| galois |

Background: Cryptol Types

Type Classes

Data structure types descend from type classes. This allows describing behaviors that are shared by more than one type. An example is the operator `(==)`. One would expect this operator to apply to various kinds of numbers, say for example, rational numbers and integers. But this operator can apply to sequences, strings, even functions with the variant `(===)`. To see how this operator is typed run this at the `Cryptol>` prompt: `:t (==)`. The result is

```
(==) : {a} (Eq a) => a -> a -> Bit
```

This says the left argument is of type a and the right argument is of the same type `a` and the output is a `Bit` which is either `True` or `False`. The only restriction on type `a` is that it is a descendant of the `Eq` class which means Cryptol knows how to check for equality, either by some built-in function or by some user supplied function. Some examples (all evaluate to True):

```
1 == 1
“peanut” == “peanut”
(ratio 7 2) == (ratio 14 4)
```

The following type classes are defined in Cryptol:

- The `Logic` typeclass includes the bitwise logical operators `&&`, `||`, `^`, and propositional connectives `/\`, `\/`, and `~` (and, or, not, respectively). Cryptol types made of bits (but not those containing unbounded integers) are instances of class `Logic`.
- The `Zero` typeclass includes the special constant `zero`. The shifting operators << and >> are also in class `Zero`, because they can shift in `zero` values. All of the built-in types of Cryptol are instances of class `Zero`.
- The `Eq` typeclass includes the equality testing operations `==` and `!=`. Function types are not in class `Eq` and cannot be compared with `==`, but Cryptol provides a special pointwise comparison operator for functions with the following signature:

  ```
  (===) : {a b} (Cmp b) => (a -> b) -> (a -> b) -> a -> Bit
  ```

- The `Cmp` typeclass includes the binary relation operators `<`, `>`, `<=`, `>=`, as well as the binary functions `min` and `max`.
- The `SignedCmp` typeclass includes the binary relation operators `<$`, `>$`, `<=$`, and `>=$`. These are like `<`, `>`, `<=`, and `>=`, except that they interpret bitvectors as signed 2’s complement numbers, whereas the `Cmp` operations use the unsigned ordering. `SignedCmp` does not contain the other atomic numeric types in Cryptol, just bitvectors.
- The `Ring` typeclass includes the binary operators `+`, `-`, `*`, and the unary operators `negate` and `fromInteger`. All the numeric types in Cryptol are members of `Ring`.

- The `Integral` typeclass represents values that are “like integers”. It includes the integral division and modulus operators `/` and `%`, and the `toInteger` casting function. The sequence indexing, update and shifting operations take index arguments that can be of any Integral type. Infinite sequence enumerations `[x ...]` and `[x, y...]` are also defined for class `Integral`. Bitvectors and integers are members of `Integral`.
- The `Field` typeclass represents values that, in addition to being a `Ring`, have multiplicative inverses. It includes the field division operation `/.` and the `recip` operation for computing the reciprocol of a value. Currently, only type `Rational` is a member of this class.
- The `Round` typeclass contains types that can be rounded to integers. It includes `floor`, `ceiling`, `trunc`, `roundAway` and `roundToEven` operations, which are all different ways of rounding values to integers. Currently, only one type, namely `Rational`, is a member of this class.
- The `Literal` typeclass includes numeric literals.

Type variables and value variables:
A value variable is the kind of well-known variable from normal programming languages. This kind of variable represents a normal run-time value. A type variable, on the other hand, allows expressing interesting (arithmetic) constraints on types such as lengths of sequences or relationships between lengths of sequences. Type variable values are computed statically and they never change at runtime.

Positional vs. named type arguments
Cryptol permits type variables to be passed either by name, or by position. For user defined functions, the position is the order in which the type variables are declared in the function’s type signature. The `:t` command can be used to find out the position of type variables. For example:

```
Cryptol> :t groupBy
groupBy : {each, parts, a} (fin each) => [each*parts]a -> [parts][each]a
```

according to the above, `parts` is in the second position of `groupBy`’s type signature, so the positional-style call equivalent in this case is:

```
Cryptol> groupBy`{_,3}[1..12]
```

which results in

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Observe, 4 can be used for each instead of `_`. Try something different and an error occurs. The `_` is used because the value of 4 is forced and supplied by Cryptol so the user does not have to worry about that in this case. Observe `groupBy `{4,_}[1..12]` could have been used for the same result. A function that will be used quite a bit later has the following signature:

```
take : {front, back, a} [front + back]a -> [front]a
```

This is applied to sequences, primarily to grab a section of a sequence, usually an infinite sequence, which would not be observable. An example:

```
take `{4,_}[1..12] = [1,2,3,4]
```

alternatively,

```
take `{4}[1..12] = [1,2,3,4]
```

works as well.

Inline argument type declarations

An example is:

```
Cryptol> let addBytes (x:[8]) (y:[8]) = x + y
```

which specifies 8 bit arguments instead of

```
Cryptol> let addBytes x y = x + y
```

which has the inferred signature

```
addBytes : {a} (Ring a) => a -> a -> a
```

The first signature above is an example of a monomorphic type and the inferred signature is an example of a polymorphic type. Monomorphic types are required to prove properties and it is often the case that such types be defined. Observe that `addBytes` could have been written like this:

```
addBytes : [8] -> [8] -> [8]
addBytes x y = x+y
```

which is not inline and would be loaded from a file, hence no `let`.

Basic Data Types

There are seven basic data types in Cryptol: bits, sequences, integers, integers mod n, rationals, tuples, records. Cryptol is statically typed and uses type inference to supply unspecified types. But, a user may change the inferred type to something more restrictive, for example to declare a function to be monomorphic instead of the more general polymorphic inferred type.

Bits

A Bit has value True or False and nothing else. The value True must be capitalized, similarly for False. Test expression in an if-then-else statement must have the type Bit. The logical operators && (and), || (or), ^ (xor), and ~ (complement) provide the basic operators that act on bit values.

Integers

Unbounded integers can be positive or negative with arbitrarily large magnitude. Unbounded Integers are used to express decimal numbers as ratios. For example 14.25 is expressed in Cryptol as `(ratio 1425 100)` which is reduced to and displayed as `(ratio 57 4)`. The number π may be approximated by `(ratio 355 113)` or `(ratio 245850922 78256779)` or something else. The inferred type of an unbounded integer, say 43, is

```
Cryptol> :t 43
43 : {a} (Literal 43 a) => a
```

Observe this is a polymorphic type. Bounded integers are typed according to the number of Bits they are composed from. That number may be inferred or user declared. For example

```
Cryptol> let x = 3:[10]
Cryptol> let y = 3:[10]
Cryptol> :t x+y
(x + y) : [10]
```

So, x and y are user-declared to be 10 Bits wide, the sum x+y is inferred to be 10 Bits wide. But,

```
Cryptol> let x = 3
Cryptol> let y = 3
Cryptol> :t x+y
(x + y) : {a} (Ring a, Literal 3 a) => a
```

results in a sum of unbounded integer type. Now

```
Cryptol> let a = 3:[10]
Cryptol> :t x+y+a
(x + y + a) : [10]
```

So, Cryptol has turned the sum into a bounded integer of 10 Bits. To make the sum an unbounded integer do something like this:

```
Cryptol> let c = toInteger (x+y+a)
Cryptol> :t c
c : Integer
```

`Modulo arithmetic is automatically implemented through bounded arithmetic. For example, (67*23):[7] = 5 mod 128 = 5 mod` 2^7. However, the modulus is a power of 2 in such cases. For another modulus, use `%`. For example, `(67*23) % 17` is `11`. If the modulus is a prime number a given number less than the modulus has an inverse in the field of that modulus and that inverse may be found using `recip`. For example, `(recip 5) : Z 17` is 7 and `7*5 % 17` is 1, demonstrating that the inverse of 5 is 7 mod 17.

Rationals

A rational is a ratio of two integers and is written, for example, as `(ratio 7 2)`. Multiplication is by operator `*`, for example `(ratio 7 2)*(ratio 4 14)` is `(ratio 1 1)`. Division is by the operator `/.` so, for example, `(ratio 7 2) /. (ratio 14 4)` is `(ratio 1 1)`. Additon by `+` works as expected, for example `(ratio 7 2) + (ratio 14 4)` is `(ratio 7 1)`. A rational can be created from integers using `fromInteger`, e.g. `fromInteger 2:Rational` is `(ratio 2 1)`.

Floating Point Numbers

To use floating point numbers the floating point module needs to be loaded with

```
Main> :m Float
Float> 1.2
Float> (ratio 6 5)
Float> :t 1.2
fraction`{6,5} : {a} (FLiteral 6 5 0 a) => a
```

By hitting the tab key at the command prompt, an additional 35 or so commands are seen to become available such as `fpAdd` and `=.=`.

Sequences
A ordered collection of elements all of which are the same type. These may be nested. Examples and their types are as follows:

```
Cryptol> let x=[1,2]
Cryptol> :t x
x : {a} (Literal 2 a) => [2]a
Cryptol> let y = [x,x+x,x+x+x]
Cryptol> y
[[1, 2], [2, 4], [3, 6]]
Cryptol> :t y
y : {a} (Ring ([2]a), Literal 2 a) => [3][2]a
```

Frequently used operations on sequences include `#,@,@@,!,!!` with some examples provided as follows:

```
Cryptol> let x = [1,2,3,4,5]
Cryptol> [0]#x
[0, 1, 2, 3, 4, 5]
Cryptol> x!0
5
Cryptol> x!![1,0]
[4, 5]
Cryptol> x@0
1
Cryptol> x@@[0,1]
[1, 2]
```

Tuples
Elements of a sequence must all be the same type. That is not the case for tuples. An example of tuples:

```
Cryptol> let x = ("peanut", 34, (ratio 6 5))
Cryptol> let y = ("butter", (ratio 3 1), 4)
Cryptol> x+y
([210, 218, 213, 226, 218, 230], (ratio 37 1), (ratio 26 5))
```

Use '.' as in the following example to extract a tuple member

```
Cryptol> x.0
[112, 101, 97, 110, 117, 116]
Cryptol> x.1
34
Cryptol> x.2
(ratio 6 5)
```

Enumerations
Compact representations of sequences. The following are examples:

```
Cryptol> [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Cryptol> [1,3..10]
[1, 3, 5, 7, 9]
Cryptol> [10,9..1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
Cryptol> [10,7..1]
[10, 7, 4, 1]
Cryptol> let x = [1..10]:[10][16]
Cryptol> :t x
x : [10][16]
Cryptol> let y = [1..10:[16]]
Cryptol> :t y
y : [10][16]
Cryptol> let z = [-1,-1 ...]
Cryptol> z
[-1, -1, -1, -1, -1 ...]
```