and Recursively Defined Matricies

Cryptol can also express matricies of unbounded size, which we'll call *streams*. Streams allow us to model such things as shift registers, recurrence relations and cryptographic modes. The simplest way to define a stream is by using an unbounded matrix enumeration

```
[ 0 .. ] ⟼ [ 0 1 2 3 4 5 6 ···]
```

Streams may also be defined recursively. We can model a simple counter with the following definition:

```
counter init = [init] # counter (init + 1);
```

The streams `[ 0 .. ]` and `counter 0` will denote the same thing.

The size of a stream is indicated in the type system by a special constant, named `ko`. Each `ko` size is annotated with a parameter indicating the cycle size of the recursion. In the above examples, both streams have type `[ko(1)]`.

The cycle size annotation indicates the amount of state necessary to generate the stream. It is calculated during type inference by noting the number of elements on the left of a `#` between a definition and each recursive reference. This is useful in code generation, but is also useful for detecting ill-formed recursion - a cycle size of zero is bad news, as is a cycle size of `ko`. For example, the following definition has a cycle size of 0, and is rejected by the type checker:

```
xs = [ x + 1 || x <- xs ];
```

3 Some Basic Idioms

Learning how to use a new language usually involves learning its idioms. Here we give some basic Cryptol idioms that we've identified.

3.1 Padding

It is very common to pad data to a larger size. In Cryptol, we can take advantage of size polymorphism to define padding in a very declarative fashion. In the following example, we pad a key on the right with a single one bit followed by zeros and finally a 64 bit size of the key. Notice that the key is constrained to have a size that can be expressed in 6 bits.

```
pad : {a} (6 >= a) => [a] -> [512];
pad key = key # [True] # 0 # sz
 where {sz : [64];
 sz = sizeOf key;};
```

The 0 in the middle is unconstrained in size, while everything else is of a fixed or given size. Thus, in order to make the result be 512 bits wide, the 0 will have to expand to fill all available space as necessary to pad the result out to 512 bits.

3.2 For Loops and Recurrence Relations

In an imperative language, for-loops are a fundamental building block of cryptographic algorithms. In Cryptol, we express this control more declaratively using matrix comprehensions and by defining recurrence relations.

For example, consider the following imperative loop in C:

```
sum = 0;
for (i = 0; i < 10; i++)
 sum = sum + i;
return sum;
```

Copyright © 2001

In Cryptol we would identify the state (`sum`) and then write a recurrence relation corresponding to this loop. We define the sequence of states (`sums`) with an initial value of `0`. To get the final value we index into the sequence of states at the desired location: `sums @ 10`.

```
result = sums @ 10;
sums = [0] # [ sum + i || sum <- sums
 || i  <- [0 .. 9]
 ];
```

Here another example from the SHA1 specification of the compression function:

1. Divide Mi into 16 words W0, W1, ... , W15, where W0 is the left-most word.
2. 2. For t = 16 to 79 let Wt = S1(Wt-3 XOR Wt-8 XOR Wt- 14 XOR Wt-16).

In Cryptol we identify the state as being initialized for the first 16 steps with `m`. Each of the succeding states is calculated by offsets into the state. For example to reference a state that occured 14 steps ago in a 16 entry state we have to drop the first two elements, ie indexing the state at an offset of 2: `(ws @@ [2 ..])`.

```
compress1 : [16][32] -> [80][32];
compress1 m = ws @@ [0 .. 79]
 where ws = m # [(w3 ^ w8 ^ w14 ^ w16) <<< 1
 || w16 <- ws
 || w14 <- ws@@[2 ..]
 || w8  <- ws@@[8 ..]
 || w3  <- ws@@[13 ..]
 ];
```

3.3 Cryptographic Modes

Cryptographic modes are easily described using matrix comprehensions. Electronic Code Book is particularly straightforward.

```
ecb (xs, key) = [ encrypt (x, key) || x <- xs ];
```

CBC mode is expressed as a simple recurrence relation.

```
cbc (iv, xs, key) = ys
 where ys = [ encrypt (x ^ y, key)
 || x <- xs
 || y <- iv # ys ];
```

4 Examples

4.1 DES

The following example defines the core encryption routine for DES.

Algorithm 1 DES encryption

```
des :  {a b} (a >= 7) => ([2**(a-1)],[b][48]) -> [64]
des (pt, keys) = permute (FP, swap last)
  where { pt' = permute (IP, pt);
          iv  = [ round (k, lr)
                  || k  <- keys
                  || lr <- [pt'] # iv ];
           last = iv @ (sizeOf keys - 1);
         };

round (k, [l r]) = r # (l ^ f (r, k));

f (r, k) = permute(PP, SBox(k ^ permute(EP, r)));

swap [a b] = b # a;

permute :  {a b} (b >= 1) => ([a][b], [2**(b - 1)]) -> [a];
permute (p, m) = [ m @ (i - 1) || i <- p ];
```

4.2 RC6

The following defines the key schedule for RC6.

Algorithm 2 RC6 key schedule

```
rc6ks :  {a} (w >= sizeOf a) => [a][8] -> [r+2][2][w];
rc6ks key = chop2 (rs >>> (v - 3 * nk))
  where {
    c = max (1, (sizeOf key + 3) / (w / 8));
    v = 3 * max (c, nk);
    initS = [ pq || pq <- [pw (pw+qw) ..]
            || i  <- [0 ..  (nk-1)] ];
    padKey = key # [ 0 || i <- [(sizeOf key + 1) ..  c] ];
    initL = combine4 padKey;

    ss = [ (s+a+b) <<< 3     || s <- initS # ss
                         || a <- [0] # ss
                         || b <- [0] # ls ];
    ls = [ (l+a+b) <<< (a+b) || l <- initL # ls
                        || a <- ss
                        || b <- [0] # ls ];
    rs = ss @@ [(v-nk) ..  (v-1)];
  };
```