disco Documentation

Release 1.0

disco team

Jul 10, 2018

Contents

CHAPTER 1

Quick Tutorial

Disco is a small yet expressive, pure functional programming language designed especially to be used in the context of a discrete mathematics course. Right now it is in a rough prototype stage; this tutorial is probably only useful for developers who already have some knowledge of functional programming.

1.1 Getting started

After you have been added as a collaborator to the disco github repository, get the source code via SSH:

```
git clone git@github.com:disco-lang/disco.git
```

If you are not a collaborator on the repository you can also get the source code via HTTPS:

```
git clone https://github.com/disco-lang/disco.git
```

Make sure you have the stack tool installed. Then navigate to the root directory of the disco repository, and execute

```
stack setup
stack build
```

(This may take quite a while the first time, while `stack` downloads and builds all the dependencies of `disco`.)

After building disco with `stack build`, to run the disco REPL (Read-Eval-Print Loop), type `stack exec disco` at a command prompt. You should see a disco prompt that looks like this:

```
Disco>
```

1.2 Arithmetic

As a computational platform for learning discrete mathematics, at the core of disco is of course the ability to compute with numbers. However, unlike many other languages, disco does not support real (*aka* floating-point) numbers at

all—they are not typically needed for discrete mathematics, and omitting them simplifies the language quite a bit. To compensate, however, disco has sophisticated built-in support for rational numbers.

1.2.1 Basic arithmetic

To start out, you can use disco as a simple calculator. For example, try entering the following expressions, or others like them, at the `Disco>` prompt:

- `2 + 5`
- `5 - 8`
- `5 * (-2)`
- `(1 + 2)(3 + 4)`
- `2 ^ 5`
- `2 ^ 5000`
- `4 .- 2`
- `2 .- 4`

The last two expressions use the saturating subtraction operator, `.-`, which takes two numeric operands, a and b, and returns $a - b$ if $a > b$, and 0 otherwise. Note that unlike regular subtraction, the result of a saturating subtraction will always be a natural number.

Also notice that it is not always necessary to write `*` for multiplication: as is standard mathematical notation, we may often omit it, as in `(1 + 2)(3 + 4)`, which means the same as `(1 + 2) * (3 + 4)`. (For precise details on when the asterisk may be omitted, see the discussion in the section on functions.) Notice also that integers in disco may be arbitrarily large.

Now try these:

- `3/7 + 2/5`
- `2 ^ (-5)`

The results may come as a bit of a surprise if you are already used to other languages such as Java or Python, which would yield a floating-point (*i.e.* real) number; as mentioned before, disco does not support floating-point numbers. However, rational numbers can still be entered using decimal notation. Try these expressions as well:

- `2.3 + 1.6`
- `1/5.`
- `1/7.`

Disco automatically picks either fractional or decimal notation for the output, depending on whether any values with decimal points were used in the input (for example, compare `1/5` and `1/5.`, or `1./5`). Note that `1/7.` results in `0.[142857]`; can you figure out what the brackets indicate?

The standard `floor` and `ceiling` operations are built-in:

```
Disco> floor (17/3)
5
Disco> ceiling (17/3)
6
```

Just for fun, disco also supports standard mathematical notation for these operations via Unicode characters:

```
Disco>  17/3
5
Disco>  17/3
6
```

Integer division, which rounds down to the nearest integer, can be expressed using `//`:

```
Disco> 5 // 2
2
Disco> (-5) // 2
-3
```

`x // y` is always equivalent to `floor (x/y)`, but is provided as a separate operator for convenience.

The counterpart to integer division is `mod`, which gives the remainder when the first number is divided by the second:

```
Disco> 5 mod 2
2
Disco> (2^32) mod 7
4
Disco> (2^32) % 7
```

The `%` operator may also be used as a synonym for `mod`.

Finally, the `abs` function is provided for computing absolute value:

```
Disco> abs 5
5
Disco> abs (-5)
5
```

1.2.2 Advanced arithmetic

Disco also provides a few more advanced arithmetic operators which you might not find built in to other languages.

- The `divides` operator can be used to test whether one number evenly divides another. Try evaluating these expressions:
  - `2 divides 20`
  - `2 divides 21`
  - `(-2) divides 20`
  - `2 divides (-20)`
  - `7 divides (2^32 - 4)`
  - `(1/2) divides (3/2)`
  - `(1/5) divides (3/2)`
  - `1 divides 10`
  - `0 divides 10`
  - `10 divides 0`
  - `0 divides 0`

The last three expressions may be surprising, but follow directly from the definition: `a divides b` is true if there is an integer `k` such that `a*k = b`. For example, there is no `k` such that `0*k = 10`, so `0 divides 10` is false.

Note that a vertical line is often used to denote divisibility, as in $3 \mid 21$, but disco does not support this notation, since the vertical line is used for other things (and besides, it is typically not a good idea to use a visually symmetric operator for a nonsymmetric relation).

- The `choose` operator can be used to compute binomial coefficients. For example, `5 choose 2` is the number of ways to select two things out of five.
- The factorial function is available via standard mathematical notation:

```
Disco> 20!
2432902008176640000
```

- Square root (`sqrt`) and base-two logarithm (`lg`) functions are provided which round their result down to the nearest integer (remember that disco does not support arbitrary real numbers).

```
Disco> sqrt (299^2 + 1)
299
Disco> sqrt (299^2 - 1)
298
Disco> lg (2^35 + 7)
35
Disco> lg (2^35 - 1)
34
```

1.3 Types

Every value in disco has a *type*. Types play a central role in the language, and help guide and constrain programs. All the types in a program must match correctly (*i.e.* the program must *typecheck*) before it can be run. The type system has for the most part been designed to correspond to common mathematical practice, so if you are used to type systems in other programming languages (even other functional languages) you may be in for a surprise or two.

Disco can often infer the type of an expression. To find out what type disco has inferred for a given expression, you can use the `:type` command. For example:

```
Disco> :type 3
3 : ℕ
Disco> :type 2/3
2 / 3 : ℚ
Disco> :type [1,2,5]
[1, 2, 5] : List ℕ
```

The colon in `3 :  ℕ` can be read "has type" or "is a", as in "three is a natural number". A colon can also be used to give an explicit type to an expression, for example, when you want to specify a type other than what disco would infer. For example:

```
Disco> :type 3 + 5
3 + 5 : ℕ
Disco> :type (3 : Integer) + 5
(3 : ℤ) + 5 : ℤ
```

The above example shows that normally, disco infers the type of `3 + 5` to be a natural number, but we can force the `3` to be treated as an `Integer`, which in turn forces the whole expression to be inferred as an integer.

1.3.1 Primitive numeric types

Disco has four built-in primitive numeric types: natural numbers, integers, positive rationals, and rationals.

- The type of natural numbers, written `Natural`, `Nat`, `N`, or , includes the counting numbers $0, 1, 2, \ldots$.
- The type of integers, written `Integer`, `Int`, `Z`, or , includes the natural numbers as well as their negatives.
- The type of positive rationals, written `QP` or , includes all ratios of the form a/b where a and b are natural numbers, with $b \neq 0$.
- The type of rational numbers, written `Rational`, `Q` or , includes all ratios of integers.

In mathematics, it is typically not so common to think of the positive rationals $\mathbb{Q}^+$ as a separate set by themselves; but this is mostly for historical reasons and because of the way the development of rational numbers is usually presented. The natural numbers support addition and multiplication. Extending them to support subtraction yields the integers; then, extending these again to support division yields the rationals. However, what if we do these extensions in the opposite order? Extending the natural numbers to support division results in the positive rational numbers; then extending these with subtraction again yields the rationals. All told, the relationship between these four types forms a diamond-shaped lattice:

```
  Q
 / \
Z   Q+
 \ /
  N
```

Each type is a subset of the type or types above it. Going northwest in this diagram ($\mathbb{N} \to \mathbb{Z}$ or $\mathbb{Q}^+ \to \mathbb{Q}$) corresponds to adding negatives, that is, subtraction; going northeast ($\mathbb{N} \to \mathbb{Q}^+$ or $\mathbb{Z} \to \mathbb{Q}$) corresponds to adding reciprocals, that is, division.

Try evaluating each of the following expressions at the disco prompt, and also request their inferred type with the `:type` command. What type does disco infer for each? Why?

- `1 + 2`
- `3 * 7`
- `1 - 2`
- `1 / 2`
- `(1 - 2) / 3`

Going southeast in the lattice (getting rid of negatives) is accomplished with the absolute value function `abs`. Going southwest (getting rid of fractions) is accomplished with `floor` and `ceiling`.

Note that disco supports *subtyping*, that is, values of type `S` can be automatically "upgraded" to another type `T` as long as `S` is a "subtype" (think: subset) of `T`. For example, a natural number can be automatically upgraded to an integer.

```
Disco> (-1 : Z) + (3 : N)
2
Disco> :type (-1 : Z) + (3: N)
(-1 : ) + (3 : ) :
```

In the above example, the natural number `3` is automatically upgraded to an integer so that it can be added to `-1`. When we discuss functions later, we will see that this principle extends to function arguments as well: if a function is expecting an integer as input, it is acceptable to give it a natural number, since the natural number can be upgraded to an integer.

1.3.2 Modular types

In addition to the four primitive numeric types discussed above, disco also supports an infinite family of "modular" or finite types of the form $\mathbb{Z}_n$. For a given natural number n, the type $\mathbb{Z}_n$ consists of the n values $\{0, 1, 2, \ldots, n-1\}$. Addition, subtraction, multiplication, and exponentiation are performed $\pmod n$.

```
Disco> (3 + 5) : Z6
2
Disco> (3 - 5) : Z6
4
Disco> 2^20 : Z7
4
```

In addition, division can be performed as long as n is prime:

```
Disco> 1 / 5 : Z7   -- compute the inverse of 5 mod 7
3
Disco> 3 * 5 : Z7   -- sure enough, 3 * 5 is 1 mod 7
1
```

1.3.3 Other types

There are many other types built into disco as well—`Bool`, `Void`, `Unit`, `List`, product, and sum types, to name a few. These will be covered throughout the rest of the tutorial and appropriate places. For now, try executing these commands and see if you can guess what is going on:

- `:type false`
- `:type ()`
- `:type [1, 2, 3]`
- `:type [1, 2, -3]`
- `:type [1, 2, -3, 4/5]`
- `:type [[1,2], [3,4,5]]`
- `:type (1, true)`

1.4 Disco files and the disco REPL

For anything beyond simple one-off calculations that can be entered at the disco prompt, disco definitions may be stored in a file which can be loaded into the REPL.

1.4.1 Disco files

Disco files typically end in `.disco`. Here is a simple example:

Listing 1: example/basics.disco

```
approx_pi : Rational
approx_pi = 22/7
```

(continues on next page)

(continued from previous page)

```
increment : N -> N
increment n = n + 1
```

This file contains definitions for `approx_pi` and `increment`. Each definition consists of a *type signature* of the form `<name> : <type>`, followed by an equality of the form `<name> = <expression>`. Both parts of a definition are required; in particular, if you omit a type signature, disco will complain that the name is not defined. The example file shown above contains two definitions: `approx_pi` is defined to be the `Rational` number $22/7$, and `increment` is defined to be the function which outputs one more than its natural number input. (Functions and the syntax for defining them will be covered in much more detail in an upcoming section of the tutorial.)

The order of definitions in a `.disco` file does not matter; each definition may refer to any other definition in the whole file.

To load the definitions in a file into the disco REPL, you can use the `:load` command. After successfully loading a file, all the names defined in the file are available for use. For example:

```
Disco> :load example/basics.disco
Loading example/basics.disco...
Loaded.
Disco> approx_pi
22/7
Disco> increment 3
4
Disco> :type increment
increment : ℕ → ℕ
Disco> approx_pi + increment 17
148/7
```

1.4.2 Comments and documentation

Comments in disco can be written in one of two ways, using the same syntax as Haskell. Two consecutive hyphens `--` will cause disco to ignore everything until the next newline character; `{- ... -}` creates a multi-line comment causing disco to ignore everything in between `{-` and `-}`.

Listing 2: example/comment.disco

```
-- This is a comment
approx_pi : Rational
approx_pi = 22/7   -- an OK approximation

{- The following function is very complicated
   and took about three weeks to write.
   Don't laugh.
-}
increment : N -> N
increment n {- the input -} = n + 1 {- one more than the input -}
```

Comments can be placed anywhere and are literally ignored by disco. In many cases, however, the purpose of a comment is to provide documentation for a function. In this case, disco supports special syntax for *documentation*, which must be placed before the type signature of a definition. Each line of documentation must begin with `|||` (three vertical bars).

Listing 3: example/doc.disco

```
||| A reasonable approximation of pi.
approx_pi : Rational
approx_pi = 22/7   -- an OK approximation

||| Take a natural number as input, and return the natural
||| number which is one greater.
|||
||| Should not be used while operating heavy machinery.
-- This comment will be ignored.
increment : N -> N
increment n {- the input -} = n + 1 {- one more than the input -}

fizz : N
fizz = 1
```

When this file is loaded into the disco REPL, we can use the `:doc` command to see the documentation associated with each name.

```
Disco> :load example/doc.disco
Loading example/doc.disco...
Loaded.
Disco> :doc approx_pi
approx_pi :

A reasonable approximation of pi.

Disco> :doc increment
increment :  →

Take a natural number as input, and return the natural
number which is one greater.

Should not be used while operating heavy machinery.

Disco> :doc fizz
fizz :
```

Since `fizz` does not have any associated documentation, the `:doc` command simply shows its type.

1.4.3 Other REPL commands

The disco REPL has a few other commands which are useful for disco developers.

- `:parse` shows the fully parsed form of an expression.

  ```
  Disco> :parse 2 + [3,4 : Int]
  TBin Add (TNat 2) (TList [TNat 3,TAscr (TNat 4) TyZ] Nothing)
  ```

- `:desugar` shows the desugared core language term corresponding to an expression.

  ```
  Disco> :desugar [3,4]
  CCons 1 [CNum Fraction (3 % 1),CCons 1 [CNum Fraction (4 % 1),CCons 0 []]]
  ```

- `:pretty` shows the pretty-printed form of a term (without typechecking it).

```
Disco> :pretty 2 + [3,4:Int]
2 + [3, (4 : )]
```

1.5 Logic

1.5.1 Booleans

The type of booleans, written `B` or `Bool`, represents logical truth and falsehood. The two values of this type are written `true` and `false`. (For convenience `True` and `False` also work.)

- Logical AND can be written `and`, `&&`, or .
- Logical OR is written `or`, `||`, or .
- Logical negation (NOT) is written `not` or ¬.

```
Disco> true and false
false
Disco> true || false
true
Disco> not (true  true)
false
Disco> ¬ (false or false or false or true)
false
```

1.5.2 Equality testing

If you have two disco values of the type, in almost all cases you can compare them to see whether they are equal using =, resulting in a `Bool` value.

```
Disco> 2 = 5
false
Disco> 3 * 7 = 2*10 + 1
true
Disco> (3/5)^2 + (4/5)^2 = 1
true
Disco> false = false
true
```

The /= operator tests whether two values are *not* equal; it is just the logical negation of =.

1.5.3 Comparison

Again, in almost all cases values can be compared to see which is less or greater, using operators <, <=, >, or >=.

```
Disco> 2 < 5
true
Disco> false < true
true
Disco> (5 : Z7) < (9 : Z7)
false
```

(The last example is `false` because `(9 : Z7)` is equivalent to `(2 : Z7)`.)

Comparisons can also be chained; the result is obtained by comparing each pair of values according to the comparison between them, and taking the logical AND of all the results. For example:

```
Disco> 1 < 3 < 8 < 99
true
Disco> 2.2 < 5.9 > 3.7 < 8.8 > 1.0 < 9
true
```

1.6 Structural types

In addition to the primitive types covered so far, disco also has sum and product types which can be used to build up more complex structures out of simpler ones.

1.6.1 Product types

The product of two types, written using an asterisk $\star$ or Unicode times symbol $\times$, is a type whose values are ordered pairs of values of the component types. Pairs are written using standard ordered pair notation.

```
pair1 : N * Q
pair1 = (3, -5/6)

pair2 : Z5 × Bool
pair2 = (17 + 22, (3,5) < (4,2))

pair3 : Bool * (Bool * Bool)
pair3 = (true, (false, true))

pair4 : Bool * Bool * Bool
pair4 = (true, false, true)
```

`pair1` in the example above has type `N * Q`, that is, the type of pairs of a natural number and a rational number; it is defined to be the pair containing `3` and $-5/6$. `pair2` has type `Z5 × Bool` (using the alternate syntax $\times$ in place of $\star$), and contains two values: $(17 + 22) \pmod 5$, and the result of asking whether $(3, 5) < (4, 2)$.

```
Disco> pair2
(4, true)
```

Pairs are compared lexicographically, which intuitively means that the first component is most important, the second component breaks ties in the first component, and so on. For example, $(a, b) < (c, d)$ if either $a < c$ (in which case b and d don't matter) or if $a = c$ and $b < d$. This is why `(3,5) < (4,2)` evaluates to `true`. Of course, two pairs are equal exactly when their first elements are equal and their second elements are equal.

`pair3` shows that pairs can be nested: it is a pair whose second component is also a pair. `pair4` looks like an ordered triple, but in fact we can check that `pair3` and `pair4` are equal!

```
Disco> pair3 = pair4
true
```

Really, `pair4` is just syntax sugar for `pair3`. In general:

- The type `X * Y * Z` is interpreted as `X * (Y * Z)`.
- The tuple `(x,y,z)` is interpreted as `(x,(y,z))`.

This continues recursively, so, for example, `A * B * C * D * E` means `A * (B * (C * (D * E)))`. Put another way, disco really only has pairs, but appears to support arbitrarily large tuples by encoding them as right-nested pairs.

If you want *left*-nested pairs you can use explicit parentheses: for example, `(Bool * Bool) * Bool` is not the same as `Bool * Bool * Bool`, and has values such as `((false, true), true)`.

1.6.2 Sum types

If `X` and `Y` are types, their *sum*, written `X + Y` (or `X  Y`), is the disjoint union of `X` and `Y`. That is, values of type `X + Y` are either values of `X` or values of `Y`, along with a "tag" so that we know which it is. The possible tags are `left` and `right` (to indicate the type on the left or right of the +). For example:

```
sum1 : N + Bool
sum1 = left 3

sum2 : N + Bool
sum2 = right false

sum3 : N + N + N
sum3 = right (right 3)
```

`sum1` and `sum2` have the same type, namely, `N + Bool`; values of this type consist of either a natural number or a boolean. `sum1` contains a natural number, tagged with `left`; `sum2` contains a boolean tagged with `right`.

Notice that `X + X` is a different type than `X`, because we get two distinct copies of all the values in `X`, some tagged with `left` and some with `right`. This is why we call a sum type a *disjoint* union.

Iterated sum types, as in `sum3`, are handled in exactly the same way as iterated product types: `N + N + N` is really syntax sugar for `N + (N + N)`. `sum3` therefore begins with a `right` tag, to show that it contains a value of the right-hand type, namely, `N + N`; this value in turn consists of another `right` tag along with a value of type `N`. Other values of the same type `N + N + N` include `right (left 6)` and `left 5`.

1.6.3 Unit and Void types

Disco has two other special built-in types which are rarely useful on their own, but often play an important role in describing other types.

- The type `Unit` has just a single value, called `()`.

  ```
  Disco> :type ()
  () : Unit
  ```

  It is isomorphic to the type `Z1`.

- The type `Void` has *no* values. It is also isomorphic to `Z0`.

1.6.4 Counting and enumerating types

For any type which has only a finite number of values, disco can count how many values there are, using the `count` operator, or list them using `enumerate` (we will learn more about lists later in the tutorial).

```
Disco> count ((Z2 * Z4) + Bool)
right 10
Disco> enumerate ((Z2 * Z4) + Bool)
```

(continues on next page)

(continued from previous page)

```
[left (0, 0), left (0, 1), left (0, 2), left (0, 3),
 left (1, 0), left (1, 1), left (1, 2), left (1, 3),
 right false, right true]
Disco> enumerate (Bool * Bool * Bool)
[(false, false, false), (false, false, true), (false, true, false), (false, true,␣
↪true),
 (true, false, false), (true, false, true), (true, true, false), (true, true, true)]
```

1.7 Functions

The type of functions with input `X` and output `Y` is written `X -> Y`. Some basic examples of function definitions are shown below.

Listing 4: example/function.disco

```
f : N -> N
f(x) = x + 7

g : Z -> Bool
g(n) = (n - 3) > 7

factorial : N -> N
factorial(0) = 1
factorial(n) = n * factorial(n .- 1)
```

- The function `f` takes a natural number as input, and returns the natural number which is 7 greater. Notice that `f` is defined using the syntax `f(x) = ...`. In fact, the basic syntax for function arguments is juxtaposition, just as in Haskell; the syntax `f x = ...` would work as well. Stylistically, however, `f(x) = ...` is to be preferred, since it matches standard mathematical notation.
- The function `g` takes an integer `n` as input, and returns a boolean indicating whether `n - 3` is greater than 7. Note that this function cannot be given the type `N -> Bool`, since it uses subtraction.
- The recursive function `factorial` computes the factorial of its input. Top-level functions such as `factorial` are allowed to be recursive. Notice also that `factorial` is defined by two cases, which are matched in order from top to bottom, just as in Haskell.

Functions can be given inputs using the same syntax:

```
Disco> f(2^5)
39
Disco> g(-5)
false
Disco> factorial(5 + 6)
39916800
```

"Multi-argument functions" can be written as functions which take a product type as input. (This is again a stylistic choice: disco certainly supports curried functions as well. But in either case, disco fundamentally supports only one-argument functions.) For example:

Listing 5: example/multi-arg-functions.disco

```
gcd : N * N -> N
gcd(a,0) = a
```

(continues on next page)

(continued from previous page)

```
gcd(a,b) = gcd(b, a mod b)

discrim : Q * Q * Q -> Q
discrim(a,b,c) = b^2 - 4*a*c

manhattan : (Q*Q) * (Q*Q) -> Q
manhattan ((x1,y1), (x2,y2)) = abs (x1-x2) + abs (y1-y2)
```

All of these examples are in fact *pattern-matching* on their arguments, although this is most noticeable with the last example, which decomposes its input into a pair of pairs and gives a name to each component.

Functions in disco are first-class, and can be provided as input to another function or output from a function, stored in data structures, *etc*. For example, here is how one could write a higher-order function to take a function on natural numbers and produce a new function which iterates the original function three times:

Listing 6: example/higher-order.disco

```
thrice : (N -> N) -> (N -> N)
thrice(f)(n) = f(f(f(n)))
```

1.7.1 Anonymous functions

The syntax for an anonymous function in disco consists of three parts: one or more *bindings*, followed by a *mapsto* symbol, followed by an arbitrary disco expression.

- Each *binding* specifies the name of an input to the function. A binding can be either a simple variable name, or a parenthesized variable name with a type annotation (*e.g.* `(x:Nat)`). There can be multiple bindings separated by whitespace, which creates a (curried) "multi-argument" function.
- disco will accept any one of several syntaxes for the *mapsto* symbol: either `->`, `|->`, or .

Note: It's quite possible this syntax might change. For example, we might want to disallow `->` as a mapsto symbol, since that may cause confusion with the same symbol used as part of a type. Also, we might want to require "lambda" syntax before the binding (*e.g.* either a backslash or an actual lambda).

The current syntax was designed to mirror the syntax in most common mathematical practice (*e.g.* $x \mapsto x^2 + 3$), but it's quite possible discrete math students will not be familiar with that notation anyway, in which case we might as well introduce them to the lambda calculus.

Currently, bindings cannot contain patterns, but in general we might want to allow this, for example, `((x,y) |-> x + y) : N*N -> N`.

Here are a few examples of using anonymous functions as arguments to `thrice`:

```
Disco> thrice(x |-> x*2)(1)
8
Disco> thrice((z:Nat)  z^2 + 2z + 1)(7)
17859076
```

TODO example of using multi-argument anonymous function

1.7.2 Comparing functions

In certain cases, functions can be compared for equality, or even compared to see which is less or greater.

```
Disco> ((x:Bool) -> x) = ((x:Bool) -> not (not x))
true
Disco> ((x:Bool) -> x) = ((x:Bool) -> not x)
false
```

There is no magic involved, and it does not work by looking at the definitions of the functions. Simply put, two functions are equal if they give the same output for every input. So disco can only test two functions for equality if they have a finite input type, in which case it simply enumerates all possible values of the input type, and tests that the two functions give equal outputs for every input.

Functions are ordered by conceptually listing all their outputs ordered by inputs (that is, list the values of the input type in order from smallest to largest and apply the function to each) and then comparing these lists of outputs lexicographically. That is, if `i` is the smallest possible input value and `f i < g i`, then `f < g`. If `f i = g i`, then we move on to consider the second smallest input value, and so on.

1.7.3 Disambiguating function application and multiplication

As previously mentioned, the fundamental syntax for applying a function to an argument is *juxtaposition*, that is, simply putting the function next to its argument (with a space in between if necessary).

However, disco also allows multiplication to be written in this way. How can it tell the difference? Given an expression of the form `X Y` (where `X` and `Y` may themselves be complex expressions), disco uses simple *syntactic* rules to distinguish between multiplication and function application. In particular, note that the *types* of `X` and `Y` do not enter into it at all (it would greatly complicate matters if parsing and typechecking had to be interleaved—even though this is what human mathematicians do in their heads; see the discussion below).

To decide whether `X Y` is function application or multiplication, disco looks only at the syntax of `X`; `X Y` is multiplication if and only if `X` is a *multiplicative term*, and function application otherwise. A multiplicative term is one that looks like either a natural number literal, or a unary or binary operation (possibly in parentheses). For example, `3`, `(-2)`, and `(x + 5)` are all multiplicative terms, so `3x`, `(-2)x`, and `(x + 5)x` all get parsed as multiplication. On the other hand, an expression like `(x y)` is always parsed as function application, even if x and y both turn out to have numeric types types; a bare variable like x does not count as a multiplicative term. Likewise, `(x y) z` is parsed as function application, since `(x y)` is not a multiplicative term.

Note: You may enjoy reflecting on how a *human* mathematician does this disambiguation. In fact, they are doing something much more sophisticated than disco, implicitly using information about types and social conventions regarding variable names in addition to syntactic cues. For example, consider $x(y+3)$ versus $f(y+3)$. Most mathematicians would unconsciously interpret the first as multiplication and the second as function application, due to standard conventions about the use of variable names x and f. On the other hand, in the sentence "Let x be the function which doubles an integer, and consider $v = x(y+3)$", any mathematician would have no trouble identifying this use of $x(y+3)$ as function application, although they might also rightly complain that x is a strange choice for the name of a function.

1.8 Case expressions

Fundamentally, the only construct available in disco which allows choosing between multiple alternatives is case analysis using a *case expression*. (The other is multi-clause functions defined via pattern-matching, but in fact that is really only syntax sugar for a case expression.)

The syntax of case expressions is inspired by mathematical notation such as

$$f(x) = \begin{cases} x + 2 & x < 0 \\ x^2 - 3x + 2 & 0 \leq x < 10 \\ 5 - x & \text{otherwise} \end{cases}$$

Here is how one would write a corresponding definition in disco:

Listing 7: example/case.disco

```
f : Z -> Z
f(x) = {? x + 2           if x < 0,
          x^2 - 3x + 2    if 0 <= x < 10,
          5 - x           otherwise
       ?}
```

The entire expression is surrounded by `{? ... ?}`; the curly braces are reminiscent of the big brace following $f(x) = \ldots$ in the standard mathematical notation, but we don't want to use plain curly braces (since those will be used for sets), so question marks are added (which remind us that case expressions are really all about asking questions).

1.8.1 Case syntax and semantics

More formally, the syntax of a case expression consists of one or more *branches*, separated by commas, enclosed in `{? ... ?}`. (Whitespace, indentation, *etc.* formally does not matter, though something like the style shown in the example above is encouraged.)

Each *branch* consists of an arbitrary expression followed by zero or more *guards*. When a case expression is evaluated, each branch is tried in turn; the first branch which has *all* its guards succeed is chosen, and the value of its expression becomes the value of the entire case expression. In the example above, this means that first `x < 0` is evaluated; if it is true then `x + 2` is chosen as the value of the entire case expression (and the rest of the branches are ignored). Otherwise, `0 <= x < 10` is evaluated; and so on.

Every *guard* starts with the word `if` or `when` (the two words are interchangeable). There are three types of guards:

- A *boolean guard* is simply an expression of type `Bool`. It succeeds if the expression evaluates to `true`.
- A *pattern guard* has the form `<expr> is <pattern>`. It succeeds if the expression `<expr>` matches the pattern `<pattern>`.
- The special guard `otherwise` always succeeds.

Here is an example using both boolean and pattern guards:

Listing 8: example/case-pattern.disco

```
g : Z*Z -> Z
g(p) = {? 0       when p is (3,_),
          x + y   when p is (x,y) if x > 5 or y > 20,
          -100    otherwise
       ?}
```

Here is the result of evaluating `g` on a few example inputs:

```
Disco> g(3,9)
0
Disco> g(4,3)
-100
```

(continues on next page)

(continued from previous page)

```
Disco> g(16,15)
31
```

When a pattern containing variables matches, the variables are bound to the corresponding values, and are in scope in both the branch expression as well as any subsequent guards. In the example above, when the pattern `(x,y)` matches `p`, both `x` and `y` may be used in the branch expression (`x + y` in this case) as well as in the second guard `if x > 5 or y > 20`. That is, the guards in this branch will only succeed if `p` is of the form `(x,y)` and either `x > 5` or `y > 20`, in which case the value of the whole case expression becomes the value of `x + y`; for example, `g(16,15) = 31`.

Warning: Be careful not to get a Boolean guard using `=` confused with a pattern guard using `is`. (This is probably something that will confuse students learning the language; ideas on how to make it less confusing are welcome. As I am writing this, I realize that it might be a good idea to require `when` with pattern guards and `if` with boolean guards, rather than allowing them to be mixed and matched.) The difference is in how variables are handled: boolean guards can only use existing variables; pattern guards create new variables. For example, `... when p is (x,y)` matches a tuple `p` and gives the names `x` and `y` to the components. On the other hand, `... if p = (x,y)` will probably complain that `x` and `y` are undefined—unless `x` and `y` are already defined elsewhere, in which case this will simply check that `p` is exactly equal to the value `(x,y)`. Use a boolean guard when you want to check some condition; use a pattern guard when you want to take a value apart or see what it looks like.

1.8.2 Function pattern-matching

As we have already seen, functions can be defined via multiple clauses and pattern-matching. In fact, any such definition simply desugars to one big case expression. For example, the `gcd` function shown below actually desugars to something like `gcd2`:

Listing 9: example/function-desugar.disco

```
gcd : N * N -> N
gcd(a,0) = a
gcd(a,b) = gcd(b, a mod b)

gcd2 : N * N -> N
gcd2 = p  {? a                 when p is (a,0),
             gcd2(b, a mod b) when p is (a,b)
          ?}
```

1.9 Lists

Disco defines a type of inductive, singly-linked lists, very similar to lists in Haskell.

1.9.1 Basic lists

All the elements of a list must be the same type, and the type of a list with elements of type `T` is written `List T`. Since it is unambiguous, nested list types can be written without parentheses, *e.g.* `List List List T`.

The basic syntax for constructing and pattern-matching on lists is almost exactly the same as in Haskell, with the one difference that the single colon (type of) and double colon (cons) have been switched from Haskell.

Listing 10: example/list.disco

```
emptyList : List Bool
emptyList = []

nums : List N
nums = [1, 3, 4, 6]

nums2 : List N
nums2 = 1 :: 3 :: 4 :: 6 :: []

  -- nums and nums2 are equal

nested : List List Q
nested = [1, 5/2, -8] :: [[2, 4], [], [1/2]]

sum : List N -> N
sum []        = 0
sum (n :: ns) = n + sum ns
```

1.9.2 List comprehensions

Disco has list comprehensions which are also similar to Haskell's. A list comprehension is enclosed in square brackets, and consists of an expression, followed by a vertical bar, followed by zero or more *qualifiers*, `[ <expr> | <qual>* ]`.

A *qualifier* is one of:

- A *binding* qualifier of the form `x in <expr>`, where `x` is a variable and `<expr>` is any expression with a list type. `x` will take on each of the items of the list in turn.
- A *guard* qualifier, which is an expression with a boolean type. It acts to filter out any bindings which cause the expression to evaluate to `false`.

For example, `comp1` below is a (rather contrived) function on two lists which results in all possible sums of two *even* numbers taken from the lists which add to at least 50. `pythagTriples` is a list of all Pythagoren triples with all three components at most 100. (There are much more efficient ways to compute Pythagorean triples, but never mind.)

Listing 11: example/comprehension.disco

```
comp1 : List N -> List N -> List N
comp1 xs ys = [ x + y | x in xs, 2 divides x, y in ys, 2 divides y, x + y >= 50 ]

pythagTriples : List (N*N*N)
pythagTriples = [ (a,b,c)
  | a in [1 .. 100]
  , b in [1 .. 100]
  , c in [1 .. 100]
  , a^2 + b^2 = c^2
  ]
```

Note: The biggest difference between list comprehensions in disco and Haskell is that Haskell allows *pattern* bindings, *e.g.* `Just x <- xs`, which keep only elements from the list which match the pattern. At the moment, disco only allows variables on the left-hand side of a binding qualifier. There is no reason in principle disco can't support binding qualifiers with patterns, it just isn't a big priority and hasn't been implemented yet.

1.9.3 Polynomial sequences

Like Haskell, disco supports ellipsis notation in literal lists to denote omitted elements, although there are a few notable differences. One minor syntactic difference is that (just for fun) disco accepts two *or more* dots as an ellipsis; the number of dots makes no difference.

Listing 12: example/basic-ellipsis.disco

```
-- Counting numbers from 1 to 100
counting : List N
counting = [1 .. 100]

-- Even numbers from 2 to 100
evens : List N
evens = [2, 4 ..... 100]

-- [5, 4, 3, ... -3, -4, -5]
down : List Z
down = [5 .. -5]

-- 1 + 3 + 5 + 7 = 16
s : N
s = {? a+b+c+d  when [1, 3 ..] is (a::b::c::d::_) ?}

-- It doesn't always have to be integers
qs : List Q
qs = [2/3, 7/5 .. 10]
```

- `[a ..]` denotes the infinite list beginning with `a` and counting up by ones.
- `[a .. b]` denotes the list that starts with `a` and either counts up or down by ones (depending on whether `b` is greater than or less than `a`, respectively), continuing as long as the elements do not "exceed" `b` (the meaning of "exceed" depends on whether the counting is going up or down).
- `[a, b .. c]` denotes the list whose first element is `a`, second element is `b`, and the difference between each element and the next is the difference `b - a`. The list continues as long as the elements do not "exceed" `c`, where "exceed" means either "greater than" or "less than", depending on whether `b - a` is positive or negative, respectively.
- `[a, b ..]` is similar but infinite.

All the above is similar to Haskell, except that `[10 .. 1]` is the empty list in Haskell, and disco's rules about determining when the list stops are much less strange (the strangeness of Haskell's rules is occasioned by floating-point error, which of course disco does not have to deal with).

However, disco also generalizes things further by allowing notation of the form `[a, b, c ..]` or `[a, b, c, d ..]`, and so on. We have already seen that two values `[a, b ..]` generate a linear progression of values; by analogy, three values generate a quadratic progression, four values a cubic, and so on. In general, when n values $a_0, a_1, \ldots, a_n$ are given before an ellipsis, disco finds the unique polynomial p of degree $n-1$ such that $p(i) = a_i$, and uses it to generate additional terms of the list. (In practice, the disco interpreter does not actually find a polynomial, but uses the *method of finite differences*, just like Charles Babbage's Difference Engine.)

Listing 13: example/general-ellipsis.disco

```
-- The infinite list of triangular numbers
triangular : List N
triangular = [1, 3, 6 ..]
```

(continues on next page)

(continued from previous page)

```
-- The infinite list of squares
squares : List N
squares = [1, 4, 9 ..]

-- Some cubes
cubes : List N
cubes = [1, 8, 27, 64 .. 1000]
```

When an ending value is specified, list elements are again included until the first one which "exceeds" the ending value. The precise definition of "exceeds" is a bit trickier to state in general, but corresponds to the eventual behavior of the polynomial: the list stops as soon as elements become either larger than or smaller than the ending value, as the polynomial diverges to $+\infty$ or $-\infty$, respectively.

1.9.4 Multinomial coefficients

We already saw that the `choose` operator can be used to compute binomial coefficients. In fact, if the second operand to `choose` is a list instead of a natural number, it can be used to compute general multinomial coefficients as well. `n choose xs` is the number of ways to choose a sequence of sets whose sizes are given by the elements of `xs` from among a set of `n` items. If the sum of `xs` is equal to `n`, then this is given by `n!` divided by the product of the factorials of `xs`; if the sum of `xs` is greater than `n`, then `n choose xs` is zero; if the sum is less than `n`, it is as if another element were added to `xs` to make up the sum (representing the set of elements which are "not chosen"). In general, `n choose k = n choose [k,n-k] = n choose [k]`.

1.10 Polymorphism

Disco also supports polymorphic functions and let expressions. This is best demonstrated by an example.

1.10.1 Generalizing using Polymorphism

Consider the task of incrementing each number in a list of natural numbers. In order to accomplish this task using Disco, you could simply write the following function:

```
incr : List N -> List N
incr [] = []
incr (a :: as) = (a + 1) :: (incr as)
```

```
Disco> incr [1,2,3]
[2,3,4]
```

Now consider the task of producing a list of singleton lists containing each element from a provided list of Integers. The following function would suffice:

```
single : List Z -> List (List Z)
single [] = []
single (a :: as) = [a] :: (single as)
```

```
Disco> single [-1,2,-3]
[[-1],[2],[-3]]
```

Do you see the similarities between the two functions? Both functions take a list, apply a function to each element in the list, concatenate the results into a new list, and return this new list. This pattern emerges quite often, and it would be tedious to have to write this same structure twice for two functions as similar as `incr` and `single`. Instead, since generalization is the bread and butter of programming, we can capture the essence of the above pattern with the following polymorphic function:

```
map : (a -> b) -> List a -> List b
map _ [] = []
map f (a :: as) = f a :: (map as)
```

The function `map` is a polymorphic, higher-order function which takes as input a function, `f`, which transforms values of type `a` to values of type `b`, a list of elements of type `a`, and returns a list of elements of type `b` by applying `f` to each element in the input list. Note that map will work for any types, `a` and `b`, so we call `a` and `b` "type variables". Therefore, you might read the type of `map` as:

```
-- map : for all types a and b, (a -> b) -> List a -> List b
```

Now we can rewrite the functions `incr` and `single` as follows:

```
incr' : List N -> List N
incr' l = let f = (x : N) -> x + 1 in map f l
```

```
single' : List Z -> List (List Z)
single' l = let f = (x : Z) -> [x] in map f l
```

It's a good idea to try these functions out and verify that they produce the same outputs as their non-generic counterparts.

Other common polymorphic, higher-order functions include `foldr` and `filter`:

```
filter : (a -> Bool) -> List a -> List a
filter _ [] = []
filter f (a :: as) = {? a :: (filter f as)    if f a,
                                  filter f as           otherwise
                               ?}
```

```
foldr : (a -> b -> b) -> b -> List a -> b
foldr _ acc [] = acc
foldr f acc (a :: as) = foldr f (f a acc) as
```

If you are unfamiliar with these functions, play around with them and see if you can figure out what they do/what pattern they capture! This will help you understand their types.

Note that the following following function would not typecheck in Disco:

```
invalid : List a -> List a
invalid [] = []
invalid (a : as) = (a + 1) :: (invalid as)
```

This is because our type definition states that `invalid` should work lists of all types (due to the type variable `a`), however, our definition implies that `invalid` only works on lists of a numeric type.

1.10.2 Limitations

1.11 Laziness

Disco takes a hybrid approach to laziness.

- To facilitate compositionality and allow for things such as infinite lists, structured types (lists, pairs, etc.) in disco are lazy.
- To avoid surprising performance issues, numeric types in disco are strict.

Examples coming soon.

1.11.1 Let expressions

Let expressions are a mechanism for defining new variables for local use within an expression. For example, `3 + (let y = 2 in y + y)` evaluates to `7`: the expression `y + y` is evaluated in a context where `y` is defined to be `2`, and the result is then added to `3`. The simplest syntax for a let expression, as in this example, is `let <variable> = <expression1> in <expression2>`. The value of the let expression is the value of `<expression2>`, which may contain occurrences of the `<variable>`; any such occurrences will take on the value of `<expression1>`.

More generally:

- A `let` may have multiple variables defined before `in`, separated by commas.
- Each variable may optionally have a type annotation.
- The definitions of later variables may refer to previously defined variables.
- However, the definition of a variable in a `let` may not refer to itself; only top-level definitions may be recursive.

Here is a (somewhat contrived) example which demonstrates all these features:

Listing 14: example/let.disco

```
f : Nat -> List Nat
f n =
  let x : Nat = n//2,
      y : Nat = x + 3,
      z : List Nat = [3,x,y]
  in  n :: z
```

An important thing to note is that a given definition in a `let` expression will only ever be evaluated (at most) once, even if the variable is used multiple times. `let` expressions are thus a way for the programmer to ensure that the result of some computation is shared. `let x = e in f x x` and `f e e` will always yield the same result, but the former might be more efficient, if `e` is expensive to calculate.

1.12 Properties

Each disco definition may have any number of associated *properties*, mathematical claims about the behavior of the definition which can be automatically verified by disco. Properties begin with `!!!`, must occur just before their associated definition, and may be arbitrarily interleaved with documentation lines beginning with `|||`.

1.12.1 Unit tests

The simplest kind of property is just an expression of type `Bool`, which essentially functions as a unit test. When loading a file, disco will check that all such properties evaluate to `true`, and present an error message if any do not.

Listing 15: example/unit-test.disco

```
!!! gcd(7,6)   = 1
!!! gcd(12,18) = 6
!!! gcd(0,0)   = 0

gcd : N * N -> N
gcd(a,0) = a
gcd(a,b) = gcd(b, a mod b)
```

When we load this file, disco reports that it successfully ran the tests associated with `gcd`:

```
Disco> :load example/unit-test.disco
Loading example/unit-test.disco...
Running tests...
  gcd: OK
Loaded.
```

On the other hand, if we change the first property to `!!! gcd(7,6) = 2` and load the file again, we get an error:

```
Disco> :load example/unit-test.disco
Loading example/unit-test.disco...
Running tests...
  gcd:
  - Test result mismatch for: gcd (7, 6) = 2
    - Expected: 2
    - But got:  1
Loaded.
```

1.12.2 Quantified properties

More generally, properties can contain universally quantified variables. The syntax for a universally quantified property is as follows:

- the word `forall` (or the Unicode symbol ∀);
- one or more comma-separated *bindings*, each consisting of a variable name, a colon, and a type;
- a period;
- and an arbitrary expression, which should have type `Bool` and which may refer to the variables bound by the `forall`.

Such quantified properties have the obvious logical interpretation: they hold only if the given expression evaluates to `true` for all possible values of the quantified variables.

Listing 16: example/property.disco

```
!!!  x:B. neg (neg x) = x
neg : B -> B
neg x = not x
```

(continues on next page)

(continued from previous page)

```
!!! p: N + N. plusIsoR (plusIso p) = p
plusIso : N + N -> N
plusIso (left n) = 2n
plusIso (right n) = 2n + 1

!!! n:N. plusIso (plusIsoR n) = n
plusIsoR : N -> N + N
plusIsoR n =
  {? left  (n // 2)   if 2 divides n
   , right (n // 2)   otherwise
  ?}

!!! forall x:N, y:N, z:N.
      f(f(x,y), z) = f(x, f(y,z))

f : N*N -> N
f (x,y) = x + x*y + y
```

In the example above, the first three properties have a single quantified variable, and specify respectively that `neg` is self-inverse, and `plusIso` and `plusIsoR` are inverse. The last function has a property with multiple quantified variables, and specifies that `f` is associative. Notice that as in this last example, properties may extend onto multiple lines, as long as subsequent lines are indented. Only a single `!!!` should be used at the start of each property.

Such properties may be undecidable in general, so disco cannot automatically *prove* them. Instead, it searches for counterexamples. If the input space is finite and sufficiently small (as in the first example above, which quantifies over a single boolean), disco will enumerate all possible inputs and check each one; so in this special case, disco can actually prove the property by exhaustively checking all cases. Otherwise, disco randomly generates a certain number of inputs (*a la* `QuickCheck`) and checks that the property is satisfied for each. If a counterexample is found, the property certainly does not hold, and the counterexample can be printed. If no counterexample is found, the property "probably" holds.

For example, consider this function with a property claiming it is associative:

Listing 17: example/failing/property.disco

```
!!! forall x:N, y:N, z:N.
      f(f(x,y), z) = f(x, f(y,z))

f : N*N -> N
f (x,y) = x + 2*y
```

The function is not associative, however, and if we try to load this file disco quickly finds a counterexample:

```
Disco> :load example/failing/property.disco
Loading example/failing/property.disco...
Running tests...
  f:
  - Test result mismatch for:  x : , y : , z : . f (f (x, y), z) = f (x, f (y, z))
    - Expected: 5
    - But got:  3
    Counterexample:
      x = 1
      y = 0
      z = 1
Loaded.
```

CHAPTER 2

Disco Language Reference

Coming soon!