Fundamentals of Computer Science — Lecture Notes

Andrew Clifton

January 15, 2016

Preliminaries – Haskell

Syntax

Comments in Haskell take two forms:

- Single-line comments (similar to `//`-style comments in C/C++/Java) start with `--` and extend to the end of the line. E.g.,

  ```
  -- This comment extends to the end of the line.
  ```

  Note that in Haskell you can define your own operators, and it is perfectly acceptable to define an operator that starts with `--`, as long as it continues with some other operator-like character. E.g., we could define

  ```
  (-->) :: Int -> Int -> Int
  a --> b   = a*a + b*b
  ```

  Some editors, however, will not be aware of this and may show everything after the start of the `-->` as a comment.

- Multi-line comments (similar to `/* ... */`) start with `{-` and end with `-}`. Note that multi-line comments *can* nest, unlike in C/C++/Java. E.g., this is perfectly valid:

  ```
  {-
     This is commented out
     {- So is this -}
     This is still commented out
  -}
  ```

Identifiers (functions, variables, types, etc.) are subject to a few rules regarding their names:

- Variable and function names must start with a lowercase letter, but may be followed by upper- and lower-case letters, numbers, underscores, or the apostrophe. The latter is commonly used to show that one variable is a slightly different version of another. E.g.,

  ```
  if a == a' then ...
  ```

The exception to this rule are operator-style functions, which, as shown above, have to start with some kind of symbol character (`-`, `*`, etc.).

- Identifiers starting with uppercase letters are reserved for module names, types, type classes, and type constructors.

  Note that this conflicts with our usual mathematical custom of writing the names of sets in uppercase. Usually we will get around this by writing sets or lists with a plural 's' at the end. E.g., the set A will become `As` ("more than one a"). Bear this in mind when translating math into Haskell; you'll have to do some renaming, so be consistent about it.

- The two type constructors you will probably see most frequently are `True` and `False`, both of type `Bool`. These are, as you might expect, the boolean constants. Note that they must start with uppercase T and F! If you write `true` by accident you will get an `undefined identifier` error.

  Literal values:

- Integer and floating-point values look like you'd expect:

```
1
105
0.5
-12.4
```

  But note that there is an unfortunate ambiguity with the unary minus, so often it's safer to write `(-12.4)` with explicit parentheses.

- Character literals are enclosed in single-quotes (forward quotes; backquotes do something different):

```
'a'
'b'
```

  Note that Haskell supports Unicode, so you can use fancy characters if you want.

- String literals are enclosed in double-quotes:

```
"Hello, world!\n"
```

  As shown, the usual backslash escapes for special characters are supported. In Haskell, strings are just lists of characters (i.e., of type `[Char]`), so technically a string literal is just another form of a list.

- List literals have two forms, explained in detail below, in the section on lists. Some examples:

```
[1,2,3,4]
1:2:3:4:[]       -- Same as the previous
1:(2:(3:(4:[]))) -- Also the same
1:2:[3,4]        -- Still the same
```

- Tuples consists of multiple values, of possibly different types, in parentheses. Tuples are explained more fully below, but here are some examples:

```
(1,"hello")                  -- A "pair" of type (Int, [Char])
(3.14,[True,False],"potato") -- A "triple" of type (Float, [Bool], [Char])
```

- Function values are explained in detail below, but they begin with a backslash, followed by arguments, followed by ->, followed by the body:

```
(\x -> x)   -- The identity function
(\v -> v+1) -- The successor function
(\x y -> x + y) -- This is the same as the function (+)
```

Any function can be treated as an infix operator, and any infix operator can be treated as a function, as you find it convenient:

- If f is a function of two arguments then

```
f a b
```

  is exactly the same as writing

```
a `f` b
```

- If + is any infix operator, then

```
a + b
```

  is exactly the same as writing

```
(+) a b
```

  This is mostly useful in situations not where you are calling `(+)` as a function, but where you are storing it in a variable, or passing it as an argument (see the section on "First Class Functions" below for examples).

Layout: Indentation is *significant* in Haskell, meaning that it affects the meaning of your code. Generally speaking, if a line is indented more than the previous line, then it is treated as being part of a new block (this is similar to the layout rule in Python). The block ends with the next line that is unindented (indented *less* than its parent). An example:

```
f x = x + 1
g y = y * 2
```

This would not work if we wrote it as

```
f x = x + 1
 g y = y * 2
```

because now the definition of `g` appears to be nested inside that of `f`, somehow. This is actually a common problem, where the definitions are separated by enough space to make the extra indentation less noticable. Always check your definitions to make sure they are flush left!

If you find the layout-based structure problematic, Haskell also has an explicit block syntax that should be more familiar:

```
let {
x = 1;
y = 2;
} in
  x + y
```

(Note that the expression following the `in` must still be indented, or you could put it on the same line as the `in`.)

The only places where layout really matters are:

- After `let` but before `in`
- After `of` (i.e., after `case ... of`)
- After `where`

Thus, these are the only situations where the block syntax will really help. Some additional examples of block syntax:

```
case x of {
1 -> "Hello";
2 -> "Goodbye";
}
```

```
f x = x + y + z
  where {
y = 4;
z = 2;
  }
```

(The crazy indentation is just to show that it doesn't matter in a block; in reality you should try to make your code readable.)

Note that you can use a semicolon anywhere where a newline would normally occur. E.g., you can put multiple definitions on a single line if you like:

```
x = 1; y = 2; z = 3;
```

Haskell File Structure

A Haskell file typically has the extension .hs. A Haskell file can optionally begin with some module imports, followed by *definitions*. (Note that, as in Java, any module imports must appear before all definitions; you cannot mix and match imports and definitions throughout the file.) A module import looks like this

```
import Data.List
```

Haskell uses a hierarchical module structure: here we are importing the List module, which is nested inside the Data module. An unqualified import will load in *every* definition provided by the file. We can qualify the import if we only want to load in specific definitions:

```
import Data.List (permutations, subsequences)
```

This will only import the two functions `permutations` and `subsequences`.

Definitions in Haskell have the (very) general form of some identifier, optionally some argument pattern(s), a literal `=`, and then the body of the definition. For example,

```
x = 10
f x = x + 2
first_two (x1:x2:xs) = x1 + x2
squared_dist a b = a^2 + b^2
```

A definition can optionally be preceded by a type declaration. This consists of the name of the definition, followed by `::`, followed by a type:

```
x :: Int
x = 10

f :: Integer -> Integer
f x = x + 2

first_two :: [Int] -> Int
first_two (x1:x2:xs) = x1 + x2

squared_dist :: Num a => a -> a -> a
squared_dist a b = a^2 + b^2
```

If the type is omitted, Haskell will figure it out for you. But if you give a type, Haskell will still figure out the type, and then check it against the type you gave. This is a good way of "checking your work"; if you and Haskell disagree about the types, probably something went wrong someplace.

The forms that a definition can take are quite varied:

- A single definition can have multiple *clauses*, each matching a different *pattern*. We've already seen this in recursive list functions: we have a clause for the empty list, and then another clause for the cons. Haskell will try to match the actual argument(s) against the clauses' patterns in the order they are given; the first one to match is used.

- A single clause of a definition can have *guards*. Guards allow a single clause to be split into multiple cases, with the case chosen depending on some boolean conditions. For example, here is a function which clamps the value of its first argument to be >= its second and <= its third:

```
clamp :: Int -> Int -> Int -> Int
clamp x a b | x <= a    = a
            | x >= b    = b
            | otherwise = x
```

  `otherwise` is just a synonym for True, acting as an "else" case. If *none* of the guards succeed, then the entire clause is treated as a failed pattern match.

- A clause can have *local definitions* via `where`:

```
f x = x + x2 - z
    where
        x2 = 2 * x
        z = 12
```

  These local definitions are full definitions in their own right: they can have types, multiple clauses, guards, even nested `where`s! The only difference is that definitions in a `where` are only visible within the body of their attached definition (e.g., above, you cannot refer to `x2` and `z` anywhere but in the definition of f).

- If Haskell gets to the end of the list of clauses and none of them has matched, then it will throw an "inexhaustive match" error.

Patterns

Patterns are what follow the name of a definition on the left-hand side. Although simple argument patterns like

```
f x y = ...
```

are not too hard to understand (f takes two arguments, named x and y within its body), argument patterns can in fact be quite complex and expressive. (Note that for functions with multiple arguments, each argument gets its own pattern.) The forms of patterns are:

- A variable, e.g., `f x y` as above. A variable matches anything, and will result in the matched value being bound to the name of the variable, within the body of the definition.
- The wildcard variable _. _ matches anything, but does *not* bind any name to the value. You can use this for arguments that you don't care about; e.g., in our length function, we did not use the value of `x` in the cons case, so we could have written it as

```
mylen (_:xs) = 1 + mylen xs
```

- A literal value, which must match exactly, and does not bind any names. E.g., in the factorial function, the base case is

```
fact 1 = 1
```

  This clause will only match if the actual argument is 1.

- A data constructor. We've already seen one of these, the cons constructor:

```
mylen (x:xs) = ...
```

  This will match the non-empty list case, and will also split the list into its head and tail, and bind x to the head and xs to the tail. This can be done with any data constructor, including those for data types you create yourself:

```
data NameOrNumber = Name String | Number Int

isName :: NameOrNumber -> Bool
isName (Name _) = True
isName (Number _) = False
```

  Data constructors can also contain nested patterns. As shown above, we can use _ within a constructor. You can even next data constructors within data constructors:

```
addFirstTwo :: [Int] -> Int
addFirstTwo (x1:(x2:_)) = x1 + x2
```

  Remember that `[a,b,c]` is just shorthand for `a:b:c:[]` (which in turn is equivalent to `a:(b:(c:[]))`), so you can use patterns of the form `[a,b,c,...]` to match a list of specific length. (And again, the elements of this list pattern could themselves be nested patterns!)

- An "as" pattern. Sometimes you want to break up a data constructor (e.g., extract the head and tail of a list) but *also* have access to the complete original value. An `@` pattern does just this:

```
f l@(x:xs) = ...
```

Here, x and xs will be bound to the head and tail as usual, but l will also be bound to the entire original list.

Expressions

While Haskell files consist of definitions, the *body* of every definition must be an expression. Hence the structure of expressions is very important. (Note that when we talk about "builtin" operators and functions, we are actually refering to the set of operators/functions defined in the *Haskell Prelude*. The Prelude is a special module that is automatically imported by every Haskell file; it is also automatically available in GHCi.)

Operators: Haskell supports the usual collection of arithmetic operators:

```
+ - * / ^
```

These are defined on all "numeric" types (i.e., types implementing the `Num` typeclass; see below for a description of what typeclasses are).

Comparison operators are similarly available:

```
> < >= <= == /=
```

The ordering operators (less-than, etc.) are defined on types implementing `Ord`, while the (in)equality operators are defined on types implementing `Eq`. (Note that all numeric and character types implement both of these; structured types like lists and products also implement `==` and `!=`.) All of these return a `Bool` result.

Two other pseudo-comparison functions are `min` and `max`. These do what you'd expect, returning the minimum/maximum of their two arguments (which must be Ord-erable).

Built-in boolean operators are as you would expect:

```
&& || not
```

Note that `not` is just a normal one-argument function, not a special operator. All of these take `Bool` arguments, and return a `Bool` as well.

Signalling errors: If something does not make any sense whatsoever, you can throw an `error`: `error` is a built-in function of type `String -> a`. Note that it's return type is `a`, completely unspecified. This means that you can use `error` *anywhere*, in any type of expression. As soon as it is evaluated, the `error` will print the String you give it and then abort your program. E.g.,

```
x = 1 + error "Whoops!"
```

Evaluating x will cause the error to be thrown.

The other magical error value is `undefined`. We use this in labs to signal parts of the file which you are supposed to fill in. `undefined` has type a, so you can use it anywhere, but like `error`, attempting to evaluate it will abort your program and print an error message.

Control Structures

In a language like C/C++/Java, control structures are procedural in nature: they affect the order in which things happen. In Haskell, control structures are expressions: they return values.

if-then-else:

```
if x == 12 then "Hello" else "Goodbye"
```

The general form is

```
if condition then
  true_expression
else
  false_expression
```

The `condition` must be of type Bool, and both the `true_expression` and the `false_expression` must be of the same type. Note that if-then-else is "lazy": only one of `true_expression` and `false_expression` will be evaluated; the unused branch is not evaluated.

Note that since if-then-else is an expression you can do things like

```
12 + (if odd x then 1 else 2) * y
```

case:

```
case x of
  12 -> "Hello"
  _  -> "Goodbye"
```

(This has exactly the same effect as the example if-then-else above.) `case` is roughly Haskell's equivalent to `switch` in C/C++/Java. It solves the problem of nested if's becoming cumbersome, by allowing multiple branches. All of the patterns (to the left of the `->`) must have the same type, the type of x, and all of the return values (to the right of `->`) must have the same type.

Note that the "conditions" on each branch (12 and _ above) can actually be arbitrary patterns, so you can do something like

```
case l of
  [] -> 0
  (x:_) -> x
```

If you want to write a case on a single line, you'll have to use semicolons to separate the cases:

```
case x of 12 -> "Hello" ; _ -> "Goodbye"
```

As with if-then-else, `case` is an expression and can be used anywhere where a value or expression is needed:

```
1 + (case x of 'A' -> 0 ; 'B' -> 1 ) * z
```

let-in:

`let..in` is Haskell's version of local variable definitions, but with a significant twist. `let` lets you bind some names to some expressions (and do pattern-matching in the process, if you like), and thus is useful for either labeling some values according to their function, or abstracting out repeated calculations for efficiency:

```
let x = huge_calculation in x*x + x
```

Rather than perform the `huge_calculation` three times, we perform it once, call the result x, and then compute `x*x + x` (which would otherwise require *three* evaluations of `huge_calculation`). But again, `let..in` is still an expression, so you can do things like

```
x * (let x' = x + 12 in x*y) + z
```

`let..in` can bind more than one name:

```
let x = 12
    y = length "Potato"
    z = [1..]
in
    x + y + head z
```

Later names can refer to earlier ones:

```
let x = 12
    y = x + 1
    z = y ^ 2
in
    x + y + z
```

You can even use `let..in` to bind *functions* locally:

```
let f x = x^2
in
    f 5
```

Lists

Lists are so useful in Haskell that they have a number of different forms. The "cons" form of a list looks like this:

```
1:2:3:4:[]
```

Note that because the : operator associates to the *right*, this is equivalent to

```
1:(2:(3:(4:[])))
```

If you want to put a single element on the front of a list, you can "cons" it on:

```
1 : [2,3,4]
```

(try typing this into GHCi!) evaluates to

```
[1,2,3,4]
```

If you want to treat a list like a stack, then this is your "push" operation.

A lot of times we want to construct a list from a range of values. For example, `[1,2,3,4]` is the list of Int values between 1 and 4 (inclusive). We can write this more succintly as just

```
[1..4]
```

This will work with any element type that supports Enum. For example:

```
['a'..'h']
```

gives

```
"abcdefgh"
```

We can vary the "step" if we like:

```
[1,3..10]
```

gives

```
[1,3,5,7,9]
```

Note that if you want to count "down", you *must* provide a decreasing step:

```
[4..1]
```

gives

```
[]
```

What you really want is

```
[4,3..1]
```

We can even use this to construct *infinite* lists:

```
[1..]
```

gives the (infinite) list

```
[1,2,3,4,...]
```

Similarly,

```
[1,1..]
```

gives the infinite list of 1s:

```
[1,1,1,1...]
```

(A better way to construct an infinite list of a single value is to use `repeat`:

```
repeat 1
```

gives

```
[1,1,1,1...]
```

The difference is that `[a,a..]` requires the type of a to support Enum, while `repeat` can be used to repeat values of *any* type.)

Sometimes we want to build a list out of another list, by applying some operation to its elements. For example suppose we want the list of the squares of the integers from 1 to 4. I.e., we want to take `[1..4]` and from it square each element, producing `[1,4,9,16]`. We can do this with a *list comprehension*:

```
[x^2 | x <- [1..4]]
```

The left-hand side (to the left of the vertical bar) is the expression that is used to compute the elements of the new list. The right-hand side specifies where the original elements come from. So here, x will be bound to an element of `[1..4]`, x^2 will be computed, and the result saved in the corresponding position of the output list.

If we want, we can filter the values according to some criteria:

```
[x^2 | x <- [1..10], even x]
```

This will give the squares of only the *even* numbers between 1 and 10. But note that Haskell will keep "running" the list comprehension as long as the list generating x produces values. E.g., you might think you could do something like this:

```
[x^2 | x <- [1,2..], x <= 10]
```

and the list would stop after `x == 10`, but in fact it will run forever. Haskell doesn't know that, after `x == 10`, there won't, eventually, be another x that is <= 10, so it keeps on trying. Running forever makes Haskell sad; do you want to make Haskell sad?

You can also "drive" a list comprehension with more than one generator list:

```
[x+y | x <- [1,2], y <- [10,20]]
```

gives

```
[11,21,12,22]
```

(Can you see why?) You can think of this as a generalization of the notion of a "cross product" over all the input lists. Strangely enough, you can even "drive" a list comprehension with *no* generators:

```
[x | x < 10]
```

In this case, x must already be defined. If the condition is True, this will evaluate to `[x]`; if it is False it will evaluate to `[]`. Sometimes it may be useful to construct a zero-or-one element list, based on some condition; this is an easy way to do just that.

"Collapsing" lists: often we will want to take a list and collapse it down to a single value. For example, we might want to find the sum or product of a list of numbers, or maybe, given a list of Bools, determine whether they are all True. Haskell has a number of "aggregate" functions that do things like this:

- `sum` – Sums the elements of the list
- `product` – Finds the product of the elements of the list
- `maximum` – Returns the largest element of the list
- `minimum` – Returns the smallest element of the list
- `and` – Returns True if *every* element of the input list is True (i.e., it `&&`s all the elements of the input list together).
- `or` – Returns True if *any* element of the input list is True (i.e., it `||`s all the elements together)

Product Types

We mentioned tuples above and showed have they have a type built from `,`; the `,` type is called a *product type*. A product type can be thought of as somewhat like a `struct` in C/C++: it aggregates together multiple values of different types, but the overall *structure* (the component types, their number, and order) is fixed at compile time. A product type is like a `struct` in which the elements are unnamed, they just have their relative ordering:

```
(1,"Hello") -- A value of product type (Int,String)
```

Tuples are useful when you want to pass around multiple values as if they were a single object. For example, you can use a tuple to return two values from a function:

```
minMax :: [Int] -> (Int, Int)
minMax l = (minimum l, maximum l)
```

In the arguments to a function, you can pattern-match against a tuple to extract the components:

```
f :: (Int, String) -> Int
f (i,s) = i + length s
```

Although this looks like a "normal" function in C/C++/Java, do not be deceived (and don't write all your functions to take tuples, just because they look familiar). A tuple is still a single value, so we can do the following:

```
f x  -- Provided that x has a value of type `(Int, String)
```

But of course, we can also construct the required tuple on-the-fly, from values of the component types:

```
f(userid,username)
```

One useful function that combines tuples and lists is `zip`:

```
zip [1,2,3,4] "ABCD" -- Gives [(1,'A'), (2,'B'), (3,'C'), (4,'D')]
```

This is useful if you want to process two lists in parallel with each other:

```
[x+y | (x,y) <- zip [1..4] [4..8]]
```

(In this case, there is another function, `zipWith`, that handles the problem of pairing up elements of two lists and then applying some binary operation to them. E.g. this is equivalent to the previous:

```
zipWith (+) [1..4] [4..8]
```

Note that if one of the lists is longer that the other, then `zip` will only work to the end of the shorter lists. This means you can use an infinite list safely:

```
numberElems :: [a] -> [(a,Int)]
numberElems l = zip l [0..]
```

There are two builtin functions that operate on pairs:

- `fst` returns the first element of a pair
- `snd` returns the second element of a pair

Note that these *only* work on pairs: for higher-dimensional tuples you will have to use pattern matching. E.g.,

```
let (x,y,z) = triple_thing in ...
```

First-class Functions

Haskell is a *functional* language, which mostly means that functions exist as values: they can be stored in variables, passed into and returned out of functions, and even built-up from other functions. For example, we can do

```
plusone :: Int -> Int
plusone x = x+1

twice :: (Int -> Int) -> Int -> Int
twice f x = f(f(x))  -- Could also be written as f $ f x
```

We can now do something like

```
twice plusone 4
```

and the result will be 6 (i.e., `plusone(plusone(4))`).

There are a whole suite of builtin "higher order functions", functions that, like `twice`, take another function as an argument. Some examples:

```
map plusone [1,2,3,4]  -- Gives [2,3,4,5]
filter odd [1..10]     -- Gives [1,3,5,7,9]
foldr (+) 0 [1,2,3,4]  -- Gives 1+2+3+4+0 = 10
```

`foldr` can be thought of as performing a search-and-replace on a list. E.g., in the example above, the input list is `1:2:3:4:[]`. `:` gets replaced with `+`, while `[]` gets replaced with 0.

The `(+)` syntax for turning an operator into a "normal" function is actual just a fragment of the *sectioning syntax* that lets you leave off one argument to an operator and get back a function

```
(+1)  -- Same as the function plusone
(^2)  -- The function that squares its argument
(<10) -- Returns True if its argument is less than 10
(0==) -- Returns True if its argument is exactly 0
```

(But note that `(-1)` is not the function that decrements its argument, but just the literal numeric value -1. If you want the decrement function, you have to write `(+ (-1))`.) The comparison operator sections are useful with `filter`:

```
filter (>=0) list_of_numbers -- Keep only the positive values
```

Currying is an extension of sections to all functions, even those you write. It means that you can leave off the later arguments of a function, and you'll get back a new function. For example, suppose we have

```
f :: Int -> Float -> String -> Char
```

(where `a,b,c,d` are some types). If we call

```
f 1 3.5 "hello"
```

we will get back a Char. But if we call

```
f 1 3.5
```

we will get back a *function*, a function that takes a String and "finishes up", returning an Int. Similarly, if we leave off 3.5 we get a function that takes a Float and a String, and so forth. We can get a feeling for why this works by looking at the type of the function. In fact, the -> type associates *to the right*, so in reality the type is

```
f :: Int -> (Float -> (String -> Char))
```

I.e., f takes an Int and returns a function. That function takes a Float and returns a function. *That* function (finally!) takes a String and returns a Char. In reality, all Haskell functions are unary; they take only one argument. But later arguments will be automatically passed to functions that are returned, so we can "fake" multiple argument functions. This ability is what lies behind the otherwise inexplicable function call syntax:

```
f 1 2.2 "3"
```

makes more sense if you imagine that the call will actually proceed like

```
((f 1) 2.2) "3"
```

Lazy Evaluation

You may have heard that Haskell is a "lazy" language. As a way of introduction to what this means, take another look at the syntax for functions:

```
f x y z = ... -- three arguments
g x y = ...   -- two arguments
h x = ...     -- one argument
```

Under Haskell's syntax, what would a zero-argument function look like?

```
x = ...
```

In Haskell, a zero-argument function is indistinguishable from a variable. In particular, *using* a variable is semantically equivalent to "calling" a zero argument function. This means that definitions like this

```
x = x + 1
```

are perfectly valid. If you ever "call" x, then your program will go into an infinite loop, but the definition itself fine, albeit useless.

A more useful zero-argument function is something like this:

```
x = 1:x
```

In order to figure out what this means exactly, let's try to figure out the type. We know `1 :: Int`, and we also know that `(:) :: a -> [a] -> [a]`. Since the first argument to `:` is 1, `a` must be `Int`, which means that the type of the second argument (i.e., `x`) must be `[Int]`. So we actually have

```
x :: [Int]
x = 1:x
```

Let's evaluate out a couple of terms. After substituting the definition of `x` into its body once we have

```
x = 1:(1:x)
```

Do it again and we get

```
x = 1:(1:(1:x))
```

In fact, x is the (lazy) infinite list of 1s. Each occurence of x within the ever-expanding definition will be evaluated lazily; not when it is used, but only when it is actually needed. This is how we can deal with infinite lists. We can use the built-in `take` function to get a fragment of the list safely:

```
take 5 x  -- will output [1,1,1,1,1]
```

Typeclasses

A Haskell typeclass is roughly akin to an interface in Java, or an abstract base class in C++. It defines a set of operations, but does not specify how they are implemented. For example, any type that supports the `Eq` typeclass supports both equality (==) and inequality (/=) but the actual implementation of these operators is left up to the type.

The most useful typeclasses to know about are

- Eq – supports (in)equality
- Ord – supports comparison operators
- Num – supports artithmetic operators (implies support for Eq and Ord as well)
- Show – supports conversion to String (i.e., for printing)
- Enum – supports enumeration over a range (i.e., we can ask for all the values of this type between a and b). The list syntax `[a..b]` requires that the type of a and b support Enum.

Note that product and list types support Eq, Ord, and Show, provided that the component types support them. I.e., because Int supports Ord, so does `[Int]`, so we can do:

```
[1,2,3] < [3,4,5]
```

This kind of comparison is done *lexicographically*; the first two components are compared, if they are equal then the second two, and so forth. (This is the kind of comparison you would do when looking a word up in the dictionary.)

In the type of a function, any type classes are shown before a `=>`:

```
f :: Eq a => [a] -> Bool
f (x1:x2:_) =   x1 == x2
```

We won't ask you to write functions with typeclass constraints, however. Knowledge of typeclasses is mostly useful for when you want to *look up* a function; most Haskell functions are polymorphic, so although you might expect to see a function of type `Int -> Int` it will probably have a type more like `Num a => a -> a` so that it works on any numeric type.

Combinatorics

Combinatorics is a fancy computer science word for "counting stuff". For example, we might want to know:

1. How many different ways can we roll a six-sided die?
2. How many different ways can we roll a pair of six-sided dice (assuming we roll them in order and keep track of which was rolled "first" and which was rolled "second")?
3. If we draw four cards from a standard 52-deck, how many ways are there of drawing all the same suit (i.e., all twos, all queens, etc.)?
4. How many bijections are there on a set of cardinality 10?

To take (1) as an example, if we write down all the possible rolls, the answer is obvious.

It's obvious even *without* writing them down, but bear with me.

There are *six* possible ways we can roll a six-sided die; that is, on a single roll, the value shown will be one of the six listed above.

Although this result appears trivial, it illustrates a fundamental principle which can always be used to solve any combinatorics problem: write down all the possibilities and count them. There may be millions, or billions of possibilities, but provided you write them down, not missing or duplicating any, your answer will always be correct. Generally speaking, we will consider our combinatorics problems from two angles:

Of course, in reality you wouldn't want to write them down, we'd prefer to write a program to find all the possibilities and count them.

- Numerically, as purely a question of developing equations for counting the number of possibilities in various settings.

- Algorithmically, as methods for *generating* all possibilities (which can then be counted, or otherwise manipulated, easily).

For many problems, developing the mathematical structure will be the hard part; generating the possible results algorithmically will be easy, and thus will serve as a useful method for "checking our work". In some cases, however, generating the possibilities will require some finesse on our part, in which case our mathematical predictions will serve to check our code.

Rules of Product and Sum

Consider example (2) above: although we could write down all the possibilities, we're already getting to the realm where that method would be tedious. Instead, we'll try to reason out a general mathematical method for solving these kinds of problems.

- Suppose we roll a ⚀ on the first die; since we have two dice and they don't affect each other at all, that means that the second die could still roll any of its 6 possible values. So we know that we have *at least* 6 possible answers.
- Suppose we roll a ⚁ on the first die; again, we could roll any of 6 values on the second. So now we have *another* 6 possible answers, to be added to the first 6. So now we have 12 possible answers that we know about.
- Continuing the pattern for ⚂, ⚃, etc., for every roll of the first die, there are always 6 possible answers for the second. If we were to add them all up we would get $6 + 6 + 6 + 6 + 6 + 6 = 6 \cdot 6 = 36$ possible results.

  This illustrates the *rule of product*:

Definition 0.1 (Rule of Product) If we have two distinct, independent events, the first of which can have m possible results, and the second which can have n possible results, then the number of possible results of these two events happening in combination is

$$m \cdot n$$

This generalizes to more than two events: if we have three distinct, independent events happening in combination then we can simply multiply the number of results for all the events. In general, if we have n events where the number of results for the i-the event is e_i then the number of results if all the events happen together is

$$\prod_{i=1}^{n} e_i$$

Note that the conditions "distinct" and "independent" are important:

- If the events are not distinct (can "overlap" in some way) then we may end up with *fewer* results than the product rule would predict. For example, suppose we rolled a pair of dice but did *not* keep track of which was first and

which was second. That is, ⚀ ⚃ would count as indistinguishable from ⚃ ⚀. Now there are fewer than 36 possibilities, because, as we've just seen, some pairs of possibilities count as only one results.

- If the events are not independent (the first can influence the number of resuls in the second) then all bets are off: we may end up with more, or fewer, results than the product rule would predict. For example, suppose we drew two cards from a standard 52-card deck, and we *did not* put the first card back in the deck after drawing it but placed it aside. The product rule would tell us that we have $52 \cdot 52 = 2074$ possible results, but this is not correct. After drawing the first card, there are only 51 cards left in the deck, so the answer is actually $52 \cdot 51 = 2652$ possible results.

(We'll cover both these variants in the next section.)

A specific variant of the product rule covers the situation in which we have a single event with e results, repeated r times. In this case, we have

$$\underbrace{e \cdot e \cdot \ldots e}_{r} = e^r$$

Haskell note:

In Haskell, to generate all possibilities of e_1 and e_2 together, we simply use the list comprehension notation to take their *cross product*:

```
d6 = [1..6]
d10 = [1..10]

ghci> length [(x,y) | x <- d6, y <- d10]
60
```

The generated list will contain all possible pairings of values from a d6, with values from a d10.

Example 0.1 *The cafeteria has implemented a sadistic new meal plan. You may choose:*

- *A hot meal with a cold drink and a dessert.*
- *A cold meal with either a hot or cold drink.*

The possible hot meals are soup, "meat" (of unspecified origin), grilled cheese, and pasta. The possible cold meals are three types of sandwiches, potato salad, or green salad. The possible desserts are cake, a cookie, or jello. The possible hot drinks are coffee and tea; the possible cold drinks are water, milk, four types of soda, and lemonade. How many possible meals are there?

We will break down the problem by first considering all the possibilities for

each component of the meal:

$$\begin{aligned}
\text{Hot meals} &= \{\text{soup, meat, grilled cheese, pasta}\} \\
\text{Cold meals} &= \{\text{sandwhich}_1, \text{sandwhich}_2, \text{sandwich}_3, \\
&\qquad \text{potato salad, green salad}\} \\
\text{Desserts} &= \{\text{cake, cookie, jello}\} \\
\text{Hot drinks} &= \{\text{coffee, tea}\} \\
\text{Cold drinks} &= \{\text{water, milk, soda}_1, \text{soda}_2, \\
&\qquad \text{soda}_3, \text{soda}_4, \text{lemonade}\}
\end{aligned}$$

So we have

$$|\text{Hot meals}| \cdot |\text{Cold drinks}| \cdot |\text{Desserts}| + |\text{Cold meals}| \cdot (|\text{Cold drinks}| + |\text{Hot drinks}|)$$

$$= 4 \cdot 7 \cdot 3 + 5(2 + 7) = 84 + 45 = 129$$

possible meals.

I'll admit: I had fun writing this one.

Suppose we have a six-sided die, and a 12-sided die, and you are given the *choice* of which to roll: one or the other, but not both. Now how many possible results are there?

- If you choose to roll the d6, then there are 6 possible results.
- If you choose to roll the d12, then there are 12 possible results.
- So in total there are 6 + 12 = 18 possible results if you roll one or the other.

Definition 0.2 (Rule of Sum) If we have two distinct events the first of which can have m possible results, and the second which can have n possible results, then the number of possible results if one *or* the other of these events (but not both) occurs is

$$m + n$$

Again, this generalizes to n events:

$$\sum_{i=1}^{n} e_i$$

Again, it's important that the events be distinct: if they overlap in some way, we will have fewer results than the sum rule would suggest. For example, suppose we want to know how many ways we could roll a d6 such that the result is either a ⚀ or odd. Applying the sum rule, there is 1 way to roll a ⚀, and 3 ways to roll an odd number (⚀, ⚂, and ⚅), so there should be 1 + 3 = 4 possible results, but this is not correct. If we write down all the possible rolls that are ⚀ or odd, we get

⚀ ⚂ ⚅

Since ⚀ occurs in *both* of the events, we have to be careful to avoid counting it twice, so there are only 3 possible results.

Haskell note:

To implement the choice of e_1 and e_2, we have to consider the types of their events. If they have the same type, then we can simply append the two event lists together (using `distinct` to eliminate duplicates, if necessary or desired):

```
ghci> d6 ++ d10
[1,2,3,4,5,6,1,2,3,4,5,6,7,8,9,10]
```

However, this strategy is problematic, because it cannot handle events of different types, and even for events of the same type, it discards the useful information about which possibility came from which event.

For events of different types we can invent a new data type to encapsulate both types, and also to differentiate between the two sources:

```
data D6orD10 = D6 Int | D10 Int
    (deriving Eq, Show)


ghci> (map D6 d6) ++ (map D10 d10)
[D6 1, D6 2, D6 3, D6 4, D6 5, D6 6,
 D10 1, D10 2, D10 3, D10 4, D10 5, D10 6, D10 7, D10 8, D10 9, D10 10]
```

In fact, Haskell has a built-in type called `Either`, with its two constructors `Left` and `Right` which can be used if we only have two types to deal with:

```
ghci> (map Left [1..3]) ++ (map Right "abc")
[Left 1, Left 2, Left 3, Right 'a', Right 'b', Right 'c']
```

We can combine the rules of sum and product to solve some problems:

Example 0.2 *You have the choice of rolling a d6, or of rolling a d8 and a d10.*

There are

$$6 + 8 \cdot 10 = 86$$

possibilities.

Or in Haskell:

```
ghci> length $ (map Left d6) ++ (map Right [(x,y) | x <- d8, y <- d10])
86
```

Example 0.3 *Roll a d10; if you roll a 10, roll again and add the result to your total.*

Here, there are 10 possibilities at the top level, but one of them expands into another 10. Hence, we have

$$9 + 10 = 19$$

possibilities. Incidentally, the complete set of possible results is

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\}$$

Note that it is not possible to roll a 10!

To do this in Haskell, we'll replace each of the values 1 ... 10 with a list of all the possibilities that it generates. For 1 ... 9 this will just be the value itself, but for 10 it will be `[10 + x | x <- d10]`. Then we'll `concat` the resulting nested list so we can count the total number of elements:

```
ghci> length $ concat [if x == 10 then map (x+) d10 else [x] | x <- d10]
19
```

Permutations

Suppose we draw five cards from a standard deck of 52 cards. Furthermore, we distinguish the first card from the second, third, etc. I.e., rather than treating the five cards as a "hand", usable in any order, the order is fixed when we draw them. We'll also assume that we draw the cards from the middle of the deck, so that they are drawn randomly. How many different 5-card arrangements can we draw?

If we examine the first card, we have 52 choices, since it can be any of the 52 cards in the deck. However, when we draw our second card, there are only *51* cards remaining in the deck; it is not possible for us to draw the same card twice. If we carry this process on to the fifth card, we will find that there are

$$52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 = 311875200$$

possible arrangements.

A problem like this, where we draw our choices from a collection *without replacement*, so that once an item is chosen it cannot be *re*-chosen, is called a *permutation*. In particular, we would call this a permutation of size 5 for 52 objects. We can generalize this to permutations of size r over n objects as

$$P(n, r) = n(n-1)(n-2)\ldots(n-r+1)$$

We can simplify this definition if we define the factorial:

$$n! = n(n-1)(n-2)\ldots(2)(1)$$

By definition, $0! = 1$.

Then our definition of $P(n, r)$ becomes just

$$P(n, r) = \frac{n!}{(n-r)!}$$

Permutations with repetition: Suppose we want to find the permutations of the letters in the word BANANA. Since there are 6 letters, we might assume that there are $6! = 720$ permutations, but this is incorrect. Because some letters

are *repeated*, there are fewer permutations. There are only 720 permutations if we distinguish the three A's, and the two N's; in fact, if we are looking for the permutations of $BA_1N_1A_2N_2A_3$. *With* repetition, we have to eliminate the duplicate permutations in order to get an accurate count. For example, BAN_1AN_2A and BAN_2AN_1A are two indistinguishable permutations. In fact, for *any* arrangement of the letters in BANANA, there will be another version in which N_1 and N_2 are swapped. Looking only at the N's, there are twice as many permutations as there should be.

A similar reasoning applies to the A's: for any particular arrangement, there are 3! arrangements of the "different" A's that are indistinguishable. Thus, the true number of permutations is actually

$$\frac{6!}{2!\,3!} = 60$$

Example 0.4 *How many arrangements of BANANA are there in which all three A's occur together, as AAA?*

To figure this out, we simply treat AAA as a *single*, indivisible symbol and permute it along with the other letters B, N, and N. So we have

$$\frac{4!}{2!} = 12$$

In general, if we have some collection of n items broken into r different *types* (with items of the same type indistinguishable), with n_1 of the first type, n_2 of the second, etc. (and $n_1 + n_2 + \ldots + n_r = n$) then there are

$$\frac{n!}{n_1!\,n_2!\ldots n_r!}$$

arrangements possible.

Example 0.5 *Six guests, $g_1 \ldots g_6$ are seated around a circular table. Because the table as circular, we consider two seating arrangements to be the same if they are rotationally equivalent; that is, if we could rotate one arrangement to get the other. How many seating arrangements are there?*

Here we have the number of permutations of 6 items taken in size 6, but there are 6 possible rotations, so we have

$$\frac{P(6,6)}{6} = \frac{6!}{6} = 120$$

possible arrangements.

Generating permutations in Haskell

Although there is a function `permutations`, defined in `Data.List`, it requires $n = r$; i.e., it generates the "full" $n!$ permutations. We are interested in a) allowing $r \leq n$ and b) implementing it ourselves, to see how its done.

We will define a function `perms l r`. The output of `perms` will be the collection of all permutations of the list l of size r, thus, it will always be the case that `length (perms l r)` = P(`length l`, r),

Note that, for the moment, we will assume that the input is duplicate-free. Why this will be an issue will become apparent later.

We will define `perm` recursively on r. Our base case is $r = 0$, which gives

```
perms _ 0 = [[]]
```

(Note that this satisfies our requirement above that $P(n, 0) = 1$.)

In our recursive case, we are given `perms l (r-1)` and wish to construct from it `perms l r`. As an aid to working out exactly what needs to happen, here is the result of `perms [1..4] 1`:

```
[[1],[2],[3],[4]]
```

and here is the result of `perms [1..4] 2`:

```
[[2,1],[3,1],[4,1],
 [1,2],[3,2],[4,2],
 [1,3],[2,3],[4,3],
 [1,4],[2,4],[3,4]]
```

There are four rows, which suggests that perhaps each row corresponds to an element from `perms [1..4] 1`?

- For the first row, we take `[1]` and cons onto it each of 2,3 and 4. 1 is skipped, because 1 is already "used up" in `[1]`.
- For the second row, we take `[2]` and cons onto it each of 1,3,4, again, skipping 2 because it is already "used up".
- etc.

It would seem that the general rule is to take each element of the recursive output p', and cons onto it each element of $l - p'$. That is, we remove all the "used up" elements from l.

Implementing all of this leaves us with:

```
import Data.List ((\\))

perms :: Eq a => [a] -> Int -> [[a]]
perms _ 0 = [[]]
perms l r = [i:p | p <- pm', i <- l \\ p]
    where
        pm' = perms l (r-1)
```

(We import the "list difference" operator `\\` from `Data.List` to find the difference $l - p'$.)

Combinations

Suppose we draw a hand of five cards, but instead of distinguishing *first*, *second*, etc., cards, we regard the hand as unordered. *With* ordering we know that we have

$$P(52,5) = \frac{52!}{(52-5)!} = 311875200$$

permutations. But if order doesn't matter then the hand

AH, KD, 9C, 3H, 2S

is equivalent to

KD, 2S, AH, 9C, 3H

In fact, if we consider a given hand of five cards and how we could reorder it, we see that for the first card we have a choice of 5 cards, for the second 4, and so forth. In fact, the total number of equivalent hands is simply the number of permutations of 5. This brings us to the *number of combinations* of n elements of size r:

$$C(n,r) = \binom{n}{k} = \frac{P(n,r)}{r!} = \frac{n!}{r!\,(n-r)!}$$

$C(n,r)$ is sometimes read as "n choose k". For the above example, we have

$$C(52,5) = \frac{52!}{5!\,(52-5)!} = 1105$$

Example 0.6 *On your midterm, you determine that you only have time to complete 15 problems, out of the 20 given on the test. How many different combinations of problems are there?*

The order in which you choose the problems does not matter, so combinations are the tool of choice here. We have

$$C(20,15) = \frac{20!}{15!\,(20-15)!} = 15504$$

combinations.

Example 0.7 *Suppose we roll two 6-sided dice, and we ignore order (i.e., (⚀,⚅) is considered identical to (⚅,⚀)). How many possible rolls are there?*

This problem is subtle; a first (and incorrect) guess might be

$$\frac{6^2}{2!} = 18$$

but this is wrong, because it removes too many "duplicates". The rolls (⚀,⚀), (⚁,⚁), etc. have no duplicates. In fact, we need to first determine the number of rolls in which the two values are *different*, divide that by 2!, and then add in the 6 double-rolls. Thinking about the rolls where they are different, we see that this is an 6-choose-2 situation (for the first roll we have 6 choices, but that

eliminates a choice for the second roll, so then we only have 5). So the number of rolls where the values are different, ignoring order is

$$C(6,2) = \frac{6!}{4!\,2!} = 15$$

plus the 6 double-rolls gives 15 + 6 = 21.

Another way to think about this problem is counting the number of rolls of 2d6 in which the first value is ≤ the second.

Combinations with repetition: We have assume that, in selecting combinations, items are "used up". That is, when we choose $C(n,r)$ after choosing the first item, there are $n-1$ items available. What if this is not the case, what if we have *n types* of items, but an unlimited supply of each? We call this the problem of selecting combinations *with repetition*.

Example 0.8 *Suppose seven students go to a restaurant where they have the choice of four items: cheeseburger, hot dog, taco, and fish sandwhich. If each student gets one item, how many possible* purchases *are there?*

In looking at purchases, we only care about how many of each item were ordered, not who got what. For each of the four items, we have a quantity from 0 to 4, and the total of all the quantities must add up to 4. We will illustrate an order as

```
CCHHHTF
```

(for an order of 2 cheeseburgers, 3 hotdogs, a taco, and a fish sandwich).

Thinking about this representation a bit, we see that as long as the different "sections" always contain the same items (i.e., the first section is always cheeseburgers, the second is always hotdogs, etc.) we don't need to keep track of *which* items are in each section, just how many. We can thus illustrate an order equivalently using Xs and separators:

```
XX|XXX|X|X
```

Note that we have one fewer dividers than the number of items (4 − 1 = 3) because three dividers split the order into four groups. We have to remember that the first group is always cheeseburgers, etc.

In fact, we now have the problem is counting permuatations with duplicates: we have 7 X's, and 3 separators, for a total of 10 items. So we have

$$\frac{10!}{3!\,7!}$$

is the total number of possible orders!

If we generalize this to the number of combinations of r of n distinct items, with repetition, is

$$\frac{n+r-1}{r!\,(n-1)!} = C(n+r-1,r)$$

Derangements

A permutation is a re-ordering of the elements in a sequence. What if we place the restriction that *no item is in its original position*? We call such an arrangement a *derangement*, and counting them is the focus of this section.

Suppose we have n items, and we choose the first, item 1, to reposition. Item 1 cannot be placed back into slot 1, so we have a choice of $n - 1$ positions:

$$D(n) = (n - 1) \cdot \ldots$$

Let us suppose that item 1 is placed into slot i (with $i \neq 1$, obviously). We now choose item i to reposition next. Here, we have two choices:

- We can choose to place item i in slot 1, effectively swapping items 1 and i. After doing so, we have $n - 2$ items to derange, and $n - 2$ slots into which to place them, so the problem is of the same kind as our original derangement problem, just smaller. So we have

$$D(n) = (n - 1)(D(n - 2) + \ldots)$$

- Alternatively, we can declare that item i's forbidden slot is slot 1, but still allow some other item to be placed there. Now, including i (which in this case we have *not* placed yet) we have $n - 1$ items, and $n - 1$ slots (including slot 1, which has not been filled yet, but can be filled by any item but i). So again, we have the smaller problem of deranging $n - 1$ items into $n - 1$ slots, leading us to the full definition:

$$D(n) = (n - 1)(D(n - 2) + D(n - 1))$$

The sequence $D(n)$ is called the *subfactorial*, and is usually written $!n$. I.e.,

$$!n = (n - 1)(!(n - 2) + !(n - 1))$$

The Inclusion-Exclusion Principle

The Inclusion-Exclusion principle is useful for solving problems where we have some large collection of objects, and we want to count all those that *do not* have property p_1 *or* p_2 or ….

Example 0.9 *Count the number of positive integers ≤ 100 that are* not *divisible by 2, 3, or 7.*

Note that this is different from from counting those that are not divisible by 2, 3, *and* 7. If we wanted that, it would be the same as asking for all the numbers that are not divisible by $2 \cdot 3 \cdot 7 = 35$.

Counting the number of integers that are divisible by some n is easy: just divide and round down. Likewise, if we want to know the number that are *not* divisible, we can simply subtract our result from the total number:

$$\text{not divisible by } 3 = 100 - \lfloor \frac{100}{3} \rfloor = 100 - 33 = 67$$

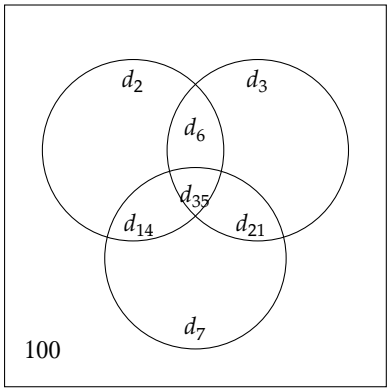
However, we cannot simply do

$$100 - d_2 - d_3 - d_7$$

(where d_n is the number of integers ≤ 100 that are divisible by n), because some integers are divisible by 2 *and* 3, or by 2 and 7, by 3 and 7, and by 2,3, and 7. In fact, if we simply subtract, we will be removing too many elements.

Way too many, in fact: $100 - d_2 - d_3 - d_7 = 100 - 50 - 33 - 14 = 3$! Surely there are more than that!

Graphically the problem is this:



If we remove one "copy" of the regions d_2, d_3 and d_7, we will be accidentally removing *two* copies of d_6, d_{14}, and d_{21}, and *three* copies of d_{35}.

We can partly remedy this situation by adding back in a single copy of d_6, d_{14}, and d_{21}:

$$100 - (d_2 + d_3 + d_7) + (d_6 + d_{14} + d_{21}) \ldots$$

But again, we've gone too far: because each of d_6, d_{14}, and d_{21} includes a copy of d_{35} we have removed three copies of d_{35}, so now we have *zero* copies of d_{35}. We can fix this and complete the calculation by subtracting a copy of d_{35}:

$$100 - (d_2 + d_3 + d_7) + (d_6 + d_{14} + d_{21}) - d_{35}$$

If we calculate out all the needed values, we find that we have

$$100 - (50 + 33 + 14) + (16 + 7 + 4) - 2 = 28$$

If we have four conditions, then a similar procedure applies: subtract out copies of each condition by itself, add in copies of the conditions in pairs, subtract them in triples, and then add in all four in combination:

$$\begin{aligned} \text{total} &- (d_a + d_b + d_c + d_d) \\ &+ (d_{ab} + d_{bc} + d_{cd} + d_{ac} + d_{ad} + d_{bd}) \\ &- (d_{abc} + d_{bcd} + d_{abd} + d_{acd}) \\ &+ d_{abcd} \end{aligned}$$

(Notice, also, that the first group consists of all combinations of 4 items taken *1* at a time, the second group *2* at a time, and so forth, with the signs alternating $-, +, -, +, \ldots$.)

Probability

With *probability* we are concerned with knowing *how often* some particular condition is fulfilled, out of some collection of events. We will re-use much of the previous section's work. In particular, we will still look at collections of *outcomes*, however, now we will highlight some particular outcomes of interest. If we have n total outcomes, and there are $m (\leq n)$ outcomes that are of interest to us, then we say that the *probability* of these occuring is m/n. This general definition will underline everything we do in this section.

This presupposes the *frequentist* interpretation of probability. The frequentist interpretation assumes that probability reflects how often an event occurs, out of the total number of times it possibly could occur. For closed systems of outcomes like ours, the frequentist model is a good one to use. It breaks down when we start to ask questions like "what is the probability that it will rain tomorrow?", since tomorrow only occurs once.

We will call a collection of "interesting" outcomes an *event*, and identify events as $e_1, e_2, \ldots$. An example of an event in the space of "rolling two d6s" might be "rolling snake-eyes" or "rolling two odd values". We will call the space of all possible outcomes Ω. We define the *probability function* P as

$$P(e) = \frac{|e|}{|\Omega|}$$

Note that $P(\Omega) = 1$, and that $0 \leq P(e) \leq 1$ for any e (the smallest e is the empty set of outcomes; the largest e is $e = \Omega$).

Combining events

Suppose we have two events e_1 and e_2. If we know $P(e_1)$ and $P(e_2)$, can we deduce $P(e_1 \cup e_2)$ or $P(e_1 \cap e_2)$? Maybe:

- If e_1 and e_2 are *disjoint* (that is, if $e_1 \cap e_2 = \varnothing$) then Yes. We have

$$P(e_1 \cup e_2) = P(e_1) + P(e_2) \qquad P(e_1 \cap e_2) = 0$$

- If e_1 and e_2 overlap in some way then No, we need more information. If e_1 and e_2 are not disjoint, then $P(e_1 \cap e_2) > 0$. If we are given $P(e_1 \cap e_2)$ then we have

$$P(e_1 \cup e_2) = P(e_1) + P(e_2) - P(e_1 \cap e_2)$$

  (Note that this definition is consistent with the previous when e_1 and e_2 are disjoint.)

If this looks kind of like the Inclusion-Exclusion principle, that's because it is!

Haskell note:

In Haskell, an event will be a function from outcomes to `Bool`. E.g., if we are dealing with the space of 4d6, an event will have type `[Int] -> Bool` (or possibly `(Int,Int,Int,Int) -> Bool`). We provide a function `prob e s` that returns the probability of event e occuring over the outcome space s.

We provide several higher-order operators that allow event functions to be combined in logical ways:

- e_1 `.&&` e_2 – Returns True if both e_1 and e_2 do.
- e_1 `.||` e_2 – Returns True if either of e_1 or e_2 do.

- e_1 `.><` e_2 – Returns True if only one of e_1 or e_2 do.
- `not . e` – Returns True if e is False and vice versa (this is just the normal `not` operator composed with e, using Haskell's function composition operator `.`).

Thus, using these operators, we could ask for the probability that a d20 roll will be odd and ≥ 6:

```
prob (odd .&& (>=6)) d20
```

or that a d6 will roll an even number, or a 4:

```
prob (even .|| (==4)) d6
```

Another (built-in) function which may be useful is `uncurry`:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
```

`uncurry` takes a normal two-argument operator (such as `(+)`) and converts it into a function that takes a *pair* as its input. This allows us to do something like

```
prob (uncurry (<)) [(x,y) | x <- d6, y <- d10]
```

This will find the probability that a d10 roll will beat a d6 roll.

(Those who have seen probability elsewhere will note that we are making a number of simplifications: we are assuming that each outcome occurs with equal probability, and that any variations are manifested in the *number* of duplicates of each outcome. E.g., an outcome that occurs twice is twice as likely as one that occurs once. This also implies that outcomes with 0 probability are not represented at all; i.e., all of our probability distributions are "positive".)

Haskell note:

Because we represent Ω as a list of outcomes in Haskell, it is impossible for any "primitive" event (i.e., an event that consists of a single actual outcome) to have probability 0.0. In order for an outcome to have probability 0.0, it would have to not occur in Ω at all, and in that case, it is effectively not an outcome at all! In probabilistic terms, we would say that in Haskell, all our distributions are restricted to their *supports*.

Some identities

Although the exact distribution of outcomes will determine their probabilities, we can still state some general identities that will always apply. We have already

seen two, above:

$$P(\Omega) = 1$$
$$P(\varnothing) = 0$$

That is, the probability of absolutely anything happening, we don't care what, is 1.0, and the probability that absolutely nothing at all will happen is 0.0.

Because "events" are really just set-like collections, we can apply a number of results from Boolean logic and set theory. E.g., if we denote the probability that *e does not* happen as $P(\neg e)$ then we have

$$P(\neg e) = P(\Omega - e) = 1 - P(e)$$

(Note that $\Omega - e$ is a *set difference*, while $1 - P(e)$ is normal arithmetic subtraction.)

Similarly, we can say that the probability of two things not happening is the same as the probability of neither of them happening:

$$P(\neg(e_1 \cup e_2)) = P(\neg e_1 \cap \neg e_2)$$

(Where $\cup$ can be read as "and" and $\cap$ can be read as "or".) Similarly, the probability of both of two things not happening is the same as the probability of either of them not happening:

$$P(\neg(e_1 \cap e_2)) = P(\neg e_1 \cup \neg e_2)$$

Finally, there is a kind of distributive law, akin to arithmetic's $a(b + c) = ab + ac$, that allows us to distribute $\cap$ over $\cup$ and vice versa:

$$P(e_1 \cup (e_2 \cap e_3)) = P((e_1 \cup e_2) \cap (e_1 \cup e_3))$$
$$P(e_1 \cap (e_2 \cup e_3)) = P((e_1 \cap e_2) \cup (e_1 \cap e_3))$$

We mentioned above that $P(e_1 \cup e_2) = P(e_1) + P(e_2) - P(e_1 \cap e_2)$. This implies that

$$P(e_1 \cup e_2) \leq P(e_1) + P(e_2)$$

The Inclusion-Exclusion Principle for Probability

There is a variation of the inclusion-exclusion principle for probability. In fact, we've already seen one version of it

$$P(e_1 \cup e_2) = P(e_1) + P(e_2) - P(e_1 \cap e_2)$$

Again, the idea is the same: if we simply add $P(e_1)$ and $P(e_2)$ we will be double-counting any events that occur in both, so we subtract out one copy of the events that happen in both. If we have three events we get the familiar sign-switching expansion:

$$\begin{aligned} P(e_1 \cup e_2 \cup e_3) = {} & P(e_1) + P(e_2) + P(e_3) \\ & - (P(e_1 \cap e_2) + P(e_1 \cap e_3) + P(e_2 \cap e_3)) \\ & + P(e_1 \cap e_2 \cap e_3) \end{aligned}$$

We can use the inclusion-exclusion principle here exactly as we did in combinatorics.

Example 0.10 *Suppose we roll 2d10, and we want to know the probability that* either *die rolled a 2.*

Let's use the notation $d_1 = 2$ to indicate that die 1 rolled a 2. By the inclusion-exclusion principle, we have

$$P(d_1 = 2 \cup d_2 = 2) = P(d_1 = 2) + P(d_2 = 2) - P(d_1 = 2 \cap d_2 = 2)$$

The probability that a d10 will roll *any* particular number is simply 0.1. The probability that *both* will roll a particular number is $\frac{1}{10 \cdot 10} = 0.01$ so we have

$$P(d_1 = 2 \cup d_2 = 2) = \frac{1}{10} + \frac{1}{10} - \frac{1}{100} = 0.19$$

This is slightly less than the $\frac{2}{10}$ that a naive approach would predict. The reason is because a roll of (2,2) does not count as rolling 2 *twice*, but only once. Note, also that the probability is the same for rolling a 3 on either, or a 4, or any other number.

Product distributions

A probability distribution over some Ω represents a single "experiment". This is true even if the experiment involves doing multiple things (e.g., rolling multiple dice). What if we have multiple experiments? We can form the *product distribution* over two outcome spaces Ω_1 and Ω_2 as

$$\Omega_1 \times \Omega_2 = \{(e_1, e_2) \mid e_1 \in \Omega_1, e_2 \in \Omega_2\}$$

Events in this space have the form of pairs of events from the two underlying experiments. We can then give the product probability function as

$$P((e_1, e_2)) = P_{\Omega_1}(e_1) P_{\Omega_2}(e_2)$$

It should be obvious that we can form product distributions of more than just 2 probability distributions. One interesting application is *repeated* experiments: if we perform the same experiment twice we have

$$P_{\Omega \times \Omega}((e, e)) = P_{\Omega}(e)^2$$

Likewise, if we repeat an experiment n times we have

$$P((e, e, \ldots, e)) = P(e)^n$$

the probability that event e will occur n times in succession.

Example 0.11 *What is the probability that an unweighted coin flipped 10 times will land heads every time?*

Suppose we designate "heads" as 1 and "tails" as zero. Then the probability distribution is

$$P(c = 1) = \frac{1}{2}$$

and we have the product distribution

$$P(c = 1)^{10} = \frac{1}{2^{10}} = \frac{1}{1024} \approx 0.00098$$

In fact, this is the probability that any *particular* configuration of 10 flips will come up. E.g., if we asked for the probability that the first 5 flips would land heads, and the last 5 tails, it would be the same.

We could just as well have considered the probability distribution of 10d2 "rolls". In that case, each outcome would be a 10-tuple of coin flips. Often we will have the choice of whether to construct a single distribution with complex structure, or to combine a number of simpler distributions into a product distribution. We'll usually make the choice based on what's easiest. The nice thing about a product distribution is that we are *guaranteed* that the events in the product cannot "interfere" with each other, because they came from completely different "experiments".

Example 0.12 *Suppose we have a* weighted *coin that lands heads some* $0 < q < 1$ *portion of the time, and tails* $1 - q$ *the rest of the time. Now what is the probability that 10 flips will all be heads? Will all be tails?*

Here we have

$$P(c = 1)^{10} = q^{10}$$

and

$$P(c = 0)^{10} = (1 - q)^{10}$$

Note that if we were looking for the probability of some specific configuration of heads/tails, we would have something like

$$q \cdot (1 - q) \cdot q \ldots (1 - q) \cdot q$$

But since multiplication is both associative and commutative we can rearrange the product into

$$q^m \cdot (1 - q)^n$$

where m is the number of heads and n is the number of tails and $m + n = 10$.

Conditional Probability

Often we will want to look at the probability of some event e occuring *assuming* that some other event e' is already known to have occured. For example, the probability of rolling a 1 on a d6 is $\frac{1}{6} = 0.167$ but the probability of rolling a 1 *assuming* we have rolled an odd number is $\frac{1}{3} = 0.333$. We write the conditional probability of e *given* e' as $P(e \mid e')$ and define it to be

$$P(e \mid e') = \frac{P(e \cap e')}{P(e')}$$

This also gives us an alternate definition of $P(e_1 \cap e_2)$:

$$P(e_1 \cap e_2) = P(e_1 \mid e_2)P(e_2)$$

Haskell note:
In Haskell we have the function `when`:

```
when :: (a -> Bool) -> (a -> Bool) -> [a] -> Float
```

where `when e f l` gives the conditional probability that event f occurs in l, *given* that e is known to have occured. Internally, it works by simply filtering l to only those outcomes where e is True, and then finding the probability of f within that restricted context.

Exercise: prove that this implementation gives an equivalent result to the above mathematical definition.

The Chain Rule: We can extend the above definition of $P(e_1 \cap e_2)$ to three events as follows:

$$P(e_1 \cap e_2 \cap e_3) = P(e_1 \mid e_2 \cap e_3)P(e_2 \mid e_3)P(e_3)$$

or

$$P(e_1 \cap e_2 \cap e_3) = P(e_1 \mid e_2 \cap e_3)P(e_2 \mid e_3)P(e_3)$$

Bayes Law: Looking again at the definition of conditional probability:

$$P(e \mid e') = \frac{P(e \cap e')}{P(e')}$$

With a bit of algebraic massage we can get

$$P(e \mid e') = \frac{P(e \cap e')}{P(e)} = \frac{P(e \mid e')P(e')}{P(e)}$$

I.e., given the conditional probability going in one direction, we can find the reverse probability, provided we know the unconditional probabilities of the underlying events.

Independence: we say that two events are *independent* if

$$P(e_1 \cap e_2) = P(e_1)P(e_2)$$

but another way of stating this is that

$$P(e_1 \mid e_2) = P(e_1) \qquad \text{and} \qquad P(e_2 \mid e_1) = P(e_2)$$

I.e., e_1 and e_2 are independent iff knowing that one has happened does not change the probability of the other.

Expected value: numeric probabilities

When our outcomes are numeric (as opposed to cards or colors or what not) we can ask additional questions about the outcome space. The *expected value* is the numeric value that we would expect to see over the entire space of outcomes. For example, for a fair d6, the expected value is 3.5, because if we rolled a d6 an

infinite number of times and averaged the values, we would get 3.5. We write the expected value as

$$E[\{1,2,3,4,5,6\}] = 3.5$$

or more generally

$$E[S] = \frac{1}{|S|}\Sigma_{i \in S} i$$

Transitive Dice (optional)

Consider a simple gambling game: two players both roll two identical, fair dice, highest roll wins. Under these settings, neither player would have an advantage over the other, because both dice have the same expected outcome. As a further variation, suppose we have three dice:

$$D_1 = \{1,2,3,4,5,6\}$$
$$D_2 = \{2,3,4,5,6,7\}$$
$$D_3 = \{3,4,5,6,7,8\}$$

The first player chooses a die, and then the second player chooses, and then they compete. For these dice, there is no reason for the first player to choose anything but D_3; this die will reliably win against the other two. Is there a way to construct a set of dice in which the *second* player will have the advantage, rather than the first?

First, consider the probability that one die will win against another. For example, for D_1 vs. D_2, what is the probability $P(D_1 < D_2)$?

Number Theory

Here we're interesting in the properties of numbers, particularly whole numbers (integers), particularly-particularly whole numbers greater than or equal to zero. We call these *natural numbers*. If we want to talk about all the natural numbers, we denote this collection $\mathbb{N}$.

Preliminaries: Inference rules

We will cover inference rules in section but we will give a brief introduction here as they will be useful for our purposes.

An inference rule is a shorthand for writing an if-then statement. For example, this rule states that "if it is raining, then I am carrying an umbrella":

$$\frac{\text{It is raining}}{\text{I'm carrying an umbrella}}$$

Note that it does *not* follow from this that if I am carrying an umbrella, then it is raining; there may be other reasons, not given, why I would be carrying an umbrella.

A rule with *nothing* above the line is called an *axiom* and it is true everywhere:

$$\frac{}{\text{It is hot}}$$

A rule may have multiple *premises* above the line, all of which must hold in order for the *conclusion* (below the line) to hold:

$$\frac{\text{It is raining} \qquad \text{I'm outside}}{\text{I'll have an umbrella}}$$

Given a collection of inference rules, we can ask what sorts of things can be *proven* from them. For example, consider the following rules:

$$\frac{}{\text{It is hot}} \qquad \frac{}{\text{It is cloudy}} \qquad \frac{}{\text{It is January}} \qquad \frac{}{\text{It is Saturday}}$$

$$\frac{\text{It is Saturday}}{\text{I am outside}} \qquad \frac{\text{It is January}}{\text{It is raining}} \qquad \frac{\text{It is January} \qquad \text{I am outside}}{\text{I am cold}}$$

$$\frac{\text{I am cold} \qquad \text{It is raining}}{\text{I am wearing a coat}}$$

If we wish to prove under these rules that *I am wearing a coat* we begin with that statement as our conclusion:

$$\frac{?}{\text{I am wearing a coat}}$$

We then expand the premises of the rule that matches the conclusion:

$$\frac{\dfrac{?}{\text{I am cold}} \qquad \dfrac{?}{\text{It is raining}}}{\text{I am wearing a coat}}$$

and continue our way upwards until every "branch" of this tree ends in an axiom:

$$\frac{\dfrac{\dfrac{}{\text{It is January}} \qquad \dfrac{\dfrac{}{\text{It is Saturday}}}{\text{I am outside}}}{\text{I am cold}} \qquad \dfrac{\dfrac{}{\text{It is January}}}{\text{It is raining}}}{\text{I am wearing a coat}}$$

This sort of tree-structure is called a *derivation*, and serves as a semi-formal proof that the conclusion follows from the given axioms and rules.

The inductive definition of $\mathbb{N}$

We are interested in finding the *simplest* definition for $\mathbb{N}$ that still allows us to investigate its properties in a useful way. We begin by asking what the *simplest* element of $\mathbb{N}$ is; zero is an obvious choice. Whatever else may be a natural number, 0 obviously is. We state this via an axiomatic inference rule:

$$\text{Nat-Zero}\ \frac{}{0 \in \mathbb{N}}$$

We now face the problem of constructing the other numbers. How can we "build" the number 1, starting with 0? One obvious method is to "add one to it". $1 + 0 = 1$. Similarly, if we want to know how to construct 2, we can add 1 to 0, and then add 1 to the result of that: $1 + (1 + 0) = 2$. We call the operation of "adding one to" something the *successor* and write it as $S(n)$. Combining this with the rule for 0 gives us a complete inductive definition of $\mathbb{N}$:

$$\text{Nat-Zero}\ \frac{}{0 \in \mathbb{N}} \qquad\qquad \text{Nat-Succ}\ \frac{n \in \mathbb{N}}{S(n) \in \mathbb{N}}$$

Using this definition we can build up any natural number we want by starting with 0 and applying the Nat-Succ rule as many times as needed. Likewise, if we are given some possibly-natural number, say, $S(S(S(0)))$ $(= 3)$ and we want to know if it is, in fact, a proper natural number, we can construct a derivation to convince ourselves:

$$\dfrac{\text{Nat-Succ}(n = S(S(0)))\ \dfrac{\text{Nat-Succ}(n = S(0))\ \dfrac{\text{Nat-Succ}(n = 0)\ \dfrac{\text{Nat-Zero}\ \dfrac{}{0 \in \mathbb{N}}}{S(0) \in \mathbb{N}}}{S(S(0)) \in \mathbb{N}}}{S(S(S(0))) \in \mathbb{N}}$$

What happens if we try to prove that, e.g., $S(\text{potato}) \in \mathbb{N}$?

$$\text{Nat-Succ}\ \frac{?\ \dfrac{?}{\text{potato} \in \mathbb{N}}}{S(\text{potato}) \in \mathbb{N}}$$

We get stuck. Note that our failure to construct a derivation does *not* constitute a proof that $S(\text{potato}) \notin \mathbb{N}$, it merely means "we don't know". We distinguish betwee *false* and *unknown*, because there are problems which are unsolved, for which we cannot say, and maybe never will be able to say, "this is true" or "this is false".

Operations on Natural Numbers

Given the inductive definition of $\mathbb{N}$, can we define the usual comparison and arithmetic operations on it? We can, using *recursion*. While induction tells us how to build something up, recursion tells us how to use that to take something apart. Consider our definition above: if we want to define an operation on $x \in \mathbb{N}$, according to the definition, the *only* situations we need to worry abour are

$x = 0$	("Base case", by rule Nat-Zero)
$x = S(n)$	("Inductive case", by rule Nat-Succ)

In the inductive case, we have one crucial piece of additional information: we know that $n \in \mathbb{N}$. So that means that whatever we are doing, we can continue to do it on n, until we reach the case where $n = 0$. Because 0 stands by itself and doesn't have anything hidden "inside" it, we can define whatever our operation is supposed to do directly.

Comparisons: An example may make this more concrete: we wish to give a recursive definition for the less-than relation: $a < b$ with $a, b \in \mathbb{N}$. The first question we have to ask is, what can we say about $<$ that is true in the context of 0? That is, what fact can we state, immediately, about $<$ and 0, without knowing anything else? Our first instinct is probably to say something like "0 is less than every other number". By "every other" number, we mean, natural numbers starting at 1 (that is, $S(0)$). We can express the idea of "every natural number, starting at 1" by writing $S(n)$. We can fill in n with anything we like, but no matter what we use, the result will always be non-zero. Combining these two insights, we get the base case for $<$:

$$\text{<-Zero}\ \frac{}{0 < S(n)}$$

If we wanted to be completely correct, we would write

$$\text{<-Zero}\ \frac{n \in \mathbb{N}}{0 < S(n)}$$

but we will leave this implicit for now.

Now, the fact that we have a 0 on the left-hand side should clue us in that we will need a $S(a)$ in that position in the inductive case (we have to cover both possible ways of constructing a $\mathbb{N}$):

$$\text{<-Succ}\ \frac{a < ?}{S(a) < S(b)}$$

(details to follow) Rewrite this to explain why $S(b)$ is required on the RHS.

Here, we need to find something that we can say about $S(a) < ?$, but we get, for free, the ability to apply whatever we are saying to $a < ?$. The purpose of this case is to "break down" the left hand side by stripping off one successor. If we do this enough times, we will eventually reach the base case and then we will be done.

In order to figure out how to complete the definition, we will re-write it in the more familiar mathematical form, with variables on the right-hand side:

$$\text{if } a < b \text{ then } 1 + a < b'$$

What relation needs to hold between b and b' in order for this statement to be true? What if we let $b' = b + 1$? Then we have

$$\text{if } a < b \text{ then } (1 + a) < (1 + b)$$

and this statement is true. Translating this back into successors gives us the complete definition:

$$\text{<-Zero}\ \frac{}{0 < S(n)} \qquad \text{<-Succ}\ \frac{a < b}{S(a) < S(b)}$$

Note that we are in fact breaking down *both* a and b; this will sometimes be necessary, but it is *always* required that we make the operand smaller which has a 0 in that position in the base case. Remember that our goal is to eventually reach the base case and thus be able to stop!

Note that, as before, just because we cannot construct a derivation (e.g., that $S(S(0)) < 0$) this does not imply that we know $S(S(0)) \geq 0$, the negation of $<$. If we want $\geq$, we will have to construct its own set of rules, which we will do next.

Our base case is fairly simple:

$$\geq\text{-Zero}\ \frac{}{n \geq 0}$$

(I.e., "anything is ≥ zero".)

In our inductive case, we will need a $S(b)$ where we have a 0, so a template of our rule will be

$$\geq\text{-Succ}\ \frac{a \geq b}{a' \geq S(b)}$$

As above, if we let $a' = 1 + a$ we have a true statement and a complete definition:

$$\geq\text{-Zero}\ \frac{}{n \geq 0} \qquad \geq\text{-Succ}\ \frac{a \geq b}{S(a) \geq S(b)}$$

Construction of the ≤, >, etc. comparisons is left as an exercise for the reader; they mostly follow the patterns of the definitions given above.

Arithmetic operators: We now wish to derive recursive definitions for the usual arithmetic operators: +, ×, −, and /. We will also be interested in constructing definitions for two operations related to division: *divisibility*, written $a \mid b$ for "b is divisible by a" and *remainder*, written $a \operatorname{rem} b$ for the remainder of a divided by b.

For addition, we have as our base case the fact that $0 + b = 0$:

$$\text{Add-Zero}\ \frac{}{0 + b = b}$$

For our inductive case, we will have something along the lines of

$$\text{Add-Succ}\ \frac{a + b = c}{S(a) + b' = c'}$$

where it is up to us to define b' and c' to make the statement true. With a little thought, we see that letting $c' = S(c)$ and $b' = b$ will work:

$$\text{Add-Zero}\ \frac{}{0 + b = b} \qquad \text{Add-Succ}\ \frac{a + b = c}{S(a) + b' = S(c)}$$

For subtraction, we have two possibilities:

- We can define normal subtraction, $a - b$, but state that it is only defined when $a \geq b$.
- We can define *truncated* subtraction, usually called "monus", which is defined as

$$a \dot{-} b = \begin{cases} 0 & \text{if } a < b \\ a - b & \text{otherwise} \end{cases}$$

$a \dot{-} b$ is defined for all a, b and thus will sometimes prove more useful.

Traditional subtraction is defined from the base case $a - 0 = a$:

$$\text{Sub-Zero}\ \frac{}{a - 0 = a} \qquad \text{Sub-Succ}\ \frac{a - b = c}{S(a) - S(b) = c}$$

Defining monus requires us to define an auxillary operation, the *predecessor*, which we write as $P(n)$:

$$\text{Pred-Zero}\ \frac{}{P(0) = 0} \qquad \text{Pred-Succ}\ \frac{}{P(S(n)) = n}$$

Intuitively, the predecessor just subtracts 1, except at 0, where it just stops.

With the predecessor, we can now proceed to define $a \dot{-} b$:

$$\text{Mon-Zero}\ \frac{}{a \dot{-} 0 = a} \qquad \text{Mon-Succ}\ \frac{a \dot{-} b = c \qquad c' = P(c)}{S(a) \dot{-} b = c'}$$

Aside: inductive proofs As an informal introduction to how we can *prove* things using inductive definitions, consider the problem of proving that, for any a, $a - a = 0$. We don't know anything about a except that $a \in \mathbb{N}$. But this tells us that we have two possibilities we need to examine:

It is also the case that $a \dot{-} a = 0$ for any a, and indeed, the proof is essentially identical.

$$a = 0\ \frac{?}{0 - 0 = 0} \qquad a = S(a')\ \frac{?}{S(a') - S(a') = 0}$$

(We call these two cases the "base case" and the "inductive case".)

The first case, $a = 0$, follows directly from the Sub-Zero rule, so we are done there. In the second case, we can use the Sub-Succ rule to see where that gets us:

$$\text{Base case } (a = 0)\ \frac{}{0 - 0 = 0} \qquad \text{Inductive case } (a = S(a'))\ \frac{\dfrac{?}{a' - a' = 0}}{S(a') - S(a') = 0}$$

This doesn't appear to have gotten us anywhere, as we have reduced the task of proving $a - a = 0$ to proving the same thing, $a' - a' = 0$! It is as if we said, "to prove P, you must prove P". However, there is a key difference: a' is *smaller* than a. Notice how, in all of our recursive definitions, we *assume* that we can (e.g.) compute $a + b$ correctly, and then from that derive $S(a) + b$. Put another way, so long as we can "shrink" the problem a bit in some way, we will eventually reach the zero case which can give us the answer right away. We will do the same thing here: we will *assume* that $a' - a' = 0$ so long as a' is smaller than what we started with (we call this assumption the "inductive hypothesis" and abbreviate it IH). And, since $a = S(a')$, a' is smaller than a and we can apply the IH, completing our proof:

$$\text{Base case } (a = 0)\ \frac{}{0 - 0 = 0} \qquad \text{Inductive Case } (a = S(a'))\ \frac{\text{IH}\ \dfrac{}{a' - a' = 0}}{S(a') - S(a') = 0}$$

If this doesn't make sense, try applying it to a *particular* $a \in \mathbb{N}$, say, $a = S(S(0))$:

$$\text{Inductive case: } a' = S(0)\ \frac{\text{Inductive case: } a' = 0\ \dfrac{\text{Base case}\ \dfrac{}{0 - 0 = 0}}{S(0) - S(0) = 0}}{S(S(0)) - S(S(0)) = 0}$$

By reducing the proof that $(a + 1) - (a + 1) = 0$ to $a - a = 0$ we ensure that we will eventually reach $0 - 0 = 0$, which is true axiomatically.

If this form of "proof" looks like the other kinds of recursive definitions we've been building, that is not a coincidence. Just as the rules Plus-Zero and Plus-Succ combine to build a recursive *algorithm* for taking two natural numbers and computing their sum, so this proof builds a recursive algorithm for taking a natural number n and computing a concrete proof that $n - n = 0$. An inductive proof is in fact just another recursive function, just one that outputs proofs instead of values!

We can define multiplication as just repeated addition:

$$\text{Mult-Zero}\ \frac{}{0 \times b = 0} \qquad \text{Mult-Succ}\ \frac{a \times b = c}{S(a) \times b = b + c}$$

(I.e., $(1 + a)b = b + ab$.)

Division

Division will turn out to be one of the more interesting areas of number theory, but for now we will build up the recursive definitions of three operations:

- *Truncated* (integer) division: $\lfloor a/b \rfloor$; aka, "division rounded down".
- *Remainder*: $a \operatorname{rem} b$
- *Divisibility*: $b \mid a$ iff $a \operatorname{rem} b == 0$

Truncated division: We define division by *repeated subtraction*. That is, given a/b, our goal is to find out how many "copies" of b will "fit" into a. We do this by subtracting out b's until we can go no further (i.e., until the result is $< b$). Our definition looks like this:

$$\text{Div-Base}\ \frac{a < b}{a/b = 0} \qquad \text{Div-Rec}\ \frac{a \geq b \qquad a - b = a' \qquad a'/b = c}{a/b = S(c)}$$

We reduce the case of computing a/b to computing $1 + ((a - b)/b)$.

Exercise: what operation is defined if we use the monus $\dot{-}$ instead of subtraction?

It is interesting to notice that this method does not break down into the usual Zero and Successor cases. Instead, our base case is defined by the relative magnitudes of the numerator and denominator, and our recursive case reduces the size of a not by 1 but by b. In fact, as we shall see later, this is the key difference between *natural number* induction and *strong* induction. Natural number induction shrinks by 1, strong induction can shrink by *any* value (> 0 of course). This also illustrates why division by 0 is undefined: if $b = 0$ then we are not subtracting *anything* and a never gets smaller.

Remainder: Division, as we've defined above, throws away some information (hence the name, "truncated"). In particular, when we reach the base case, we

may have $a \neq 0$ in which case b does not divide a evenly, but rather with some *remainder*. The remainder is in fact just this last a, whatever is left over after we have subtracted out all the b's we can. If we have *both* the truncated quotient and the remainder, then we can reconstruct the original a. The definition of the remainder is very similar to that of division: we work by repeated subtraction, except that we return the final a:

$$\text{Rem-Base}\ \frac{a < b}{a \text{ rem } b = a} \qquad \text{Rem-Rec}\ \frac{a \geq b \qquad a - b = a' \qquad a' \text{ rem } b = c}{a \text{ rem } b = c}$$

In fact, the two operations are so similar that they are sometimes combined into a single operation $\text{quotRem}(a, b) = (q, r)$:

$$\text{QR-Base}\ \frac{a < b}{\text{quotRem}(a, b) = (0, a)} \qquad \text{QR-Rec}\ \frac{a \geq b \qquad a - b = a' \qquad \text{quotRem}(a', b) = (q, r)}{\text{quotRem}(a, b) = (S(q), r)}$$

(This is built-in to Haskell as the `quotRem` function.)

Note that, given these definitions, for any $a \in \mathbb{N}, b \in \mathbb{N}, b \neq 0$ if we have $\text{quotRem}(a, b) = (q, r)$ then it is the case that $r + qb = a$. Remember that q is the number of copies of b we were able to pull out of a, and r is whatever was left over. So if we *start* from r, and then add q copies of b back in, we'll get back to where we started.

Divisibility: Finally, we define divisibility $b \mid a$ by repeated subtraction as well, following the above definitions:

$$\mid\text{-Base}\ \frac{}{b \mid 0} \qquad \mid\text{-Rec}\ \frac{a \geq b \qquad b \mid (a - b)}{b \mid a}$$

Division with negative numbers: If we want to extend division to $\mathbb{Z}$, the set of integers (positive and negative whole numbers) we run into some interesting questions. We wish to define integer division $a/b = q$ with remainder r such that

$$a = bq + r \qquad \text{with} \qquad |r| < |b|$$

$|r| < |b|$ expresses our condition that the remainder be "smaller", in magnitude, than the divisor. This still leaves us with several possible choices about the *sign* of q and r.

The definition that most of us learned in elementary school is that if a and b have the *same sign* then q is positive, whereas if their signs are different, then q is negative. If we examine this in the context of an example, we can determine what this forces the sign of the remainder to be:

$$\begin{aligned} 4 \text{ quot } -3 \qquad & q = -1, 4 = (-3)(-1) + 1, r = +1 \\ -4 \text{ quot } 3 \qquad & q = -1, -4 = (3)(-1) - 1, r = -1 \\ -4 \text{ quot } -3 \qquad & q = +1, -4 = (-3)(1) - 1, r = -1 \end{aligned}$$

In this scheme, the sign of the remainder is the same as the sign of a, the dividend.

An alternate scheme is to require the sign of the remainder to match that of the *divisor*:

$$\begin{aligned} 4 \text{ div } -3 \quad & 4 = -3q + r, q = -2, r = -2 \\ -4 \text{ div } 3 \quad & -4 = 3q + r, q = -2, r = +2 \\ -4 \text{ div } -3 \quad & -4 = -3q + r, q = +1, r = -1 \end{aligned}$$

Finally, we have the "Euclidean modulo" scheme, in which the remainder is *always positive*:

$$\begin{aligned} 4/-3 \quad & 4 = -3q + r, q = -1, r = +1 \\ -4/3 \quad & -4 = 3q + r, q = -2, r = +2 \\ -4/-3 \quad & -4 = -3q + r, q = +2, r = +2 \end{aligned}$$

Haskell note:

In Haskell, the first definition is given by the built-in functions `quot` and `rem`. The second is given by `div` and `mod`. If you need *both* the quotient and remainder, then there are functions `quotRem` and `divMod` which return them as a pair.

The Euclidean modulo is unfortunately not built-in to any programming language, despite its many desirable qualities. ———————————

Greatest Common Denominators and the Euclidean Algorithm

As a prerequisite to several topics we'll look into later, we are interested in finding the *greatest common denominator* (GCD) of two (or more) integers a and b. There are several definitions we can use for the GCD:

- If we consider our inputs as *products of prime factors* then we have something like this:

$$\begin{aligned} a &= 2^{m_1} \times 3^{m_2} \times 5^{m_3} \times 7^{m_4} \times \cdots \\ b &= 2^{n_1} \times 3^{n_2} \times 5^{n_3} \times 7^{n_4} \times \cdots \end{aligned}$$

  Under this definition, the GCD is the product of the *minimum powers* of all the prime factors; i.e., all the factors that both values have in common:

$$\gcd(a, b) = 2^{\min(m_1, n_1)} \times 3^{\min(m_2, n_2)} \times 5^{\min(m_3, n_3)} \times 7^{\min(m_3, n_3)} \ldots$$

  The exponents n_i are called the *multiplicites* of the various factors. Instead of writing $12 = 2 \times 2 \times 3$ we write $12 = 2^2 \times 3^1 \times 5^0 \cdots$.

- An alternate definition is based on the idea of a common denominator: c is a common denominator if

$$c \mid a \qquad \text{and} \qquad c \mid b$$

  For c to be the "greatest" common denominator, it must not just be larger than any other common denominator, it must also *include* them. I.e., if

d is a common denominator of a and b then if c is the greatest common denominator

$$d \mid c$$

Both definitions will prove useful to us: the first is more intuitive, but the second will be more algebraically (and algorithmically) useful.

Given these definitions, we want to derive the actual greatest common denominator $c = \gcd(a, b)$. One method would be to factor both a and b, and then form the product of all the common factors. However, factoring integers is computationally difficult; if we were to use this method on large numbers it would quickly become intractable. Instead, we will derive a recursive method for reducing $\gcd(a, b)$ to the GCD of some "smaller" inputs.

Looking at the second definition, we have

$$c \mid a \qquad \text{and} \qquad c \mid b$$

This implies that

$$a = cx_1 \qquad \text{and} \qquad b = cx_2$$

If we subtract the second equation from the first we find that

$$a - b = c(x_1 - x_2)$$

Letting $x' = x_1 - x_2$ we can put this into the form corresponding to divisibility, and thus derive:

$$c \mid a - b$$

Thus, if we assume that a is the larger of the two inputs, we can reduce the problem of finding $\gcd(a, b)$ to that of finding $\gcd(b, a - b)$. Our base case will be $\gcd(a, 0) = a$. This leads us to the recursive definition:

We use $\gcd(a, 0) = a$ as our base case because $a \mid 0$ for any a.

$$\gcd(a, 0) = a$$
$$\gcd(a, b) = \gcd(b, a - b)$$

where we take it as implicit that a is always the larger of the two arguments. (It's easy enough to swap the arguments if this is not the case.)

Working through an example, suppose we want to find gcd(250, 111):

$$\begin{aligned}
\gcd(250, 111) &= \\
\gcd(139, 111) &= && (250 - 111 = 139) \\
\gcd(111, 28) &= && (139 - 111 = 28) \\
\gcd(83, 28) &= && (111 - 28 = 83) \\
\gcd(55, 28) &= && (83 - 28 = 55) \\
\gcd(28, 27) &= && (55 - 28 = 27) \\
\gcd(27, 1) &= && (28 - 27 = 1) \\
\gcd(26, 1) &= \\
&\vdots \\
\gcd(2, 1) &= \\
\gcd(1, 1) &= \\
\gcd(1, 0) &= 1
\end{aligned}$$

We find that the greatest common denominator of 250 and 111 is 1 (i.e., 250 and 111 do not share *any* factors). If two numbers do not share any factors we call them "relatively prime".

Looking at the sequence of GCDs performed above, we seem to be doing a lot of *repeated subtraction*, which should clue us in to the possibility of simplifying our procedure by using division. In fact, if we look at the inputs above we find that

$$\begin{aligned}
250 \text{ rem } 111 &= 28 \\
111 \text{ rem } 28 &= 27 \\
28 \text{ rem } 27 &= 1 \\
27 \text{ rem } 1 &= 0
\end{aligned}$$

We can "shortcut" through the subtraction by replacing it with a remainder. This also saves us the trouble of having to keep track of which argument is larger, because $a \text{ rem } b < b$. Thus, we have the new-and-improved recursive definition

$$\begin{aligned}
\gcd(a, 0) &= a \\
\gcd(a, b) &= \gcd(b, a \text{ rem } b)
\end{aligned}$$

Properties of the GCD

We've already seen one property of the GCD in our base case:

$$\gcd(a, 0) = a$$

Given that the GCD is defined as being the factors a and b have in common, we would expect that

$$\gcd(a, b) = \gcd(b, a)$$

although we will leave the problem of proving this to later. (This implies that, in all of the following properties, we can swap the arguments to get a related property.)

Note that usually, we will say that

$$\gcd(0,0) \text{ is undefined.}$$

Because $1 \mid 1$, we find that

$$\gcd(a,1) = 1$$

If p is some prime, then for any $a < p$ we have

$$\gcd(a,p) = 1$$

This property will become useful later.

The GCD is *associative* in that

$$\gcd(a,\gcd(b,c)) = \gcd(\gcd(a,b),c)$$

This implies that we can ask for the GCD of a *set* of values:

$$\gcd(\{n_1,n_2,\ldots,n_i\}) = \gcd(n_1,\gcd(\{n_2,\ldots,n_i\}))$$

The least common multiple: A closely-related concept is the *least common multiple* of a and b, the smallest value that both a and b divide into evenly. Its definitions are very similar to those for the GCD:

- If

$$a = 2^{m_1} \times 3^{m_2} \times 5^{m_3} \times 7^{m_4} \times \cdots$$
$$b = 2^{n_1} \times 3^{n_2} \times 5^{n_3} \times 7^{n_4} \times \cdots$$

  then

$$\operatorname{lcm}(a,b) = 2^{\max(m_1,n_1)} \times 3^{\max(m_2,n_2)} \times 5^{\max(m_3,n_3)} \times 7^{\max(m_3,n_3)} \ldots$$

- If $c = \operatorname{lcm}(a,b)$ then

$$a \mid c \qquad \text{and} \qquad b \mid c$$

  and for any other common multiple d

$$c \mid d$$

While the GCD represents the idea of all the factors that the two values have *in both*, the LCM represents all the factors that are *in either*. In set-theoretic terms, the GCD is the intersection while the LCM is the union. In fact, this view lets us define the LCM in terms of the GCD, by using a variant of the inclusion-exclusion principle. The inclusion-exclusion principle tells

us that if we want everything that is in either set, without duplicates, we take everything either set, and then remove one copy of everything that is in both. To get "everything in either" we simply multiple, since this will add together the corresponding exponents. To remove "everything in both" we divide by the GCD, since the GCD is what is common, and division amounts to subtracting from the exponents. This gives us the definition

$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

This also implies that

$$\text{lcm}(a, b) \times \gcd(a, b) = ab$$

Together with the LCM, we have a kind of "distributive property":

$$\text{lcm}(a, \gcd(b, c)) = \gcd(\text{lcm}(a, b), \text{lcm}(a, b))$$
$$\gcd(a, \text{lcm}(b, c)) = \text{lcm}(\gcd(a, b), \gcd(a, c))$$

Modular arithmetic

Remainders have an interesting property in that we can move operations before or after them without changing the result. For example:

$$(12 + 37) \text{ rem } 5 = 49 \text{ rem } 5 = 4$$

But we could just as easily do

$$(12 + 37) \text{ rem } 5 = ((12 \text{ rem } 5) + (37 \text{ rem } 5)) \text{ rem } 5 = (2 + 2) \text{ rem } 5 = 4$$

We have the choice of doing the addition "outside", in the world of actual integers, or "inside", in the world of all numbers remainder 5; the result will be the same in either case. This will become more useful when our operations are on very large values: it will often be (much!) more computationally efficient to keep them reduced remainder some n, rather than wait until the end of the procedure to do the reduction.

We refer to doing things "remainder n" as "modulo n". If $a \text{ rem } n = b$ then we write

$$a \equiv b \mod n$$

and read this as "a is *congruent* to b mod n". Some quick properties follow from the definition of the remainder:

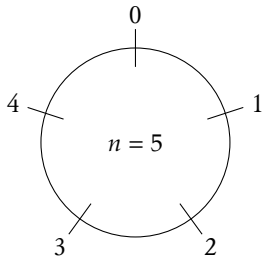
$$a \equiv 1 \mod 1$$
$$0 \equiv 0 \mod n$$
$$1 \equiv 1 \mod n$$
$$kn \equiv 0 \mod n \qquad \text{(for any integer } k\text{)}$$

As mentioned above, we can move operations inside the modulo freely:

$$\text{if } a \equiv a' \mod n \text{ and } b \equiv b' \mod n \text{ then } a + b \equiv a' + b' \mod n$$
$$\text{if } a \equiv a' \mod n \text{ and } b \equiv b' \mod n \text{ then } ab \equiv a'b' \mod n$$

We will often take advantage of this to reduce some portion of an equation. For example, if we have $a + bn \equiv c \mod n$ we can eliminate the bn term because $bn \equiv 0 \mod n$, leaving us with $a \equiv c \mod n$.

Another, perhaps more intuitive way to think about modular arithmetic is "arithmetic around the face of a clock":



In order to compute (for example) $6 + 8 \mod 5$ we first go around the "clock", 6 steps clockwise (clockwise, because 6 is positive). From that point (which will be at "1") we continue another 8 steps, again clockwise, leaving us at "4". Because multiplication is just repeated addition, the same procedure can be used: to find $3 * 8 \mod 5$ we simply go 8 steps forward, 3 times, leaving us at "4".

Modular Multiplicative Inverses: A *multiplicative inverse* of a value a is a value a^{-1} such that

$$aa^{-1} = 1$$

In the world of fractions, every value except 0 has a multiplicative inverse, because we can always do

$$a\frac{1}{a} = 1$$

But for integers this is not the case. For example, there is no integer x such that

$$2x = 1$$

However, when we compute modulo n, we *sometimes* have modular multiplicative inverses:

$$1(1) \equiv 1 \mod 5$$
$$2(3) \equiv 1 \mod 5$$
$$3(2) \equiv 1 \mod 5$$
$$4(4) \equiv 1 \mod 5$$

Here, as we expect, every value except 0 has a multiplicative inverse. But this is not always the case:

$$\begin{aligned} 1(1) &\equiv 1 \mod 6 \\ 2(?) &\equiv 1 \mod 6 \\ 3(?) &\equiv 1 \mod 6 \\ 4(?) &\equiv 1 \mod 6 \\ 5(5) &\equiv 1 \mod 6 \end{aligned}$$

Try as we might, there is no value x such that $2x \equiv 1 \mod 6$. Observe:

$$\begin{aligned} 2(0) &\equiv 0 \mod 6 \\ 2(1) &\equiv 2 \mod 6 \\ 2(2) &\equiv 4 \mod 6 \\ 2(3) &\equiv 0 \mod 6 \\ 2(4) &\equiv 2 \mod 6 \\ &\vdots \end{aligned}$$

Our remainders repeat in an endless cycle of $0, 2, 4, \ldots$. This happens because 2 and 6 *share a factor*, namely 2. Because of this, we can never escape the cycle of multiples of that factor. In fact, for any modulo n, a will have a multiplicative inverse $\mod n$ if and only if a and n are *relatively prime* (i.e., iff $\gcd(a, n) = 1$). If we want to ensure that we have all modular inverses, we will have to choose our n's such that *every* a is relatively prime to them. The only way to do this is to only choose *actual* prime numbers for our n's. That way, for any $a \neq 0$, we will have $\gcd(a, n) = 1$ ensuring that a has a modular inverse.

Finding modular inverses: To find the modular inverse of some a (with $\gcd(a, n) = 1$) we make use of *Bezout's identity* which states

$$\text{There exist } x, y \text{ such that } \gcd(a, b) = ax + by$$

To see why this is useful, remember that we have

$$1 = ax + ny$$

because a and n are relatively prime. If we take this modulo n we have

$$ax + ny \equiv 1 \mod n$$

but $ny \equiv 0 \mod n$ so really we have

$$ax \equiv 1 \mod n$$

which makes x the modular inverse of a. (Note that x and y need not be unique, but all $x \mod n$ will reduce to the same value.)

To find x and y to satisfy the identity, we will make a relatively simple change to the Euclidean algorithm we used to compute the GCD, giving us the *Extended Euclidean Algorithm*. We want

$$\text{egcd}(a, b) = (c, x, y) \quad \text{with} \quad c = ax + by$$

Our base case will be similar to that for the ordinary GCD:

$$\text{egcd}(a, 0) = (a, x, y) \quad \text{with} \quad a = ax + 0y$$

Obviously $x = 1$ and the value of y does not matter; we choose $y = 0$ for simplicity, giving us

$$\text{egcd}(a, 0) = (a, 1, 0)$$

To construct our recursive case, we *assume* that we have already executed the recursive call and gotten its results:

$$\text{egcd}(b, a \mod b) = (c, x, y) \quad \text{with} \quad c = bx + (a \mod b)y$$

From this, we want to reconstruct the solution to

$$\text{egcd}(a, b) = (c', x', y') \quad \text{with} \quad c' = ax' + by'$$

First, we notice that, as in the original GCD, c does not change when we move from the recursive call to the outer call. Thus, we have $c' = c$. Substituting this in and setting the two equations equal to each other gives us

$$bx + (a \mod b)y = ax' + by'$$

Here, we will assume that we have computed

$$q = a \operatorname{div} b$$
$$r = a \mod b$$
$$\text{with } a = qb + r$$

giving us

$$bx + ry = ax' + by'$$

Here, we have one equation with two unknowns (x' and y'). In order to make some progress, we will let $x' = y$ becausLOOK OVER THERE

This gives us

$$bx + ry = ay + by'$$

which we can solve for y' giving

$$y' = \frac{bx}{b} + \frac{y(r - a)}{b}$$

If we rearrange our division identity, we find that $r - a = -qb$ and thus

$$y' = x - yq$$

Thus, our full recursive definition is

$$\begin{aligned} \text{egcd}(a, 0) &= (a, 1, 0) && \text{(Base case)} \\ \text{egcd}(a, b) &= \text{let } (c, x, y) = \text{egcd}(b, a \mod b) \\ &\quad \text{in } (c, y, x - y(a \operatorname{div} b)) && \text{(Recursive case)} \end{aligned}$$

And then, if $\text{egcd}(a, n) = (1, i, -)$ we have $ai \equiv 1 \mod n$.

The Chinese Remainder Theorem

Suppose we have a value s, and we compute s modulo several different values:

$$\begin{aligned} s &\mod m_1 = a_1 \\ s &\mod m_2 = a_2 \\ &\vdots \\ s &\mod m_n = a_n \end{aligned}$$

If we are only given all the m_i and a_i, can we reconstruct the original s? In fact, under certain circumstances, the answer is Yes. We will examine how this can be done, and demonstrate an application to the problem of *secret sharing*.

To start, we will look at the situation in which we have only two modulos: m_1 and m_2. Furthermore, we will assume both these are prime. Since they are both prime, Bezout's identity implies that

$$\text{gcd}(m_1, m_2) = 1 = xm_1 + ym_2$$

for some x and y. In fact, $x = m_1^{-1} \mod m_2$ and likewise $y = m_2^{-1} \mod m_1$, from the definition of the modular inverse we found above.

If we multiply both sides by s we have

$$s = sxm_1 + sym_2$$

If we take this modulo m_1 and m_2 we have

$$\begin{aligned} s &\equiv sx\underbrace{m_1}_{0} + s\underbrace{ym_2}_{1} \mod m_1 \\ s &\equiv s\underbrace{xm_1}_{1} + sy\underbrace{m_2}_{0} \mod m_2 \end{aligned}$$

(Note that $m_1 \equiv 0 \mod m_1$ and likewise for m_2.)

But note that $s \equiv a_1 \mod m_1$, and likewise $s \equiv a_2 \mod m_2$ so we can substitute a_1 for s when taken modulo m_1, and likewise a_2 for s when modulo m_2:

$$\begin{aligned} s &\equiv a_1x\underbrace{m_1}_{0} + a_1ym_2 \mod m_1 \\ s &\equiv a_2xm_1 + a_2y\underbrace{m_2}_{0} \mod m_2 \end{aligned}$$

We can now *add* these two equations together, dropping the zero terms and the modulos, giving

$$s = a_2 x m_1 + a_1 y m_2$$

I.e., to reconstruct s, we find $x = m_1^{-1} \mod m_2$ and $y = m_2^{-1} \mod m_1$. We then multiply each a_i by the *other* m_j, $j \neq i$ and its corresponding inverse, and sum the results. In fact, the sum may not be the final solution, but merely congruent to it, so we actually have to do

$$s = a_2 x m_1 + a_1 y m_2 \mod (m_1 m_2)$$

Generalizing to n modulos: Generalizing the CRT to an arbitrary number of modulos is not difficult. We glossed over the reason why each term in the final sum has a_1 but m_2 in it by saying that it was the "other" modulo. In fact, the general form of for $n = 2$ is

$$\begin{gathered} M = m_1 m_2 \\ d_1 = \frac{M}{m_1} \qquad d_2 = \frac{M}{m_2} \\ d_1^{-1} = \text{minv}_{m_1}(d_1) \qquad d_2^{-1} = \text{minv}_{m_2}(d_2) \\ s = \left[a_1 d_1 d_1^{-1} + a_2 d_2 d_2^{-1} \right] \mod M \end{gathered}$$

(Where $\text{minv}_m(x)$ gives the modular inverse of x, modulo m.) I.e., we form the product of *all* the modulos, and then dividing by each modulo in turn gives us the d_i that forms the core of each term in the final sum.

Expanded out to n terms, we have the solution to the series of congruences

$$\begin{aligned} s \mod m_1 &= a_1 \\ s \mod m_2 &= a_2 \\ &\vdots \\ s \mod m_n &= a_n \end{aligned}$$

given by

$$\begin{aligned} M &= m_1 m_2 \cdots m_n && \text{(Product of all modulos)} \\ d_i &= \frac{M}{m_i} \quad \text{for all } i && \text{(Quotient of each modulo)} \\ d_i^{-1} &= \text{minv}_{m_i}(d_i) \quad \text{for all } i && \text{(Inverse of each quotient)} \\ s &= \left[\sum_{0 \leq i \leq n} a_i d_i d_i^{-1} \right] \mod M \end{aligned}$$

Note that the m_i do not actually all need to be prime; this was done above purely for convenience. By Bezout's identity, it is sufficient for them to all be *pairwise coprime* (i.e., for any pair m_i, m_j with $i \neq j$, m_i and m_j are relatively prime). Likewise, a further restriction is implied by the final $\mod M$ step: the original s must be $< M$. If it is not, the solution we get will be $s' = s \mod M$ and we will not be able to recover the original s.

An application to secret sharing: A *secret sharing* problem is one in which we have a secret s which we wish to share with some n individuals, in such a way that s can only be recovered if at least $k \leq n$ of them cooperate. In the simplest case, $k = n$ and all the individuals must cooperate to recover the secret.

It's fairly easy to see how the CRT can be used for the simple case of secret sharing: s is the secret, and a pair (a_i, m_i) is generated and given to each individual. If all individual come together, they can use the CRT to recover s. However, there is an important condition on the m_i used: not only must the product $M = m_1 m_2 \ldots$ be greater than s, it must be the case that the product of *any subset* of the m_i is *less than* s. That is for any $0 \leq j \leq n$

$$\left[\frac{1}{m_j} \prod_{0 \leq i \leq n} m_i\right] < s < M$$

Remember that if $s \geq M$ then we cannot recover the original s, we can only recover $s \mod M$. If we want to hide the secret when $n - 1$ or fewer individuals come together, we must ensure that the product of any $n - 1$ or fewer modulos is $< s$.

This generalizes to $k \leq n$: we must form our original set of modulos such that at least k of them are required. If we say that $[M]^k$ is all the subsets of the modulos of size k then

$$\text{for all } K \in [M]^k\text{: } \prod K < s < M$$

We could further generalize this to isolating some individual(s) m_j whose presence was *required*, by making it so that no product of size k without m_j was big enough.

Although the above description establishes the conditions for secret sharing, it does not tell us how to *find* a set of n modulos, such that they are all pairwise coprime, and the product of any k or more is greater than s, but the product of any $k - 1$ or fewer is not. A brute-force method will work, but is slow. General schemes usually exploit specialized sequences of modulos which are ordered in such a way that subsequences have the required threshold conditions.

An interesting further application of the CRT to secret sharing is the fact that modular arithmetic allows us to operate on the secret *without knowing it*. For example, suppose we have a secret value divided among three individuals, who have not come together (i.e., the secret is not being recovered). We want to change the secret, to replace it with $s' = s + 1$ for example. We simply instruct each individual to modify their own portion as follows:

$$a_i' = a_i + 1 \mod m_i$$

(The modulos of course stay the same.) Just as we can move operations into and out of a modular ring freely, we can move an operation *into all* the modular rings of our secret-sharers. This can be used to apply almost *any* mathematical operation(s) to the secret without revealing it, so long as the result still falls within the threshold range ($\prod [M]^k < s' < M$).

Division is problematic, but doable under certain circumstances.

This technique also has applications to parallel computing: if s is not a known value, but rather an expression with a known magnitude, we can choose our m_i so that their product is big enough to contain s. We then distribute the computation defining s along with each m_i to n different *computers*. Each computer computes $a_i = s \mod m_i$ independently, and in parallel, and the final step is to reconstitute the a_i into the actual value of s.

Conversion between number bases

Although we usually don't think about it, the written form of a number as, for example "10,365" is *not* the value of the number itself, but rather a representation of it. Here we are concerned with converting between representations and values.

Given a number represented in some *base* b, we assume that every digit of the number is in the range $0 \ldots b - 1$. (Thus, in base 10, we have digits 0 through 9.) If this is the case, and the number is written with digits

$$d_n \ldots d_2 d_1 d_0$$

then the value of the number is given by

$$v = b^0 d_0 + b^1 d_1 + b^2 d_2 + \cdots + b^n d_n$$

For example, the base-7 number 1036_7 has value

$$v = 6(7^0) + 3(7^1) + 0(7^2) + 1(7^3) = 370$$

Since our calculators and computers normally work in base-10, this serves as a general procedure for converting a value to base-10, albeit implicitly.

If we want to convert a value to some other base, we rely on two observations:

- $v \mod b$ gives us the *lowest* digit of the representation. This corresponds to the "ones" digit in any base. Note that every higher terms in the sum above is multiplied by some non-zero power of b. Thus, when taken $\mod b$, all terms other than the lowest one are congruent to 0.
- $v \operatorname{div} b$ has the effect of "shifting" all the digits down. E.g., $1023 \operatorname{div} 10 = 102$. Thus, we can recurse on $v \operatorname{div} b$ to get the new lowest digit, which will be the *second* lowest digit in the final representation, and so forth.

We will terminate our procedure when $v \div b = 0$ as at that point all higher digits will be 0 (because $0 \operatorname{div} b = 0$ and $0 \mod 3 = 0$).

Thus, our procedure is

- Base case: $v = 0$ then the representation is just "0".
- Recursive case:

$$d = v \mod b$$
$$v' = v \operatorname{div} b$$

The low digit is given by d, and then we find the higher digits recursively on v'.

Boolean Algebra

In Boolean algebra we are primarily concerned with operations over a domain of only two values, *true* which we denote **1** and *false*, **0**. Over these values, we have the following operations:

We will see later that not all Boolean algebras have *just* true and false. It is possible to construct an (uninteresting) Boolean algebra with only *one* value, or with more than two.

Notation	Description
$\overline{a}$	NOT; if $a = 1$ then $\overline{a} = 0$ and vice versa
ab	AND; if a *and* b are 1 then the result is 1, otherwise it is 0
$a + b$	OR; if either of a *or* b is 1 then the result is 1, otherwise it is 0
$a \oplus b$	Exclusive-OR (XOR); if *one* of a or b is 1 then the result is 1, otherwise it is 0

(Other operations can be defined by combining these; we will examine several later.)

We can define these operations by giving *truth tables* for them, specifying, for every possible combination of inputs, what the result will be:

a	$\overline{a}$
0	1
1	0

a	b	ab
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a + b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

These tables *define* their respective operations, but we can construct a truth table for any boolean expression we like, by simply breaking it down and applying the above tables. For example, suppose we want to construct the table for $\overline{a\overline{b}}$. If we break down the expression into its components, we find that we need to know a, b, $\overline{b}$, $a\overline{b}$ and finally $\overline{a\overline{b}}$:

a	b	$\overline{b}$	$a\overline{b}$	$\overline{a\overline{b}}$
0	0	1	0	1
0	1	0	0	1
1	0	1	1	0
1	1	0	0	1

It is also possible to construct a truth table for a Boolean function of more than two variables. Because each variable can take on two values, a function of n variables will have 2^n rows in its definition.

From looking at the above tables, we can derive several important identities

about them:

$1 \cdot a = a \cdot 1 = a$	(Identity)
$0 + a = a + 0 = a$	(Identity)
$a + b = b + a$	(Commutativity)
$a \cdot b = b \cdot a$	(Commutativity)
$\bar{\bar{a}} = a$	(Double-negation)
$a + a = a$	(Idempotence)
$aa = a$	(Idempotence)
$\overline{(a + b)} = \bar{a}\bar{b}$	(DeMorgan's laws)
$\overline{ab} = \bar{a} + \bar{b}$	(DeMorgan's laws)
$\bar{a} + a = 1$	(Inverse)
$\bar{a}a = 0$	(Inverse)
$a + (b + c) = (a + b) + c$	(Associativity)
$a(bc) = (ab)c$	(Associativity)
$a(b + c) = ab + ac$	(Distributivity)
$a + bc = (a + b)(a + c)$	(Distributivity)

We can verify these properties by building the truth tables for them. We will only build tables for two of the properties: associativity ($a(bc) = (ab)c$) and DeMorgan's law ($\overline{ab} = \bar{a} + \bar{b}$).

a	b	c	$a(bc)$	$(ab)c$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

a	b	$\overline{ab}$	$\bar{a} + \bar{b}$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Normal forms for Boolean expressions

It can be difficult, by looking at a pair of Boolean expressions, to determine whether they "do the same thing" (i.e., produce the same output for all identical inputs). One way to check this would be to generate truth tables for both. Another way is to transform both into a *normal form*, a particular expression structure such that any two equivalent expressions $e_1 = e_2$ will also be *identical* $e_1 \equiv e_2$. The two main normal forms for Boolean expressions are SUM-OF-PRODUCTS (also known as "disjunctive normal form": DNF) and PRODUCT-OF-SUMS (also known as "conjunctive normal form": CNF).

Sum-of-Products: An expression in SoP normal form has the general structure

$$x_1 x_2 \ldots x_n + \overline{x_1} x_2 \ldots x_n + \ldots$$

where each product includes *all* variables, and any of the individual variables may be negated, but otherwise no negations are allowed on anything "larger" than a variable. Obviously, since $a+a = a$, there is no point in including multiple copies of a particular product, so all of the products should be distinct, in terms of which variables are negated.

Products of possibly-negated variables are sometimes called *minterms*, for reasons that only computer architecture nerds understand or care about.

Any Boolean expression can be put into SoP form by using the distributive identies and DeMorgan's laws to move + outward, and negation inward. For example:

$\overline{ab}(b + \overline{c})$	
$\overline{ab}b + \overline{ab}\overline{c}$	(By distributivity)
$(\overline{a} + \overline{b})b + (\overline{a} + \overline{b})\overline{c}$	(By DeMorgan's laws)
$\overline{a}b + \overline{b}b + \overline{ac} + \overline{b}\overline{c}$	(By distributivity)
$\overline{a}b + 0 + \overline{ac} + \overline{b}\overline{c}$	(By absorbtion)
$\overline{a}b + \overline{ac} + \overline{b}\overline{c}$	(Identity of +)

At this point, we need to ensure that every product includes *all* the variables in our system. In order to add the missing variables, we note that $bc = (a + \overline{a})bc = abc + \overline{a}bc$. In other words, if a variable is missing, then we expand that product into a sum in which it is present in both its negated and unnegated forms. Doing this on the above, and eliminating the resulting duplicates, gives

$$\overline{a}\overline{b}\overline{c} + \overline{a}b\overline{c} + \overline{a}bc + a\overline{b}\overline{c}$$

Although this process works, there is another way: we can generate out the truth table for an expression and then simply *read off* the SoP form from it:

a	b	c	$\overline{ab}(b+\overline{c})$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

For each row whose result is a 1, we form a product based on the values of the inputs for that row: if an input is 0 then the variable is negated in the product, otherwise it is left un-negated. For example, the first row above would correspond to:

$$\overline{a}\overline{b}\overline{c}$$

Note that when a, b, and c take on the values 0,0,0, the result of this product is 1, as it should be according to the table.

To form the complete SoP, we simply take the sum of all rows (rows whose result is 0 are ignored). Thus, working from the above table, we have:

$$\overline{a}\overline{b}\overline{c} + \overline{a}b\overline{c} + \overline{a}bc + a\overline{b}\overline{c}$$

This is the same result we got above, if we ignore the order of variables within a product, and the order of products within a sum (since both sums and products are commutative and associative).

A formal definition of Boolean algebra

We have defined Boolean algebras somewhat informally, above. Here we will give a more formal definition, and show that all the properties you'd expect can be derived from the formal definition.

Definition 0.3 Boolean Algebra A Boolean algebra is a set B containing two distinguished elements 1 and 0, with a pair of operators $\cdot$ and $+$ that fulfil the

following conditions:

$a + b \in S$	$(\forall a, b \in S)$
$a \cdot b \in S$	$(\forall a, b \in S)$
$1 \cdot a = a \cdot 1 = a$	(Identity of $\cdot$)
$0 + a = a + 0 = a$	(Identity of $+$)
$a + b = b + a$	(Commutativity of $+$)
$a \cdot b = b \cdot a$	(Commutativity of $\cdot$)
$a + (b + c) = (a + b) + c$	(Associativity of $+$)
$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	(Associativity of $\cdot$)
$a \cdot (b + c) = a \cdot b + a \cdot c$	(Distributivity of $\cdot$ over $+$)
$(a + b) \cdot c = a \cdot c + b \cdot c$	(Distributivity of $\cdot$ over $+$)
$a + (b \cdot c) = (a + b) \cdot (a + c)$	(Distributivity of $+$ over $\cdot$)
$(a \cdot b) + c = (a + c) \cdot (b + c)$	(Distributivity of $+$ over $\cdot$)
$\forall a \in B : \exists \bar{a} : a + \bar{a} = 1 \wedge a \cdot \bar{a} = 0$ with $\bar{a}$ unique	

Note that we do not actually require 1 and 0 to be *distinct* elements, although for any $|B| > 1$ this will be the case.

Other interpretations of Booleans

We've defined a Boolean algebra, above, purely in terms of the operations it supports. There are, in fact, several algebraic structures that line up with this definition.

Those familiar with Java or C++ can think of the above as the interface, or abstract base class, for Boolean algebra. Now we are interested in looking at implementations.

Natural numbers mod *2:* If we take 1 and 0 to be themselves, and $\cdot \equiv \times$, $\bar{x} \equiv 1 - x \mod 2$ we can get pretty close to building a Boolean algebra on $\mathbb{N}$ mod 2. It remains to define $+$. We can define $a \oplus b$ easily: $a \oplus b \equiv a + b \mod 2$. Since

$$a + b = (a \oplus b) \oplus ab$$

we have

$$a + b \equiv a + b + ab \mod 2$$

giving us a complete "implementation" of a Boolean algebra with two elements.

The Boolean algebra of sets: Although we have only covered sets informally, it is possible to use a set S to form an n-ary Boolean algebra, where $n = 2^{|S|}$. We

take

$$B = \mathscr{P}(A)$$
$$0 \equiv \varnothing$$
$$1 \equiv S$$
$$a + b \equiv a \cup b$$
$$ab \equiv a \cap b$$
$$\overline{a} \equiv S - a$$

We can verify, from the axioms of set theory, that this implementation conforms to the rules for Boolean algebras.

A Boolean algebra based on a set offers varying degrees of "trueness". $1(= S)$ is the "purely true" value, and $0(= \varnothing)$ is "purely false", but between these are the various proper subsets of S which represent various degrees of trueness. We can say that a is "more true" than b iff $b \subset a$ but note that there are subsets for which neither is more true than the other. For example, if we take $S = \{x, y, z\}$ then

$$\varnothing \subset \{x\}$$
$$\{x\} \subset \{x, y\}$$
$$\{x\} \subset \{x, z\}$$
$$\{x, y\} \not\subset \{x, z\} \quad \text{and} \quad \{x, z\} \not\subset \{x, y\}$$

Logic and Proofs

We are interested here in the *theory of proofs*: how proofs are constructed, what constitutes a "valid" proof, and what sort of things can be (dis)proved. We will begin by looking at *inference rules*.

Inference rules

An inference rule describes a conditional statement of the form "if P then Q". The inference rule for this statement would be written:

$$\frac{P}{Q}$$

An inference rule should be read as "if everything *above* the line holds, then whatever is *below* the line holds". As another example, we can state "if both P and Q, then R" by writing

$$\frac{P \quad Q}{R}$$

We use the term "holds" rather than "is true" because we may want to use inference rules to establish properties other than just logical truth.

The elements above the line (here, P and Q) *premises* while the statement below the line is called the *conclusion*.

An inference rule without anything above the line is called an *axiom*, and it always holds:

$$\frac{}{P}$$

Often we will give names to our rules, so that we can refer back to them later, by writing the name next to the rule, like so:

$$\text{Rule name}\ \frac{P}{Q}$$

Derivations: Given a collection of inference rules, we can chain them together, matching conclusions to premises. For example, suppose we are given the rules

$$\mathrm{A}_1\ \frac{}{P} \qquad \mathrm{A}_2\ \frac{}{Q} \qquad \mathrm{R}_1\ \frac{P \quad Q}{R} \qquad \mathrm{R}_2\ \frac{R}{S}$$

From these we can construct a *derivation*, showing that, within this collection of rules, S holds:

$$\mathrm{R}_2\ \frac{\mathrm{R}_1\ \dfrac{\mathrm{A}_1\ \dfrac{}{P} \quad \mathrm{A}_2\ \dfrac{}{Q}}{R}}{S}$$

A derivation like this constitutes a (graphical) proof that the final conclusion follows from the rules, by showing exactly how to reach it from the axioms.

Logical connectives

The basic machinery of inference rules, although useful, is so simple as to make many common tasks unbearably tedious. For example, suppose we want to represent the compound proposition "P *and* Q". We could construct, for each pair of primitive propositions P_1 and P_2 the compound proposition $P_1 and P_2$, all defined by rules of the form

$$\frac{P_1 \quad P_2}{P_1 and P_2}$$

but this is obviously too cumbersome. Instead, we add a number of *logical connectives*, operators which allow us to combine propositions into compound structures. For example, the logical connective $\wedge$ represents *and*, and is defined by the single rule

$$\wedge\mathrm{R}\ \frac{P \quad Q}{P \wedge Q}$$

That is, if P and Q are both true (provable) separately, then $P \wedge Q$ is true. Put another way, to prove $P \wedge Q$, prove P and then prove Q.

The "R" in the name of the rule will be explained later.

A related connective is $\vee$, logical *or*. It is defined by two rules that allow us to choose which alternative to focus on, and which to throw away. This captures our intuition that to prove a disjunction, we only have to prove *half* of it.

$$\vee\mathrm{R}_1\ \frac{P}{P \vee Q} \qquad \vee\mathrm{R}_2\ \frac{Q}{P \vee Q}$$

The primitive proposition $\top$ is equivalent to true; it is true everywhere, and requires no proof:

$$\top\mathrm{R}\ \frac{}{\top}$$

When we come to logical implication, we run into a problem, and have to take a detour.

If you're noticing the similarity to Boolean algebra, you might be wondering where the equivalent to Boolean negation ("not") is. *Not* is not as important in our system of logic, because we are interested in what is *provable*. In fact, we explicitly deny the Boolean idea that everything which is *not* true must be false. Some things are unprovable, and we make no claims about their truth or falsity in that case.

Hypothetical judgments

Often we want to allow ourselves the ability to reason not just about what *is* true in some system, but also about what might happen if something was *assumed* to be true. That is, we want to be able to say, "prove Q, *assuming* P". A hypothetical judgment gives us this ability.

A hypothetical judgment has the form $\Gamma \vdash C$ where Γ is a comma-separated list of *assumptions* and C is the conclusion. The *assumption* rule allows us to make use of one of the assumptions to prove an identical conclusion:

$$\text{Assume}\ \frac{}{\Gamma, P \vdash P}$$

That is, if we assume P, then we can prove P.

Γ, here, as elsewhere, just stands for "everything else". We might have other assumptions that we are not using right now, but we still need to keep track of them. When you do something with a particular assumption, every other assumption is grouped into Γ and just carried along for the ride.

Structural rules: If you want to think of Γ as just an unordered collection of assumptions in which duplicates are ignored (assuming P more than once is no different from assuming it once) you will be fine, but technically the *structure* of the collection of assumptions (typically called the "context") is itself defined by rules.

The *rule of exchange* says that the order of the assumptions does not matter:

$$\text{Exchange}\ \frac{\ldots P_1, P_2 \ldots \vdash C}{\ldots P_2, P_1 \ldots \vdash C}$$

The *weakening rule* says that adding assumptions does not change the provability of something; if C can be concluded assuming Γ, then C can also be concluded assuming Γ and P:

$$\text{Weakening}\ \frac{\Gamma \vdash C}{\Gamma, P \vdash C}$$

This also says that you can "throw away" irrelevant elements of the context if you want.

Finally, the *rule of contraction* says that duplicate assumptions can be contracted into a single assumption:

$$\text{Contract}\ \frac{\Gamma, P \vdash C}{\Gamma, P, P \vdash C}$$

All of these work together to capture our intuition that all the assumptions we have are just floating around; we don't care about their ordering, and irrelevant assumptions don't matter, and assuming the same thing more than once is no different from assuming it more than once.

Right and left rules

Now we are in a position to give the rule for $\rightarrow$: we can prove $P \rightarrow Q$ by *assuming* P and then trying to prove Q with that assumption. That is, P implies Q if having a proof of P enables us to prove Q. We capture this with the $\rightarrow$R rule:

$$\rightarrow\text{R}\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \rightarrow Q}$$

This is the first rule with any assumptions, and thus we have to have Γ. In order to prove $P \rightarrow Q$, we *add* P to whatever assumptions we already had.

Becuase we now have assumptions, we need to modify our rules for $\wedge, \vee$ and $\top$ to show how Γ should be handled.

- To prove $P \wedge Q$ with some assumptions, we just prove P and Q independently, but with the *same* assumptions. That is

$$\wedge\text{R}\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

- Similarly, to prove $P \vee Q$ we choose the particular branch we want to focus on, but the assumptions stay the same:

$$\vee\text{R}_1\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \vee\text{R}_2\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}$$

- If $\top$ is true everywhere, then it should be true with any assumptions:

$$\top\text{R}\frac{}{\Gamma \vdash \top}$$

All of the rules we've seen so far have operated on the proposition to the *right* of the $\vdash$, which is why we call them *right* rules. There is a corresponding set of *left* rules that tell us how to operate on the connectives when they occur to the *left* of the $\vdash$.

- Assuming $\top$ doesn't get us anything, because $\top$ is always true, so there is no $\top$L rule. If you want, you can think of an extension of the contraction rule as

$$\frac{\Gamma \vdash P}{\Gamma, \top \vdash P}$$

I.e., $\top$ is "redundant" in *any* context.

- Assuming $P \wedge Q$ is no different from assuming P and Q separately, so the $\wedge$L rule just lets us split up a $\wedge$:

$$\wedge\text{L}\frac{\Gamma, P, Q \vdash G}{\Gamma, P \wedge Q \vdash G}$$

- $\vee$ is a little trickier. If we assume $P \vee Q$ and we want to show G, we don't know which of P or Q might be used, so we have to show that we can prove

G *both* when we have just P, and when we have just Q. Thus, the $\vee$L rule splits the derivation into two sub-derivations, each with a *different* context.

$$\vee\text{L}\frac{\Gamma, P \vdash G \qquad \Gamma, Q \vdash G}{\Gamma, P \vee Q \vdash G}$$

- If we want to show that G follows from $P \to Q$ we have to show two things: we have to show that we can actually *use* $P \to Q$, that is, we have to have a P in the first place; second, we have to show that once we have used $P \to Q$ to get a Q, we can use a Q to get a G. This gives us the $\to$L rule

$$\to\text{L}\frac{\Gamma, P \to Q \vdash P \qquad \Gamma, Q \vdash G}{\Gamma, P \to Q \vdash G}$$

  Note that when showing that we can get a P, we keep $P \to Q$ around as an assumption, because it might be necessary for the process of proving P. By the weakening rule, we can always throw it away if it isn't.

- If we *assume* false then anything could be true! That is, if we are in a universe where false is true, water might not be wet, or the sky might be orange, or anything else. Hence, in the $\bot$L rule, we can use a $\bot$ assumption to prove *any* proposition P:

$$\bot\text{L}\frac{}{\Gamma, \bot \vdash P}$$

With this full set of rules, we can proceed to prove the logical versions of some (but not all) of the Boolean identities. For example, the associativity of $\vee$:

$$\to\text{R}\frac{\vee\text{L}\dfrac{\vee\text{R}_1\dfrac{\vee\text{R}_1\dfrac{\dfrac{}{A \vdash A}}{A \vdash (A \vee B)}}{A \vdash (A \vee B) \vee C} \qquad \vee\text{L}\dfrac{\vee\text{R}_1\dfrac{\vee\text{R}_2\dfrac{\dfrac{}{B \vdash B}}{B \vdash (A \vee B)}}{B \vdash (A \vee B) \vee C} \qquad \vee\text{R}_2\dfrac{\dfrac{}{C \vdash C}}{C \vdash (A \vee B) \vee C}}{(B \vee C) \vdash (A \vee B) \vee C}}{A \vee (B \vee C) \vdash (A \vee B) \vee C}}{A \vee (B \vee C) \to (A \vee B) \vee C}$$

(We've proved this in the left-to-right direction only. The corresponding proof that $(A \vee B) \vee C) \to A \vee (B \vee C)$ is left as an exercise for the reader. Generally, when we want to show that $P = Q$ this means constructing *two* proofs, one that $P \to Q$ and another that $Q \to P$. When we do examples, we will prove the left-to-right version, and leave the right-to-left version as an exercise.)

Since every proof of $P \to Q$ must start with $\to$R, we will normally skip this step and begin with the conclusion $P \vdash Q$.

(details to follow)

Negation

As mentioned above, negation is not as important here as it was in Boolean algebra. If we want to say that P is *not* true, we interpret this as "P is contradictory" and define

$$\neg P \equiv P \to \bot$$

That is, to show that P is not true, we must show that P is "carrying" $\bot$ inside it somehow.

Some of the Boolean identities for negation still apply, but others do not. For example, we can show that $\neg P \wedge P = \bot$:

$$\wedge\mathrm{L}\,\dfrac{\rightarrow\mathrm{L}\,\dfrac{\overline{P \rightarrow \bot, P \vdash P} \qquad \overline{P, \bot \vdash \bot}}{P \rightarrow \bot, P \vdash \bot}}{(P \rightarrow \bot) \wedge P \vdash \bot} \qquad \bot\mathrm{L}\,\dfrac{}{\bot \vdash P \rightarrow \bot \wedge P}$$

But only one direction of the $\vee$ version, $P \vee \neg P \rightarrow \top$ is provable:

$$\vee\mathrm{L}\,\dfrac{\top\mathrm{R}\,\dfrac{}{P \vdash \top} \qquad \top\mathrm{R}\,\dfrac{}{P \rightarrow \bot \vdash \top}}{P \vee (P \rightarrow \bot) \vdash \top}$$

$$\vee\mathrm{R}_1\,\dfrac{\dfrac{?}{\vdash P}}{\top \vdash P \vee (P \rightarrow \bot)} \quad \text{or} \quad \vee\mathrm{R}_2\,\dfrac{\dfrac{?}{\vdash P \rightarrow \bot}}{\top \vdash P \vee (P \rightarrow \bot)}$$

Similarly, the double-negation identity $P = \neg\neg P$ is provable

$$\rightarrow\mathrm{R}\,\dfrac{\rightarrow\mathrm{L}\,\dfrac{\overline{P, P \rightarrow \bot \vdash P} \qquad \overline{\bot \vdash \bot}}{P, P \rightarrow \bot \vdash \bot}}{P \vdash (P \rightarrow \bot) \rightarrow \bot} \qquad \bot\mathrm{L}\,\dfrac{}{\bot \vdash (P \rightarrow \bot) \rightarrow \bot}$$

But only half of DeMorgan's laws are

$$\rightarrow\mathrm{L}\,\dfrac{\dfrac{?}{(P \wedge Q) \rightarrow \bot \vdash P \wedge Q} \qquad \bot\mathrm{L}\,\dfrac{}{\bot \vdash (P \rightarrow \bot) \vee (Q \rightarrow \bot)}}{(P \wedge Q) \rightarrow \bot \vdash (P \rightarrow \bot) \vee (Q \rightarrow \bot)}$$

$$\wedge\mathrm{R}\,\dfrac{\rightarrow\mathrm{R}\,\dfrac{\rightarrow\mathrm{L}\,\dfrac{\vee\mathrm{R}_1\,\dfrac{\overline{(P \vee Q) \rightarrow \bot, P \vdash P}}{(P \vee Q) \rightarrow \bot, P \vdash P \vee Q} \qquad \dfrac{}{\bot \vdash \bot}}{(P \vee Q) \rightarrow \bot, P \vdash \bot}}{(P \vee Q) \rightarrow \bot \vdash (P \rightarrow \bot)} \qquad \rightarrow\mathrm{R}\,\dfrac{\rightarrow\mathrm{L}\,\dfrac{\vee\mathrm{R}_2\,\dfrac{\overline{(P \vee Q) \rightarrow \bot, Q \vdash Q}}{(P \vee Q) \rightarrow \bot, Q \vdash P \vee Q} \qquad \dfrac{}{\bot \vdash \bot}}{(P \vee Q) \rightarrow \bot, Q \vdash \bot}}{(P \vee Q) \rightarrow \bot \vdash (Q \rightarrow \bot)}}{(P \vee Q) \rightarrow \bot \vdash (P \rightarrow \bot) \wedge (Q \rightarrow \bot)}$$

All of these differences in fact stem from our inability to prove

$$\dfrac{?}{\vdash P \vee (P \rightarrow \bot)}$$

In Boolean algebra, it is safe to assume that every Boolean expression evaluates to either 0 or 1 (or, generalized to Boolean algebras of more than two values, that every x has a negation $\overline{x}$). But logically, there are propositions for which we have neither a proof, nor a contradiction. Asserting $P \vee (P \rightarrow \bot)$ amounts to saying "every proposition has either a proof or a refutation" but we know this is not true. There are some propositions, those representing unsolved problems, which have neither.

Universal and existential quantifiers

Our logical system as we have presented it allows us to construct derivations about "atomic" propositions, propositions that are true or false in and of themselves, but we lack the ability to make statements *about things*. That is, we cannot say (for example) "every natural number is either odd or even", because this is a statement about *things* (natural numbers, in this case). We will remedy this by adding *variables* and *predicates* to our system, and then examine their use.

Variables for us (typically lowercase letters $x, y,$ etc.) work like they do in algebra: they stand for *unknowns*. A *predicate* is a proposition which is *about* some value or values. We usually write predicates as $P(x)$ although other forms are possible. For example, $x \in \mathbb{N}$ is a predicate, signifying the property of x being a natural number.

Universal quantification: Suppose we have the statement "every natural number is either odd or even". We could write this logically as

$$x \in \mathbb{N} \to \mathsf{even}(x) \vee \mathsf{odd}(x)$$

but this is rather vague. How do we know we are looking to prove this for "every" x, and not just for some particular x. We introduce the *universal quantifier* to make our intention clear:

$$\forall x\colon x \in \mathbb{N} \to \mathsf{even}(x) \vee \mathsf{odd}(x)$$

$\forall x$ introduces x as a universally quantified variable. When we say $\forall x\colon P(x)$ we are saying the P is true for *all* x, not just *some* x.

(Note that since the form $\forall x\colon P(x) \to Q(x)$ comes up a lot, we have a shorthand for it: *guarded quantifiers*. We could write this as $\forall x, P(x)\colon Q(x)$.)

How can we prove $\forall x\colon P(x)$? Intuitively, proving a $\forall$ is like proving a very big $\wedge$. That is, if x can be any of $x_1, x_2, \ldots x_n$ then we have to show $P(x_1) \wedge P(x_2) \wedge \cdots \wedge P(x_n)$. Obviously we can't do this, because the range of values for x might be infinite. Instead, the right-rule for $\forall x$ replaces x with a new, completely unknown variable x'. If we can construct a proof about x' despite knowing nothing about it, then obviously we could plug in any *actual* value for x' and the proof would still hold. Thus, a *generic* proof serves as a valid proof of a universal:

$$\forall\mathrm{R}\frac{\Gamma \vdash P(x')}{\Gamma \vdash \forall x\colon P(x)} \quad \text{where } x' \text{ is a new variable}$$

If we assume $\forall x\colon P(x)$ then we could assume P for any *particular* value we like. (This is equivalent to using $\wedge$L to split up the $\wedge$ implicit in the $\forall$, and then using the weakening rule to throw away all the $P(-)$ assumptions we don't need.) Thus, the $\forall$L rule lets us assume P for any value we like:

$$\forall\mathrm{L}\frac{\Gamma, \forall x\colon P(x), P(t) \vdash G}{\Gamma, \forall x\colon P(x) \vdash G} \quad \text{or} \quad \forall\mathrm{L}\frac{\Gamma, P(t_1), P(t_2), \ldots \vdash G}{\Gamma, \forall x\colon P(x) \vdash G}$$

(The first version keeps the $\forall x\colon P(x)$ assumption around so that we can use it to generate additional $P(t')$'s if we need them. The second version just constructs all the $P(t)$ assumptions we need all at once, and then uses weakening to throw away the $\forall$.)

We can use this to show that, for example, the successor of every natural number is itself a natural number:

$$\forall\text{R}\;\frac{\to\text{R}\;\dfrac{\text{Nat-S}\;\dfrac{\overline{x' \in \mathbb{N} \vdash x' \in \mathbb{N}}}{x' \in \mathbb{N} \vdash S(x') \in \mathbb{N}}}{x' \in \mathbb{N} \to S(x') \in \mathbb{N}}}{\forall x\colon x \in \mathbb{N} \to S(x) \in \mathbb{N}}$$

Similarly, if we define odd and even in terms of each other

$$\text{Even-0}\;\frac{}{\mathsf{even}(0)} \qquad \text{Odd-1}\;\frac{}{\mathsf{odd}(1)} \qquad \text{Even-S}\;\frac{\mathsf{odd}(n)}{\mathsf{even}(S(n))} \qquad \text{Odd-S}\;\frac{\mathsf{even}(n)}{\mathsf{odd}(S(n))}$$

we can show that the success of every even number is odd:

$$\forall\text{R}\;\frac{\to\text{R}\;\dfrac{\text{Odd-S}\;\dfrac{\overline{\mathsf{even}(x') \vdash \mathsf{even}(x')}}{\mathsf{even}(x') \vdash \mathsf{odd}(S(x'))}}{\mathsf{even}(x') \to \mathsf{odd}(S(x'))}}{\forall x\colon \mathsf{even}(x) \to \mathsf{odd}(S(x))}$$

(Note that a proof of $\forall x, P(x)\colon Q(x)$ will always start with $\forall$R, followed by $\to$R. We'll normally leave these as implicit, and start the proof at $P(x') \vdash Q(x')$.)

Suppose we want to show that $(\forall x\colon P(x)) \to (\forall y\colon P(y))$. This basically amounts to showing that variable *names* do not matter. We begin by doing $\to$R and then $\forall$R (with $y = y'$) implicitly:

$$\frac{?}{\forall x\colon P(x) \vdash P(y')}$$

When we apply the $\forall$L rule, we get to *choose* t. In particular, we can choose t to be the new variable y' (y' must be new when we introduce it, but after that, it exists and we can use it to construct values.) So we have

$$\forall\text{L}\;(t = y')\;\frac{\overline{P(y') \vdash P(y')}}{\forall x\colon P(x) \vdash P(y')}$$

This hopefully illustrates the difference between the "new" variable x' and the *value* t: x' must be new when we introduce it, while t must be built up from "old" (existing) things: variables, or, if needed, values like 0 or constructors like the successor.

Existential quantification: Suppose we are asked to prove

$$x \in \mathbb{N} \wedge x > 1$$

What exactly are we asking for here? Surely not $\forall x\colon x \in \mathbb{N} \wedge x > 1$, because that is not true (0 is not > 1). Instead, we are asking whether there is *any* x that will satisfy these conditions. We call this *existential quantification* and write it

$$\exists x\colon x \in \mathbb{N} \wedge x > 1$$

To prove an existential, we need to supply an actual *value* which makes the statement true:

$$\exists\mathrm{R}\,\frac{\Gamma \vdash P(t)}{\Gamma \vdash \exists x\colon P(x)}$$

(As $\forall$ corresponds to a kind of implicit $\wedge$, $\exists$ corresponds to an implicit $\vee$. This rule amounts to choosing the branch of the $\vee$ to pursue.)

If we *assume* that "there exists an x such that $P(x)$" then we can assume P about anything. To capture this notion, we assume $P(x')$ where x' is new:

$$\exists\mathrm{L}\,\frac{\Gamma, P(x') \vdash G}{\Gamma, \exists x\colon P(x) \vdash G}$$

We can double-check our left and right rules by proving that we can prove an existential by assuming it:

$$\exists\mathrm{L}\,\frac{\exists\mathrm{R}\ (y = x')\ \dfrac{\dfrac{}{P(x') \vdash P(x')}}{P(x') \vdash \exists y\colon P(y)}}{\exists x\colon P(x) \vdash \exists y\colon P(y)}$$

Rules for formal logic

With $\forall$ and $\exists$ we have completed our system for doing formal logic. The rules are summarized in figure 1.

Proof by induction

The generic method of proving a $\forall x$ is often too weak; although we can make use of assumptions about x, we cannot exploit knowledge about the *structure* of x. For example, if we have $\forall x, x \in \mathbb{N}\colon P(x)$ we have almost no way of using the structure of $\mathbb{N}$ in our proof. *Proof by induction* is a "trick" for proving a universal that relies on the recursive structure of the set we are operating in.

For example, in $\mathbb{N}$ every natural number has the property that it is either a 0 or a successor to some other natural number. Strip off enough successors and eventually we'll get down to 0. Proof by induction *starts* at 0, by showing that $P(0)$ is directly true. It then proceeds to show how to "strip off" a success in the proof of $P(S(n))$ to get down to a proof of $P(n)$. Because we could, given any particular natural number n, repeat this process enough times to reach the 0-case, we could use this method to *construct* a particular chain of proofs ($P(n)$ is true because $P(n-1)$ is true because ...because $P(0)$ is true directly). Proof by induction is in fact an *algorithm* for building proofs, but the existence of such an algorithm is itself a proof of the *universal* property!

Put another way, if we can write a program that takes any $\mathbb{N}$ as an input, and returns a proof of P for that input as its output, then the program itself serves as proof that P holds for all $\mathbb{N}$.

Figure 1: Rules for logical propositions

$$\text{Assume}\ \frac{}{\Gamma, P \vdash P}$$

$$\top\text{R}\ \frac{}{\Gamma \vdash \top} \qquad \bot\text{L}\ \frac{}{\Gamma, \bot \vdash P}$$

$$\wedge\text{R}\ \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \qquad \wedge\text{L}\ \frac{\Gamma, P, Q \vdash G}{\Gamma, P \wedge Q \vdash G}$$

$$\vee\text{R}_1\ \frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \vee\text{R}_2\ \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \qquad \vee\text{L}\ \frac{\Gamma, P \vdash G \qquad \Gamma, Q \vdash G}{\Gamma, P \vee Q \vdash G}$$

$$\to\text{R}\ \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q} \qquad \to\text{L}\ \frac{\Gamma, P \to Q \vdash P \qquad \Gamma, Q \vdash G}{\Gamma, P \to Q \vdash G}$$

$$\forall\text{R}\ \frac{\Gamma \vdash P(x')}{\Gamma \vdash \forall x\colon P(x)} \quad \text{where } x' \text{ is a new variable}$$

$$\forall\text{L}\ \frac{\Gamma, \forall x\colon P(x), P(t) \vdash G}{\Gamma, \forall x\colon P(x) \vdash G} \quad \text{or} \quad \forall\text{L}\ \frac{\Gamma, P(t_1), P(t_2), \ldots \vdash G}{\Gamma, \forall x\colon P(x) \vdash G}$$

$$\exists\text{R}\ \frac{\Gamma \vdash P(t)}{\Gamma \vdash \exists x\colon P(x)} \qquad \exists\text{L}\ \frac{\Gamma, P(x') \vdash G}{\Gamma, \exists x\colon P(x) \vdash G} \quad \text{where } x' \text{ is a new variable}$$

Consider the recursive definition of $\text{double}(x) = 2x$:

$$\text{Dbl-Z}\ \frac{}{\text{double}(0) = 0} \qquad \text{Dbl-S}\ \frac{\text{double}(x) = y}{\text{double}(S(x)) = S(S(y))}$$

Suppose we want to prove that

$$\forall x \in : \exists y\colon \text{double}(x) = y$$

This amounts to showing that double is *total*; that is, it is defined for all possible inputs. If we try to prove this generically, we will get stuck. But we *can* prove it inductively, by induction on x. We divide the proof into two cases, one for 0 and one for the successor:

- Base case ($x = 0$): Then we have

$$\exists\text{R}\ (y = 0)\ \frac{\text{Dbl-Z}\ \dfrac{}{\text{double}(0) = 0}}{\exists y\colon \text{double}(0) = y}$$

- Inductive case ($x = S(x')$): Here we get to *assume* the "inductive hypothesis", the smaller version of our proof statement. In this case, our IH is

$$\text{IH}\ \frac{}{\exists y' : \text{double}(x') = y'}$$

  That is, while we are trying to *prove* that $S(x')$ has a double, we get to *assume* that x' has a double, and it is y'. So all we have to do is reconstruct y from y' by following the Dbl-S rule:

$$\exists R\ (y = S(S(y'))) \ \frac{\text{Dbl-S}\ \dfrac{\text{IH}\ \dfrac{}{\text{double}(x') = y'}}{\text{double}(S(x')) = S(S(y'))}}{\exists y : \text{double}(S(x')) = y}$$

QED

In order to construct an inductive proof, we need several to identify several elements:

- The variable of induction. We need to choose the variable (which must be universally quantified) that our inductive proof is going to be about. If there is only one universally quantified variable (as in the above proof) then the "choice" is easy. If there are two or more (e.g., in the proof of $\forall x, y \in \mathbb{N} : \exists z : x + y = z$) then we make our decision by looking at the definition. The inductive variable must a variable which the definition recurses on. E.g., $x + y$ is defined by recursion on x, so we would choose x.
- The base case: The base case must be true directly, because we cannot rely on the IH to prove it. Again, usually the choice of the base case will be dictated by the base case of the definition. The base case for + is $0 + y = y$, so we would choose 0 to be the value of the variable of induction in the base case.
- The inductive case: Similarly, the value of the variable of induction in the inductive case will be dictated by the definition. It must be "larger" than some smaller value, for example, $x = S(x')$.
- The inductive hypothesis: Having chosen the value for the inductive case, we rephrase the original proof statement in terms of the "smaller" value to give the inductive hypothesis.

  As a further example, consider the inductive proof that

$$\forall x, y \in \mathbb{N} : \exists z : x + y = z$$

(Proof that + is total.)

- There are two universally quantified variables, x and y, so our choice of the variable of induction will be between them. + is defined by recursion on its first operand (here, x) so x will be our variable of induction.

- The base case for + is $0 + y = y$ so we choose $x = 0$ as our base case.
- The recursive case for + is $S(x') + y = S(z')$ so we choose $x = S(x')$ to be our inductive case.
- In the inductive case, the inductive hypothesis is just the original proof statement, rephrased to be about the "smaller" x':

$$\forall y \in \mathbb{N}\colon \exists z\colon x' + y = z$$

Paragraph proofs: Derivation-style proofs can become cumbersome, especially for inductive proofs. An alternate presentation for proofs is *paragraph-style proofs*, where the steps of the proof are presented linearly. This sometimes requires careful bookkeeping (as when different branches of the proof have different goals and assumptions) but gives us the opportunity to gloss over the more obvious steps.

(details to follow)

Set Theory

Here we are interested in the properties of *sets*. A set is an unordered collection of objects, all sharing some important property or properties, with an important restriction: *no duplicates* are present in a set. For example, this is a set:

$$\{1, 2, 3, 4\}$$

But this is not:

$$\{1, 2, 3, 4, 3\}$$

Sets are normally written between curly-braces.

Since writing out the elements of a set is often tedious (for large sets), and sometimes impossible (for infinite sets), we also use *set builder notation*:

$$\{x^2 \mid x \text{ is a prime number }\}$$

would give us the set of all squares of primes (i.e., $\{4, 9, 25, 49, \ldots\}$).

Some sets are so common that they have special names (we've seen some of these before); these are listed in figure 2.

Set	Description
$\mathbb{N}$	Natural numbers (i.e., whole numbers ≥ 0)
$\mathbb{Z}$	Integers
$\mathbb{Q}$	Rational numbers (fractions)
$\mathbb{R}$	Real numbers
$\mathbb{C}$	Complex numbers

Figure 2: Special sets

When we need to write a variable for an unknown set, we'll usually write it as an uppercase letter.

Sometimes you'll see the notation $\mathbb{Z}^*$ for "integers ≥ 0" and $\mathbb{Z}^+$ for "integers ≥ 1". Likewise, $\mathbb{N}_k$ is sometimes used for natural numbers modulo k, i.e., $\{0, 1, \ldots, k-1\}$.

Often when we talk about sets we need to make it clear what kinds of things they can contain. We do this by stating what the *universe of discourse* is. The universe is just another set, denoted $\mathscr{U}$, but every other set we talk about must draw its elements from $\mathscr{U}$.

I.e., as described below, every set must be a *subset* of $\mathscr{U}$.

Haskell note:

In Haskell, we'll use lists as sets, so a set of elements of type `a` will have type `[a]`. We'll have to take care to ensure that our lists are always sorted and de-duplicated. We will provide you with a function `distinct` which will take an arbitrary list and transform it into a valid set:

```
ghci> distinct [1,3,2,1,3,5]
[1,2,3,5]
```

We will use the `elem` function as the equivalent to $\in$:

```
ghci> 3 `elem` [1,2,3,5]
True
```

And finally, we can use list comprehensions as an analogue to set builder notation. For example,

$$\{x^2 \mid x \in \{1 \ldots 10\}\}$$

becomes

```
[x^2 | x <- [1..10]]
```

Operations on sets

Perhaps the fundamental operation on sets is *membership*. The logical proposition $e \in S$ states "e is a member of set S". Likewise, $e \notin S$ states "e is not a member of S". Since $\in$ is a proposition, we can make logical statements using it:

$$e \in S \rightarrow e \in \mathscr{U}$$

This expresses our statement above that all the elements of the sets we are talking about must be elements of the universe.

If we want to relate two sets to each other, we can ask, to what extent do they overlap? If, given two sets A and B, we find that every element of A is also an element of B then we say that "A is a *subset* of B" and write this as $A \subseteq B$. Logically,

$$A \subseteq B \Leftrightarrow \forall a \in A : a \in B$$

If $A \subseteq B$ then we can also say that B is a *superset* of A.

Sometimes we write $A \subset B$ to mean that A is a "proper" subset of B, that is, $A \subseteq B \wedge A \neq B$.

There are a few easy-to-verify tautologies about set membership and the subset relation:

$$a \in A \land A \subseteq B \to a \in B \qquad \text{(Definition of } \subseteq\text{)}$$
$$A \subseteq B \land B \subseteq A \to A = B \qquad \text{(Symmetry)}$$
$$A \subseteq B \land B \subseteq C \to A \subseteq C \qquad \text{(Transitivity)}$$
$$A \subseteq A, \text{ for any set } A \qquad \text{(Reflexivity)}$$

Given two sets, there are a number of operations we can perform that will result in a new set:

$$A \cup B \qquad \text{(Union)}$$
$$A \cap B \qquad \text{(Intersection)}$$
$$A \setminus B \qquad \text{(Set difference)}$$
$$A \ominus B \qquad \text{(Symmetric difference)}$$

Each of these can be defined by a set builder on $\mathscr{U}$ and the logical equivalent to the above descriptions:

Operation	Set builder expression
$A \cup B$	$\{e \mid e \in \mathscr{U} \land (e \in A \lor e \in B)\}$
$A \cap B$	$\{e \mid e \in \mathscr{U} \land (e \in A \land e \in B)\}$
$A \setminus B$	$\{e \mid e \in \mathscr{U} \land (e \in A \land e \notin B)\}$
$A \ominus B$	$\{e \mid e \in A \cup B \land e \notin A \cap B\}$

Figure 3: Set operations

The *cardinality* of a set A, written $|A|$ roughly defines the "size" of the set. For finite sets, this is simply the number of (unique) elements in the set.

If $A \subseteq B$ then we have $|A| \leq |B|$. Similarly, if $A \subset B$ then $|A| < |B|$.

Some properties relating cardinality and the above set operations are easy to derive:

For sets constructed via the set builder notation, note that we have

$$|\{\ldots \mid s_1 \in S_1, s_2 \in S_2, \ldots, s_n \in S_n, \ldots\}| \leq \prod_{i=1}^{n} |S_i|$$

Interestingly, there are different "sizes" of infinite sets. For example, $|\mathbb{N}| < |\mathbb{R}|$; the set of real numbers has *more* elements than the set of natural numbers, even though both are infinite!

Powersets: The *powerset* of a set A is the *set of all subsets* of A. That is

$$\mathscr{P}(A) = \{A' \mid A' \subseteq A\}$$

For example,

$$\mathscr{P}(\{1, 2\}) = \{\varnothing, \{1\}, \{2\}, \{1, 2\}\}$$

Figure 4: Cardinality of set operations

$$|A \cup B| = |A| + |B| - |A \cap B|$$
$$|A \cap B| \leq |A| \wedge |A \cap B| \leq |B|$$
or equivalently
$$|A \cap B| \leq \max(|A|, |B|)$$
$$|A \setminus B| = |A| - |A \cap B|$$
$$|A \ominus B| = |A \cup B| - |A \cap B|$$

Note that the powerset of *any* set always includes the empty set. This is true even of the empty set itself:

$$\mathscr{P}(\varnothing) = \{\varnothing\}$$

Note that $\{\varnothing\} \neq \varnothing$. The former is a set containing a single element, while the later contains no elements.

One way to view the construction of the powerset is as if each element of the input set had a "switch" attached to it, labeled IN/OUT. We can construct any particular subset by flipping the switches: those whose elements are IN will be in the subset, and those which are OUT will not. I.e., we have something like

$$\{1_{\text{IN}}, 2_{\text{OUT}}, 3_{\text{OUT}}, 4_{\text{IN}}\} \longrightarrow \{1, 4\}$$

By looking at all possible configurations of switches, we can see all the possible subsets.

Here, for example, are all the subsets of $\{1, 2, 3\}$:

1	2	3	Subset
OUT	OUT	OUT	$\varnothing$
IN	OUT	OUT	$\{1\}$
OUT	IN	OUT	$\{2\}$
IN	IN	OUT	$\{1, 2\}$
OUT	OUT	IN	$\{3\}$
IN	OUT	IN	$\{1, 3\}$
OUT	IN	IN	$\{2, 3\}$
IN	IN	IN	$\{1, 2, 3\}$

We can also generate the powerset by an inductive formulation:

$$\mathscr{P}(\varnothing) = \{\varnothing\} \qquad \text{(Base case)}$$

Let $A = \{x\} \cup A'$ with $x \notin A'$. Then

$$\mathscr{P}(A) = \mathscr{P}(\{x\} \cup A') = \mathscr{P}(A') \cup \{\{x\} \cup P \mid P \in \mathscr{P}(A')\} \qquad \text{(Inductive case)}$$

That is, at each inductive step, we choose an arbitrary element of A, remove it producing A', recursively construct $\mathscr{P}(A')$, and then add x to every set in $\mathscr{P}(A')$ and union that with the (unchanged) $\mathscr{P}(A')$. At some point we will remove the last element, at which point $A' = \varnothing$ and the recursive powerset will use the base case.

Cardinality of $\mathscr{P}$: How many subsets does a given set have? Looking at the above table, we see that for a set with three elements, there are eight possible subsets, including the empty set. For each element in the original set, it's "switch" can be in one of two states. Thus, by the rule of product, the total number of possible configurations is $2 \times 2 \times 2 = 2^3 = 8$. In general, if $n = |A|$ then $2^n = |\mathscr{P}(A)|$. Figuring out how many subsets to expect is a good way to check your work, if you are trying to generate all the subsets.

Equality between sets: Since sets are unordered, we define equality between sets to mean "containing the same elements". That is,

$$A = B \text{ iff } (\forall a \in A \colon a \in B) \land (\forall b \in B \colon b \in A)$$

or equivalently,

$$A = B \text{ iff } A \subseteq B \land B \subseteq A$$

Two sets with the same cardinality (possibly infinite, but of the "same size" infinite) are *isomorphic* even if they are not equal. Isomorphism just means that we can construct a pairing of elements between the two sets, such that each element from A is paired with a single unique element from B, and each element of B is paired with a single unique element of A. Because we can always move "back and forth" between the two sets, we can convert any operation on A into an operation on B, and vice versa, without loss of information. Thus, two isomorphic sets can be regarded as being "extensionally equivalent", because anything we can do on the one can be done on the other.

Relations on Sets

Relations allow us to construct "connections" between sets in various ways. In this section, we will examine the various kinds of relations and their properties.

Cross products: The *cross product* of two sets A and B is defined to be

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

For example,

$$\{1, 2\} \times \{3, 4\} = \{(1, 3), (2, 3), (1, 4), (2, 4)\}$$

The cross product of two sets essentially gives us every possible pairing of items from the first set with items from the second set. It should be relatively obvious from the definition that $|A \times B| = |A||B|$.

We can easily extend the cross product to three (or more) sets:

$$A \times B \times C = \{(a, b, c) \mid a \in A, b \in B, c \in C, \}$$
$$A \times B \times C \times D = \{(a, b, c, d) \mid a \in A, b \in B, c \in C, d \in D\}$$

and so forth.

Note that if $A = \varnothing$ or $B = \varnothing$ then $A \times B = \varnothing$. This generalizes to n-dimensional products: if any of the sets composing the product is empty, then the entire product is empty, too.

If it isn't obvious why $A \times \varnothing = \varnothing$, consider that $|A \times \varnothing| = |A||\varnothing|$ and $|\varnothing| = 0$, and that $\varnothing$ is the *only* set which has cardinality 0.

An n-dimension product includes n *projection* functions, $p_1 \ldots p_n$, which can extract the individual elements of the product. That is, given $x \in A \times B$ (i.e., $x = (a, b)$ for some $a \in A, b \in B$) we have

$$p_1(x) = a$$
$$p_2(x) = b$$

Sometimes we will need a cross product of a set with itself: $A \times A$. In this case, we will denote this with the shorthand A^2 (generalizes to $A \times A \times \ldots = A^n$) unless this would be ambiguous.

Relations

A *relation* R between two (or more) sets is just a subset of their cross product. That is,

$$\mathsf{R} \subseteq A \times B$$

Thus, a relation can be thought of as some kind of "pairing" of some elements from A to some elements of B. Note that in the most general sense, it's possible for a single element of A to relate to 0, 1, or many elements of B, and vice versa. We will later examine specialized relations in which this is restricted in some way.

Although we can indicate that a and b are related via R by writing $(a, b) \in \mathsf{R}$ it is usually easier to just write $a \,\mathsf{R}\, b$. That is, we treat R as defining a logical proposition: given a and b, we can ask, is it the case that $a \,\mathsf{R}\, b$? and the answer will be Yes or No. This implies that we can connect statements about relational membership using the logical connectives $\wedge$, $\vee$, etc.

We can generalize these "two-dimensional" relations to relations between three or more sets. Higher-dimensional relations are the basis for *relational* database systems, so we'll look at them, briefly, later.

For a two-dimensional relation $\mathsf{R} \subseteq A \times B$, we usually refer to A as the *domain* and B as the *range*. Also, since saying that R is defined between A and B by writing $\mathsf{R} \subseteq A \times B$ is rather cumbersome, we will write $\mathsf{R} : A \times B$.

Given $\mathsf{R} : A \times B$ and a particular $a \in A$, we can ask for the set of all $b \in B$ that are related to a:

$$\{b \mid a \,\mathsf{R}\, b\}$$

Similarly, given a b we can ask for all the related a's.

Properties of relations

We can classify relations by the properties they have.

- If $A = \varnothing$ or $B = \varnothing$ then $\mathsf{R} = \varnothing$ for any $\mathsf{R} : A \times B$.

Given two relations $\mathsf{R} : A \times B$ and $\mathsf{S} : B \times C$ we can form the *composition* of the two relations, $\mathsf{R} \circ \mathsf{S} : A \times C$:

$$\mathsf{R} \circ \mathsf{S} = \{(a, c) \mid a \,\mathsf{R}\, b, b \,\mathsf{S}\, c\}$$

Given a relation $\mathsf{R} : A \times B$ we can define its *converse*, $\mathsf{R}^c : B \times A$ as

$$\mathsf{R}^c = \{(b, a) \mid a \,\mathsf{R}\, b\}$$

or equivalently

$$b \,\mathsf{R}^c\, a \text{ iff } a \,\mathsf{R}\, b$$

Property	Logical definition
Functionality	$\forall a : \exists b_1, b_2 : (a \,\mathsf{R}\, b_1 \wedge a \,\mathsf{R}\, b_2) \to b_1 = b_2$
Totality	$\forall a \in A : \exists b \in B : a \,\mathsf{R}\, b$
Partiality	$\exists a \in A : \neg\exists b \in B : a \,\mathsf{R}\, b$

Figure 5: Properties of relations on $A \times B$

A functional relation is one in which, if $a \,\mathsf{R}\, b$ then b is *unique*; i.e., there is only one b related to that particular a. (But note that functionality does not require that *every* $a \in A$ have a related b; if this is not the case then the function is *partial*.) Since functional relations "act like" functions, we typically name them f, g, etc., and write $a \,\mathsf{f}\, b$ as $f(a) = b$.

Note that there is some ambiguity in the use of the term "partial". Some sources consider *all* functional relations to be partial, and the total functions as a (proper) subset of these. Others consider the sets of partial and total functions to be distinct.

A relation which is both functional and total is a *total function*; this is what we normally think of a function as being: we can plug in *any* $a \in A$ and be guaranteed to get one, and only one, $b \in B$ out as a result. Note that totality is simply the negation of partiality; a not-partial function is, by-definition, total.

Often we'll specify that a relation f is a partial function from A to B by writing $f : A \rightharpoonup B$. Similarly, if f is total we will write $f : A \to B$

Many interesting properties are only meaningful for relations defined from a set to itself; i.e., $\mathsf{R} : A^2$. Figure 6 lists these properties, together with their logical definitions.

Property	Logical definition
Reflexivity	$\forall x \in \mathscr{U} : x \,\mathsf{R}\, x$
Symmetry	$\forall a, b : a \,\mathsf{R}\, b \to b \,\mathsf{R}\, a$
Antisymmetry	$\forall a, b : (a \,\mathsf{R}\, b \wedge b \,\mathsf{R}\, a) \to a = b$
Antisymmetry (def. 2)	$\forall a, b, a \neq b : \neg(a \,\mathsf{R}\, b \wedge b \,\mathsf{R}\, a)$
Transitivity	$\forall a, b, c : (a \,\mathsf{R}\, b \wedge b \,\mathsf{R}\, c) \to a \,\mathsf{R}\, c$

Figure 6: Properties of relations on $A \times A$

Note that symmetry and antisymmetry are *not* negations of each other. Symmetry requires that all relations "go both ways", while antisymmetry requires that relations between *distinct* elements only go one way. The distinction

"distinct" is important. It is in fact possible (and even easy) to construct a relation that is both symmetric and antisymmetric, or both not-symmetric and not-antisymmetric.

What are the converses of these properties? E.g., if we form the converse of a transitive relation, is the result transitive?)

Special relations: functions

For a functional relation $f : A \to B$ we can ask, for some $A' \subset A$, what is the *image* of f under A'? That is, what is the subset of B that would result if we were to plug in every value in A' and collect all the results? Formally, the definition of the image is

$$f(A') = \{f(a') \mid a' \in A\}$$

(Note that we are "overloading" our function call notation to work on sets as well as values.)

For functional relations (particularly for total functions), there are three interesting properties:

Property	Logical definition
One-to-one	$\forall a_1, a_2 \in A : a_1 \mathrel{R} b \wedge a_2 \mathrel{R} b \to a_1 = a_2$
Onto	$\forall b \in B : \exists a \in A : a \mathrel{R} b$
One-to-one correspondence	Injective $\wedge$ Surjective

Figure 7: Properties of functions

One-to-one is sometimes called *injective*. Onto is sometimes called *surjective*. And one-to-one correspondence is sometimes called *bijective*. But it's a lot harder to remember the meaning of these names.

A one-to-one function *preserves distinctness*; that is, for distinct inputs it will always produce distinct outputs.

An onto function $f : A \to B$ is one whose image under A is all of B. That is, nothing of B is "left out".

A one-to-one correspondence both preserves distinctness, and "covers" all of B. This implies that every $a \in A$ maps to a distinct $b \in B$, and similarly, every $b \in B$ is the result of some $f(a)$. Thus, for any b we can find its *inverse* a; one-to-one correspondences are thus *invertible*. For any f that is a one-to-one correspondence, there is a (unique) inverse function f^{-1}, which is also a one-to-one correspondence.

A one-to-one correspondence is sometimes called an *isomorphism* and if an isomorphism exists between two sets then they are *isomorphic* in the sense defined above. This implies that the sets have the same cardinality.

An isomorphism from a set to itself is called a *permutation*. For example, we can define the isomorphism $f : \{1, 2, 3, 4\} \to \{1, 2, 3, 4\}$ by

$$f(1) = 4$$
$$f(2) = 1$$
$$f(3) = 2$$
$$f(4) = 3$$

Special relations: operators

A *binary operator* is a ternary (3-dimensional) relation which functionally maps pairs to a set. That is, an operator $\star : A \times B \to C$. We will only consider operators defined on a single set. For example, natural number addition is $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Operators can have several interesting properties:

Property	Logical definition
Commutative	$\forall a, b : a \star b = b \star a$
Associative	$\forall a, b, c : (a \star b) \star c = a \star (b \star c)$
Distributive (of $\star$ over $\boxplus$)	$\forall a, b, c : a \star (b \boxplus c) = (a \star b) \boxplus (a \star c)$

Figure 8: Properties of operators

Special relations: equivalences and partial orders

Equivalence relations: A relation which is reflexive, symmetric, and transitive is call an *equivalence* relation. An equivalence relation is, roughly speaking, one that "acts like" equality: everything is equivalent to itself, you can "swap the sides" of an equivalence and it will still hold, and equivalnce "propagates" so that if $a \equiv b$ and $b \equiv c$ then we know that $a \equiv c$.

The simplest example of an equivalence relation is equality itself. Some more interesting examples include:

- Set isomorphism. If we define $A \equiv B$ to mean "A is isomorphic to B" then $\equiv$ is an equivalence relation. Every set is isomorphic to itself (reflexivity), if two sets are isomorphic then there are isomorphisms in both directions (symmetry), and the composition of two isomorphisms is also an isomorphism (transitivity).
- Natural number equivalence *mod* p.

Given an equivalence relation $\equiv$ on a set S, we can construct the *quotient* of S on $\equiv$:

(details to follow)

We can use this to *partition* S into its *equivalence classes*. An equivalence class is a set whose elements are all equivalent according to some equivalence relation. We denote the equivalence class of a with respect to $\equiv$ as $[a]_\equiv$ and define it as

$$[a]_\equiv = \{b \mid a \equiv b\}$$

Note that because of reflexivity, we have $\forall a : a \in [a]_\equiv$.

If we split a set up into all its equivalence classes, this *partitions* the set. Technically, any partition *defines* an equivalence relation, just as any equivalence relation defines a partition. The partition of S on $\equiv$ is denoted

$$\mathscr{P}_\equiv(S) = \{[a]_\equiv \mid a \in S\}$$

(details to follow)

To be more precise, a partition P of S is a set of sets that satisfies the following properties:

- Nontrivial: $\varnothing \notin P$
- Exhaustive: $S = \bigcup_{p \in P} p$
- Disjoint: $\forall p_1 \in P, p_2 \in P, p_1 \neq p_2 : p_1 \cap p_2 = \varnothing$.

Given a partition P on S, we can also (re)construct the equivalence relation that it defines:

$$\equiv = \{(a, b) \mid C \in P, a \in C, b \in C\}$$

or, with slightly different phrasing

$$\equiv = \bigcup_{C \in P} C \times C$$

Partial orders: A relation which is reflexive, *anti*symmetric, and transitive is called a *partial order*. Partial orders can be thought of as a generalization of the idea of "less-than or equal to". $a \preceq b$ can be thought of as stating that *a precedes b* (or $a = b$, as reflexivity requires that every element be related to itself). But note that, unlike the $\leq$ relation which is total, partial orders are, as their name implies, *partial*; it is not necessarily the case that, for every $a_1, a_2 \in A$, either $a_1 \preceq a_2$ or $a_2 \preceq a_1$. Some elements may be completed unrelated, neither preceding nor succeeding each other. (Not surprisingly, an order that is not partial is called a *total* order.)

Question: is it possible for a partial order to contain a "loop"? That is, can we have a situation $a_1 \preceq a_2 \preceq \ldots \preceq a_n \preceq a_1$? In fact, the answer is no; although proving this rigarously is left as an exercise for the reader, the intuitive explanation is that the transitive property will "collapse" the loop so that $a_1 = a_2 = \ldots = a_n$. This loop-free property makes partial orders very useful in algorithms; we can always follow the relation without worrying about getting stuck in a loop.

The converse of a partial order is also a partial order. To prove this, we simply look at the above results which show that the converse preserves each of the three required properties. This implies that we can take any partial order and construct the converse partial order which "goes the other way".

Least and greatest elements: Sometimes a partial order will have a (unique) element that is $\preceq$ every element, or an element that is $\succeq$ every other element. If these exist (and they need not both exist) they are referred to as the *least* and greatest elements according to the order. Note that the least and greatest elements are defined by the combination of the partial order *and* the set it is defined on. It makes no sense to ask for the least/greatest element of a set without also specifying the partial order.

Logically, we can define the presence of the least/greatest elements as

$$\exists a_0 : \forall a \in A : a_0 \preceq a \qquad \text{(Least)}$$
$$\exists a_1 : \forall a : a \preceq a_1 \qquad \text{(Greatest)}$$

If they exist, the least/greatest elements are guaranteed to be *unique*: i.e., if a_1 and a_2 are both least elements, then $a_1 = a_2$. A partial order which has both least and greatest elements is called a *bounded* partial order. This implies that we can follow the relation, in either direction, and eventually stop (again, algorithmically, this is a useful property as it implies that an algorithm which follows the relation will terminate, and not run forever!).

Can you prove this?

We can similarly define the least and greatest elements of some subset of A, A'. These will come in handy when defining least-upper- and greatest-lower-bounds, so we will define the functions:

$$\text{least} : (\mathscr{P}(A) \setminus \varnothing) \to A$$
$$\text{greatest} : (\mathscr{P}(A) \setminus \varnothing) \to A$$

to give us the least/greatest element of any non-empty subset of A.

Least-upper- and Greatest-lower-bounds: Given two elements a and c, not necessarily ordered with respect to each other, we can ask whether there exists a b such that $b \preceq a$ and $b \preceq c$. If this is the case then b is called a *lower bound* of a and c. Similarly, if $a \preceq b$ and $c \preceq b$ then b is an *upper bound* of a and c.

Two elements may have more than one upper or lower bound. For example, in the total order $\leq$: $\mathbb{N}^2$ we have $1 \leq 3$, $1 \leq 4$ but also $2 \leq 3$, $2 \leq 4$. Thus both 1 and 2 are lower bounds for 3 and 4. We will define the functions $\text{lb} : A^2 \to \mathscr{P}(A)$ and $\text{ub} : A^2 \to \mathscr{P}(A)$ to give us the complete sets of lower/upper bounds for any pair of elements:

$$\text{lb}(a, c) = \{b \mid b \preceq a \wedge b \preceq c\}$$
$$\text{ub}(a, c) = \{b \mid a \preceq b \wedge c \preceq b\}$$

Note that for a given a, c it is possible for $\text{lb}(a, c) = \varnothing$ or $\text{ub}(a, c) = \varnothing$ or both.

We are now in a position to find the *least upper* and *greatest lower* bounds of a pair of elements. Intutively, the least upper bound is the "smallest" upper bound, the element which is closest to the original two. Similarly, the greatest lower bound is the "largest" (again, closest) of all the lower bounds. To find

them, we simply compose the lb/ub functions with the least/greatest functions:

$$\begin{aligned} \text{lub} &: A^2 \to A \\ \text{lub}(a, c) &= \text{least}(\text{ub}(a, c)) \\ \text{glb} &: A^2 \to A \\ \text{glb}(a, c) &= \text{greatest}(\text{lb}(a, c)) \end{aligned}$$

Note that if $\text{ub}(a, c) = \varnothing$ then the least upper bound is not defined, and similarly for the greatest lower bound.

We can extend lb and ub to work on sets rather than pairs of elements. This allows us to ask for the least/greatest upper/lower bound of a set of elements. However, this extension does not add any new properties, and constructing it is left as an exercise for the reader.

A partial order over a set in which *every* pair of elements has both a least-upper and greatest-lower bound is called a *lattice* (and if it also has least and greatest elements, then a *bounded lattice*).

Well-orders: A set with a partial order is *well-ordered* if every non-empty subset has a least element by the order. This is stronger than a partial order (which does not require least elements at all) but weaker than a lattice

A well-ordered set has the useful property that every element except the greatest element, if it exists, has a *successor*. We define $\text{succ}(n)$ as

$$\text{succ}(n) = \text{least}(\{n' \mid n \prec n'\})$$

I.e., the successor is the least element of the subset consisting of everything greater than n.

Here, $a \prec b$ means $a \preceq b \wedge a \neq b$.

Other Topics

Type Theory

Type theory is concerned with the properties of *types* and the expressions and computations which they categorize. It should be noted in particular that in type theory, a *type* can be abstractly defined as any interesting property of a computation that can be extracted from a *static* description of that computation. I.e., types are everything that you can find out at compile-time; run-time is concerned with values.

As an example of a type theory, consider the problem of classifying the types of expressions in a simple programming language consisting of

- Integer literals `0`,`1`, etc.
- String literals `"string"`
- The arithmetic operators `+` and `*`

- The string concatenation operator `++`
- The function `show` which converts a integer expression to a string.
- The function `len` which finds the length of a string expression, as an integer.

We give inference rules which define the judgment `e :: τ` meaning that e has type τ.

We use the familar Haskell notation `::`, but bear in mind that literature on type theory will typically use a single colon $e : \tau$.

$$\text{Int}\ \frac{n \in \mathbb{Z}}{n :: \texttt{Int}} \qquad \text{Str}\ \frac{}{\texttt{"..."} :: \texttt{String}}$$

$$\text{Add}\ \frac{e_1 :: \texttt{Int} \quad e_2 :: \texttt{Int}}{e_1 + e_2 :: \texttt{Int}} \qquad \text{Mul}\ \frac{e_1 :: \texttt{Int} \quad e_2 :: \texttt{Int}}{e_1 * e_2 :: \texttt{Int}}$$

$$\text{Cat}\ \frac{e_1 :: \texttt{String} \quad e_2 :: \texttt{String}}{e_1 \texttt{ ++ } e_2 :: \texttt{String}}$$

$$\text{Show}\ \frac{e :: \texttt{Int}}{\texttt{show}(e) :: \texttt{String}} \qquad \text{Len}\ \frac{e :: \texttt{String}}{\texttt{len}(e) :: \texttt{Int}}$$

Given this theory, we can ask whether a given expression has a type, and if so, what type it is, by constructing a derivation:

(details to follow)

Let us make our theory a bit more interesting. Suppose we want to add a let-in construct to allow local name binding. Its syntax will be `let` n `=` e `in` e', where the name n can be used (and is bound to e) within e'. Although we can infer the type of $e :: \tau$, how can we determine the type of e'? We need some way of keeping track, within e' of the fact that $n :: \tau$. In fact, we will reuse the contexts and hypothetical judgements of proof theory: $\Gamma \vdash e :: \tau$ is a judgment stating that if the typing judgments in Γ are assumed, then e will have type τ. Similarly, the "contents" of Γ will be typing judgments. We augment all of our original rules to take this into account, and then add an addiitonal rule for `let-in`:

$$\text{Var}\ \frac{}{\Gamma, v :: \tau \vdash v :: \tau} \qquad \text{Int}\ \frac{n \in \mathbb{Z}}{\Gamma \vdash n :: \texttt{Int}} \qquad \text{Str}\ \frac{}{\Gamma \vdash \texttt{"..."} :: \texttt{String}}$$

$$\text{Add}\ \frac{\Gamma \vdash e_1 :: \texttt{Int} \quad \Gamma \vdash e_2 :: \texttt{Int}}{\Gamma \vdash e_1 + e_2 :: \texttt{Int}} \qquad \text{Mul}\ \frac{\Gamma \vdash e_1 :: \texttt{Int} \quad \Gamma \vdash e_2 :: \texttt{Int}}{\Gamma \vdash e_1 * e_2 :: \texttt{Int}}$$

$$\text{Cat}\ \frac{\Gamma \vdash e_1 :: \texttt{String} \quad \Gamma \vdash e_2 :: \texttt{String}}{\Gamma \vdash e_1 \texttt{ ++ } e_2 :: \texttt{String}}$$

$$\text{Show}\ \frac{\Gamma \vdash e :: \texttt{Int}}{\Gamma \vdash \texttt{show}(e) :: \texttt{String}} \qquad \text{Len}\ \frac{\Gamma \vdash e :: \texttt{String}}{\Gamma \vdash \texttt{len}(e) :: \texttt{Int}}$$

$$\text{Let } (v \text{ is free in } e')\ \frac{\Gamma \vdash e :: \tau \quad \Gamma, v :: \tau \vdash e' :: \tau'}{\Gamma \vdash (\texttt{let } v = e \texttt{ in } e') :: \tau'}$$

Our new Var rule allows us to use any of the assumed types in Γ to make a typing judgment about a variable. Our new Let rule lets us *add* such assumptions to the context, but scoped within the "body" of the let; the part after `in`. You should read the Let-rule as "if e has type τ in Γ, and *assuming* that v has type τ leads to e' having type τ', then we can conclude that `let` $v = e$ `in` e' has type τ'". Just as, at run-time, the *value* of e will be substituted into e' in place of the variable v, here, at type-checking type, we substitute the *type* of e into the type of e', whenever we would need the type of v.

As above, we can construct typing derivations to verify the type of an expression:

(details to follow)

Abstract Algebra

Category Theory

Category theory can be viewed as "object-oriented mathematics". What we mean by that is that, while traditional mathematics is very concerned with how structures are *built*, and their behavior is derived from the structure, category theory seeks to describe concepts purely in terms of behavior, with minimal reference to how things "look inside". Thus, it can be viewed as analogous to the object-oriented goal of building software purely around opaque interfaces; we don't need to know how a class *works* internally, we just need to know how we can interact with it.

To make this rather vague definition more concrete, we will look at onto and one-to-one functions, with the goal of building up definitions for "onto" and "one-to-one" that never mentions the *elements* of the sets involved.

I am indebted to M.A. Arbib and E.G. Manes' *Arrows, Structures, and Functors: The Categorial Imperative* for this style of introduction.

Recall that the definition of an onto function is one whose image under its domain is its range:

$$f : A \to B \text{ is onto iff } f(A) = B$$

A more explicit definition would be

$$\forall b \in B : \exists a \in A : f(a) = b$$

Suppose $f : A \to B$ is onto, and suppose that we have two other functions $g, h : B \to C$, and finally, suppose it is the case that

$$\forall a \in A : g(f(a)) = h(f(a))$$

Because f is onto, this implies that we are "testing" the equality of g and h on every b:

$$\forall b \in B : g(b) = h(b)$$

But if this is the case, and g and h produce identical results for all inputs from their mutual domain, we can just say $g = h$ because they are indistinguishable.

What happens if we try this with a f' that is *not* onto? In that case $\exists b \in B : \neg\exists a \in A : f'(a) = b$; i.e., there are at least some b's $\in B$ that will not be produced

by f' as output. This means that when we say that $\forall a \in A : g(f'(a)) = h(f'(a))$ we are no longer "testing" g and h on all possible inputs, and so we cannot conclude $g = h$. Thus, another way of stating that f is onto is

$$(\forall a \in A : g(f(a)) = h(f(a))) \Rightarrow g = h$$

How can we describe this property without ever refering to the elements of A and B? We will exploit the behavior of onto functions under *composition* with other functions. Again, function composition is defined as

$$f : B \to C$$
$$g : A \to B$$
$$(f \cdot g) : A \to C$$
$$(f \cdot g)(x) = f(g(x))$$

So we will rewrite the above definition as simply

$$f \text{ is onto iff } g \cdot f = h \cdot f \Rightarrow g = h$$

Notice how this definition makes no reference to the elements of A or B, or even to the domains/ranges of the functions. And yet this definition will work just as well as the element-centric one.

What about one-to-one functions? Taking a guess, is it the case that

$$f \text{ is onto iff } f \cdot g = f \cdot h \Rightarrow g = h$$

In fact, the answer is Yes, as we will demonstrate. Recall that $f : B \to A$ being one-to-one implies that

$$b_1 \neq b_2 \Rightarrow f(b_1) \neq f(b_2)$$

Again, suppose we have $g, h : C \to B$ such that

$$\forall c \in C : f(g(c)) = f(h(c))$$

Then we can conclude, based purely on f being one-to-one, that $g = h$. How? Suppose $g \neq h$; then there must be some c such that $g(c) \neq h(c)$. But if that were the case, then $f(g(c)) \neq f(h(c))$ if f is one-to-one. So f being one-to-one forces the results of g and h to be identical.

Again, by using function composition, we can rewrite

$$\forall c \in C : f(g(c)) = f(h(c)) \Rightarrow g = h$$

to

$$f \text{ is one-to-one iff } f \cdot g = f \cdot h \Rightarrow g = h$$

The fact that the principle behind onto turns into something different, but still valid, when we "do it backwards" is a central result of category theory: every valid principle has a *dual* that can be found by simply swapping all the domains and codomains.

This development demonstrates the general idea behind category theory: to describe the behavior of function-like mappings based purely on how they interact with each other, without any reference to what they "do" internally.

Categories, objects and morphisms

The category Hask

One use for category theory is in providing a semantics for type theory. Here, we will look at the category Hask. The objects of Hask are the types in Haskell, while the morphisms are the functions between those types.