DISCO: A Functional Programming Language for Discrete Mathematics

Brent A. Yorgey
Hendrix College
Conway, Arkansas, USA
yorgey@hendrix.edu

DISCO is a pure, strict, statically typed functional programming language designed to be used in the setting of a discrete mathematics course. The goals of the language are to introduce students to functional programming concepts early, and to enhance their learning of mathematics by providing a computational platform for them to play with. It features mathematically-inspired notation, property-based testing, equirecursive algebraic types, subtyping, built-in list, bag, and finite set types, a REPL, and student-focused documentation. DISCO is implemented in Haskell, with source code available on GitHub,[1] and interactive web-based REPL available through replit.[2]

1 Introduction

Many computer science curricula at the university level include *discrete mathematics* as a core requirement [CM13]. Often taken in the first or second year, a discrete mathematics course introduces mathematical structures and techniques of foundational importance in computer science, such as induction and recursion, set theory, logic, modular arithmetic, functions, relations, and graphs. In addition, it sometimes serves as an introduction to writing formal proofs. Although there is wide agreement that discrete mathematics is foundational, students often struggle to see its relevance to computer science.

Functional programming is a style of programming, embodied in languages such as Haskell, OCaml, Scala, F#, and Racket, which emphasizes functions (*i.e.* input-output processes) rather than sequences of instructions. It enables working at high levels of abstraction as well as rapid prototyping and refactoring, and provides a concise and powerful vocabulary to talk about many other topics in computer science. It is becoming critical to expose undergraduate students to functional programming early, but many computer science programs struggle to make space for it. The Association for Computing Machinery's 2013 curricular guidelines [CM13] do not even include functional programming as a core topic.

One creative idea is to combine functional programming and discrete mathematics into a single course. This is not a new idea [Wai92, Hen02, SW02, DE04, OHP06, Van11, Xin08], and even shows up in the 2007 model curriculum of the Liberal Arts Computer Science Consortium [Lib07]. The benefits of such an approach are numerous:

- It allows functional programming to be introduced at an early point in undergraduates' careers, since discrete mathematics is typically taken in the first or second year. This allows ideas from functional programming to inform students' thinking about the rest of the curriculum. By contrast, when functional programming is left until later in the course of study, it is in danger of being seen as esoteric or as a mere curiosity.

[1] https://github.com/disco-lang/disco
[2] https://replit.com/@BrentYorgey/Disco#README.md

Submitted to:
TFPIE 2023

- The two subjects complement each other well: discrete math topics make good functional programming exercises, and ideas from functional programming help illuminate discrete math topics.
- In a discrete mathematics course with both math and computer science majors, math majors can have a “home turf advantage” since the course deals with topics that may be already familiar to them (such as writing proofs), whereas computer science majors may struggle to connect the course content to computer science skills and concepts they already know. Including functional programming levels the playing field, giving both groups of students a way to connect the course content to their previous experience. Computer science majors will be more comfortable learning math concepts that they can play with computationally; math majors can leverage their math experience to learn a bit about programming.
- It is just plain fun: using programming enables interactive exploration of mathematics concepts, which leads to higher engagement and increased retention.

However, despite its benefits, this model is not widespread in practice. This may be due partly to lack of awareness, but there are also some real roadblocks to adoption that make it impractical or impossible for many departments.

- Existing functional languages—such as Haskell, Racket, OCaml, or SML—are general-purpose languages which (with the notable exception of Racket) were not designed specifically with teaching in mind. The majority of their features are not needed in the setting of discrete mathematics, and teachers must waste a lot of time and energy explaining incidental detail or trying to hide it from students.
- Again with the notable exception of Racket, tooling for existing functional languages is designed for professional programmers, not for students. The systems can be difficult to set up, generate confusing error messages, and are generally designed to facilitate efficient production of code rather than interactive exploration and learning.
- As with any subject, effective teaching of a functional language requires expertise in the language and its use, or at least thorough familiarity, on the part of the instructor. General-purpose functional languages are large, complex systems, requiring deep study and years of experience to master. Even if only a small part of the language is presented to students, a high level of expertise is still required to be able to select and present a relevant subset of the language and to help students navigate around the features they do not need. For many instructors, spending years learning a general-purpose functional language just to teach discrete mathematics is a non-starter. This is especially a problem at institutions where the discrete mathematics course is taught by mathematics faculty rather than computer science faculty.
- There is often an impedance mismatch between standard mathematics notation and the notation used by existing functional programming languages. As one simple example, in mathematics one can write $2x$ to denote multiplication of x by 2; but many programming languages require writing a multiplication operator, for example, `2*x`. Any one such impedance mismatch is small, but the accumulation of many such mismatches can be a real impediment to students as they attempt to move back and forth between the worlds of abstract mathematics and concrete computer programs.

DISCO is a new functional programming language, specifically designed for use in a discrete mathematics course, which attempts to solve many of these issues:

- Although DISCO is Turing-complete, it is a teaching language, not a general-purpose language. It includes only features which are of direct relevance to teaching core functional programming and discrete mathematics topics; for example, it does not include a floating-point number type. Section 2 has many examples of the language's features and some discussion of features which are explicitly excluded.

- As much as possible, the language's features and syntax mirror common mathematical practice rather than other functional languages. Section 2 has many examples, and Section 3.1 discusses some notable exceptions.
- As a result—although there is as yet no data to back this up—the language should be easy for instructors to learn, even mathematicians without much prior programming experience.

DISCO is an open-source project, implemented in Haskell, with source code licensed under a BSD 3-clause license and available on GitHub.[3] Although it is possible to install DISCO locally, either from Hackage[4] or directly from source, one can also interact with DISCO in the cloud via a web browser, through the magic of replit.[5] This is the primary way that students will be instructed to use Disco, so that students do not need to install a Haskell toolchain or worry about exhausting the computational resources of their device. Via replit, it is entirely feasible to play with DISCO on any device with a web browser, including Chromebooks, tablets, or phones. Documentation for DISCO is hosted on `readthedocs.org`.[6]

2 DISCO by Example

We will begin by exploring some of the major features and uses of the language via a series of examples.

2.1 Greatest common divisor

Our first example is an implementation of the classic Euclidean algorithm for computing the greatest common divisor of two natural numbers, shown in Listing 1.

```
||| The greatest common divisor of two natural numbers.

!!! gcd(7,6)   == 1
!!! gcd(12,18) == 6
!!! gcd(0,0)   == 0
!!! forall a:N, b:N. gcd(a,b) divides a /\ gcd(a,b) divides b
!!! forall a:N, b:N, g:N. (g divides a /\ g divides b) ==> g divides gcd(a,b)

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

Listing 1: Definition of `gcd` in DISCO

Lines beginning with `|||` denote special documentation comments attached to the subsequent definition, similar to docstrings in Python (regular comments start with `--`). This documentation can be later accessed with the `:doc` command at the REPL prompt:

[3] `https://github.com/disco-lang/disco`
[4] `https://hackage.haskell.org/package/disco`
[5] `https://replit.com/@BrentYorgey/Disco#README.md`
[6] `https://disco-lang.readthedocs.io`

```
Disco> :doc gcd
gcd : ℕ × ℕ → ℕ

The greatest common divisor of two natural numbers.
```

Lines beginning with `!!!` denote *tests* attached to the subsequent definition, which can be either simple Boolean unit tests (such as `gcd(7,6) == 1`), or quantified properties (such as the last two tests, which together express the universal property defining `gcd`). Such properties will be tested exhaustively when feasible, or, when exhaustive testing is impossible (as in this case), tested with a finite number of randomly chosen inputs. Under the hood, this uses the `QuickCheck` [CH00] and `simple-enumeration` packages to generate inputs. For example:

```
Disco> :test forall a:N, b:N. let g = gcd(a,b) in g divides a /\ g divides b
  - Possibly true: ∀a, b. let g = gcd(a, b) in g divides a /\ g divides b
    Checked 100 possibilities without finding a counterexample.

Disco> :test forall a:N, b:N. let g = gcd(a,b) in g divides a /\ (2g) divides b
  - Certainly false: ∀a, b. let g = gcd(a, b) in g divides a /\ 2 * g divides b
    Counterexample:
      a = 0
      b = 1
```

In the first case, DISCO reports that 100 sample inputs were checked without finding a counterexample, leading to the conclusion that the property is *possibly* true. In the second case, when we modify the test by demanding that `b` must be divisible by twice `gcd(a,b)`, DISCO is quickly able to find a counterexample, proving that the property is *certainly* false.

Every top-level definition in DISCO must have a type signature; `gcd : N * N -> N` indicates that `gcd` is a function which takes a pair of natural numbers as input and produces a natural number result. The recursive definition of `gcd` is then straightforward, featuring multiple clauses and pattern-matching on the input.

2.2 Primality testing

The example shown in Listing 2, testing natural numbers for primality via trial division, is taken from Doets and van Eijck [DE04, pp. 4–11], and has been transcribed from Haskell into DISCO. (DISCO also has a much more efficient built-in primality testing function that calls out to the highly optimized `arithmoi` package.)

There are a few interesting things to point out about this example. The most obvious is the use of a *case expression* in the definition of `ldf` delimited by `{? ... ?}`. It is supposed to be reminiscent of typical mathematical notation like

$$\mathit{ldf}\ k\ n = \begin{cases} k & \text{if } k \mid n, \\ n & \text{if } k^2 > n, \\ \mathit{ldf}\ (k+1)\ n & \text{otherwise.} \end{cases}$$

However, we can't use a bare curly brace as DISCO syntax since it would conflict with the notation for literal sets (and we can't use a giant, multi-line curly brace in any case!). The intention is that writing

```
||| ldf k n calculates the least divisor of n that is at least k and
||| at most sqrt n.  If no such divisor exists, then it returns n.
ldf : N -> N -> N
ldf k n =
  {? k             if k divides n,
     n             if k^2 > n,
     ldf (k+1) n   otherwise
  ?}

||| ld n calculates the least nontrivial divisor of n, or returns n if
||| n has no nontrivial divisors.
ld : N -> N
ld = ldf 2

||| Test whether n is prime or not.
isPrime : N -> Bool
isPrime n = (n > 1) and (ld n == n)
```

Listing 2: Primality testing in DISCO

`{? ... ?}` lends itself to the mnemonic of "asking questions" to see which branch of the case expression to choose. In general, each branch can have multiple chained conditions, each of which can either be a Boolean guard, as in this example, or a pattern match introduced with the `is` keyword. In fact, all multi-clause function definitions with pattern matching really desugar into a single case expression. For example, the definition of `gcd` in Listing 1 desugars to

```
gcd : N * N -> N
gcd = \p. {? a if p is (a,0), gcd(b, a mod b) if p is (a,b) ?}
```

Notice that the definition of `isPrime` uses the `and` keyword instead of `/\`. These are synonymous—in fact, `&&` and ∧ (U+2227 LOGICAL AND) are also accepted. In general, DISCO's philosophy is to allow multiple syntaxes for things with common synonyms rather than imposing one particular choice. Typically a Unicode representation of the "real" math notation is supported (and used when pretty-printing), along with an ASCII equivalent, as well as (when applicable) syntax common in other functional programming languages. Another good example is the natural number type, which can be written ℕ, `N`, `Nat`, or `Natural`. There are several reasons for this design choice:

- It makes code easier to *write* since students have to spend less time trying to remember the one and only correct syntax choice, or worrying about whether a particular syntax they remember comes from math class, Python, or DISCO.
- Although having many different syntax choices can make code harder to *read*, helping students learn how to interpret formal notation and how to translate between mathematics and programming notation are typical explicit learning goals of the course, so this could be considered a feature.

Notice that `ldf` is defined via currying, and is partially applied in the definition of `ld`. Just as in Haskell, every function in DISCO takes exactly one argument, but some functions can return other functions (curried style) and some functions can take a product type as input (uncurried style). Via tutorials, documentation, and the types of standard library functions, DISCO encourages the use of an *uncurried* style, since students are already used to notation like `f(x,y)` for multi-argument functions in mathematics.

Finally, this example introduces the primitive `Bool` type in addition to the natural number type `N` seen previously. DISCO also has a primitive `Char` type for Unicode codepoints, and several other numeric types to be discussed later.

2.3 Z-order

The "Morton Z-order" is one of my favorite bijections showing that $\mathbb{N} \times \mathbb{N}$ has the same cardinality as $\mathbb{N}$; it takes a pair of natural numbers, expresses them in binary, and interleaves their binary representations to form a single natural number. DISCO code to compute this bijection (and check that it really is a bijection) is shown in Listing 3.

```
!!! forall n:N. zOrder(zOrder'(n)) == n
!!! forall p:N*N. zOrder'(zOrder(p)) == p

zOrder : N*N -> N
zOrder(0,0) = 0
zOrder(2m,n) = 2 * zOrder(n,m)
zOrder(2m+1,n) = 2 * zOrder(n,m) + 1

zOrder' : N -> N*N
zOrder'(0)    = (0,0)
zOrder'(2n)   = {? (2y,x) when zOrder'(n) is (x,y) ?}
zOrder'(2n+1) = {? (2y+1,x) when zOrder'(n) is (x,y) ?}
```

Listing 3: Morton Z-Order

This example again uses case expressions; it may seem odd to use case expressions with only one branch, but this is done in order to be able to pattern-match on the result of the recursive call to `zOrder'`. The most interesting thing about this example is its use of *arithmetic patterns*, such as `zOrder'(2n) = ...` and `zOrder'(2n+1) = ...`. This is common mathematical notation, but perhaps less common in programming languages. Any expression with exactly one variable and only basic arithmetic operators can be used as a pattern; the pattern matches if there exists a value for the variable which makes the expression equal to the input. For example, the pattern `2n` will match only even natural numbers, and `n` will then be bound to half of the input.

2.4 Finite sets

DISCO has built-in *finite sets*; in particular, values of type `Set(A)` are finite sets with elements of type `A`. DISCO supports the usual set operations (union, intersection, difference, cardinality, power set), and sets can be created by writing a finite set literal, like `{1,3,5,7}`, using ellipsis notation, like `{1, 3 .. 7}`, or using a set comprehension, as in `{2x+1 | x in {0 .. 3}}`. Listing 4 shows a portion of an exercise (with answers filled in) to help students practice their understanding of set comprehensions.

Set comprehensions in DISCO work similarly to list comprehensions in Haskell (DISCO has list and bag comprehensions as well). In these examples we can see both *filtering* the generated values via Boolean guards, as well as *transforming* the outputs via an expression to the left of the vertical bar.

```
-- Exercise D1. For each of exA through exF below, replace the empty
-- set with a *set comprehension* so that the tests all pass, as in
-- the example.  (Remember, Disco will run the tests when you :load
-- this file.)
--
-- Some relevant documentation you may find useful:
--
-- https://disco-lang.readthedocs.io/en/latest/reference/set.html
-- https://disco-lang.readthedocs.io/en/latest/reference/comprehension.html
-- https://disco-lang.readthedocs.io/en/latest/reference/size.html
-- https://disco-lang.readthedocs.io/en/latest/reference/power.html

||| An example to illustrate the kind of thing you are supposed to do
||| in the exercises below.  We have defined the set using a *set
||| comprehension* so that it has the specified elements and the test
||| passes.

!!! example =!= {1, 4, 9, 16, 36}   -- the test specifying the elements of 'example'
example : Set(N)
example = {x^2 | x in {1 .. 6}, x /= 5}  -- a set comprehension defining it

-- Now you try.

!!! exA =!= {1, 3, 5, 7, 9, 11, 13, 15}
exA : Set(N)
exA = {2x+1 | x in {0..7}}

!!! exD =!= {{1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}}
exD : Set(Set(N))
exD = {S | S in power({1..4}), |S| == 3}
```

Listing 4: Set comprehension exercise

While on the subject of sets, it is worth mentioning that the distinction between *types* and *sets* is something of a pedagogical minefield: the distinction is nonexistent in typical presentations of mathematics, but crucial in a computational system with static type checking. This issue is discussed in more detail in Section 3.4.

One other thing this example highlights is that there is extensive, student-centered documentation available at `https://disco-lang.readthedocs.io/`. Students are pointed to this documentation not just from links in homework assignments such as this, but also by the DISCO REPL itself. Encountering an error, or asking for documentation about a function, type, or operator, are all likely to result in documentation links for further reading, as illustrated in Listing 5.

```
Disco> :doc +
~+~ : ℕ × ℕ → ℕ
precedence level 7, left associative

The sum of two numbers, types, or graphs.

https://disco-lang.readthedocs.io/en/latest/reference/addition.html

Disco> x + 3
Error: there is nothing named x.
https://disco-lang.readthedocs.io/en/latest/reference/unbound.html
```

Listing 5: DISCO generates links to online documentation

2.5 Trees and Catalan numbers

Listing 6 is a fun example generating and counting binary trees. It defines a recursive type `BT` of binary tree shapes, along with a function to generate a list of all possible tree shapes of a given size (via a list comprehension), and uses it to generate the first few Catalan numbers. This list is then extended via lookup in the Online Encyclopedia of Integer Sequences (OEIS) [OEI22].

The first thing to note is that DISCO has *equirecursive* algebraic types. The `type` declaration defines the type `BT` to be *the same type as* `Unit + BT*BT` (*i.e.* the tagged union of the primitive one-element `Unit` type with pairs of `BT` values). This is a big departure from the *isorecursive* types of Haskell and OCaml, where *constructors* are required to explicitly "roll" and "unroll" values of recursive types. We can see in the example that `size` takes a value of type `BT` as input, but can directly pattern-match on `left(unit)` and `right(l,r)` without having to "unfold" or "unroll" it first. Using equirecursive types makes the implementation of the type system more complex, but it is a very deliberate choice:

- There is less incidental complexity for students to stumble over. In my experience, students learning Haskell often get confused over the idea of constructors and how to use them to create and pattern-match on data types.
- DISCO has no special syntax for declaring (recursive) sums-of-products; it simply has sum types, product types, and recursive type synonyms. Of course, it would be very tedious to write "real" programs in such a language—values of large sum types like `type T = A + B + C + ...` have to be written as `left(a)`, `right(left(b))`, `right(right(left(c)))`, and so on. However, the sum types used as examples in a discrete mathematics class rarely have more than two or three

```
import list
import oeis

-- The type of binary tree shapes: empty tree, or a pair of subtrees.
type BT = Unit + BT*BT

||| Compute the size (= number of binary nodes) of a binary tree shape.
size : BT -> N
size(left(unit)) = 0
size(right(l,r)) = 1 + size(l) + size(r)

||| Check whether all the items in a list satisfy a predicate.
all : List(a) * (a -> Bool) -> Bool
all(as, P) = reduce(~/\~, true, each(P, as))

||| Generate the list of all binary tree shapes of a given size.
!!! all([0..4], \n. all(treesOfSize(n), \t. size(t) == n))
treesOfSize : N -> List(BT)
treesOfSize(0)   = [left(unit)]
treesOfSize(n+1) =
  [ right (l,r) | k <- [0 .. n], l <- treesOfSize(k), r <- treesOfSize(n .- k) ]

||| The first few Catalan numbers, computed by brute force.
catalan1 : List(N)
catalan1 = each(\n. length(treesOfSize(n)), [0..4])

||| More Catalan numbers, extended via OEIS lookup!
catalan : List(N)
catalan = extendSequence(catalan1)
```

Listing 6: Counting trees

summands, and working directly with primitive sum and product types helps students explicitly make connections to other things they have already seen, such as Cartesian product and disjoint union of sets. It also reinforces the *algebraic* nature of algebraic data types.

The `oeis` module is inessential, but can be a fun way for students to explore integer sequences and the OEIS. In addition to `extendSequence`, the module also provides a `lookupSequence` function, which returns the URL of the first OEIS result, if there is any:

```
Disco> lookupSequence(catalan1)
right("https://oeis.org/A000108")
```

The last things illustrated by this example are some facilities for computing with collections. The built-in `each` function is like Haskell's `map`, but works for sets and multisets in addition to lists. `reduce` is like `foldr`, but again working over sets and multisets in addition to lists. In this case, the `all` function is defined by first mapping a predicate over `each` element of a list, then reducing the resulting list of booleans via logical conjunction. (Putting twiddles (`~`) in place of arguments is the way to turn operators into standalone functions, thus: `~/\~`.) Notice also that the `all` function is polymorphic: DISCO has support for standard parametric polymorphism.

2.6 Defining and testing bijections

Listing 7 shows part of another exercise I give to my students, asking them to define the inverse of a given function and use DISCO to check that their inverse is correct. This exercise makes essential use of the testing facility we have aleady seen: if a student defines a function which is not inverse to the given function, DISCO is usually able to quickly find a counterexample. Running this counterexample through the functions hopefully gives the student some insight into why their function is not correct. For example, if we try (incorrectly) defining `g2(x) = x - 1/2`, DISCO reports

```
g2:
- Certainly false: ∀x. f2(g2(x)) == x
  Counterexample:
    x = 1
```

In this example we can also see more numeric types besides the natural numbers. DISCO actually has four primitive numeric types:

- The natural numbers $\mathbb{N} = \{0,1,2,\dots\}$, which support addition and multiplication.
- The integers $\mathbb{Z} = \{\dots,-2,-1,0,1,2,\dots\}$, which besides addition and multiplication also support subtraction.
- The *fractional numbers* $\mathbb{F} = \{a/b \mid a,b \in \mathbb{N}, b \neq 0\}$, *i.e.* nonnegative rationals, which besides addition and multiplication also support division.
- The *rational numbers* $\mathbb{Q}$, which support all four arithmetic operations.

DISCO uses *subtyping* to match standard mathematical practice. For example, it is valid to pass a natural number value to a function expecting an integer input. Mathematicians (and students!) would find it very strange and tedious if one were required to apply some sort of coercion function to turn a natural number into an integer.

These four types naturally form a diamond-shaped lattice, as shown in Fig. 1. $\mathbb{N}$ is a subtype of both $\mathbb{Z}$ and $\mathbb{F}$, which are in turn both subtypes of $\mathbb{Q}$. Moving up and left in the lattice (from $\mathbb{N}$ to $\mathbb{Z}$, or $\mathbb{F}$ to $\mathbb{Q}$) corresponds to allowing subtraction; moving up and right corresponds to allowing division. Moving

```
----------------------------------------------------------------------
-- Each of the functions below is a bijection.  Define another Disco
-- function which is its inverse, and write properties showing that
-- the functions are inverse.  Part (a) has already been done for you
-- as an example.  Part (b) has been done partially.  You should
-- complete parts (c)-(g) on your own.

-- (a) ---------------------------------------------------

f1 : Z -> Z
f1(n) = n - 5

-- EXAMPLE SOLUTION for part (a).  Definition of g1 as the inverse of
-- f1, with two test properties demonstrating they are inverse.

!!! forall z:Z. f1(g1(z)) == z
!!! forall z:Z. g1(f1(z)) == z

g1 : Z -> Z
g1(n) = n + 5

-- (b) ---------------------------------------------------

f2 : Q -> Q
f2(x) = 2x + 1

-- PARTIAL SOLUTION for part (b).  Some test properties and a type
-- declaration for g2; you should fill in a definition for g2.

!!! forall x:Q. f2(g2(x)) == x
!!! forall x:Q. g2(f2(x)) == x

g2 : Q -> Q
-- FILL IN YOUR DEFINITION HERE
```

Listing 7: Defining and testing bijections

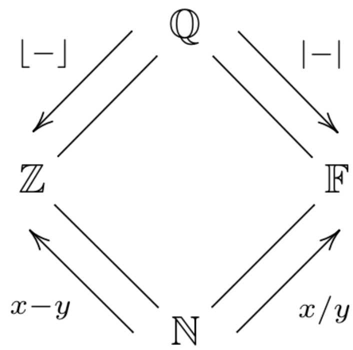

Figure 1: DISCO's numeric type lattice

```
Disco> :type -3
-3 : ℤ
Disco> :type |-3|
abs(-3) : ℕ
Disco> :type 2/3
2 / 3 : 𝔽
Disco> :type -2/3
-2 / 3 : ℚ
Disco> :type floor(-2/3)
floor(-2 / 3) : ℤ
Disco> :type [1,2,3]
[1, 2, 3] : List(ℕ)
Disco> :type [1,-2,3/5]
[1, -2, 3 / 5] : List(ℚ)
```

Listing 8: Numeric types and subtyping

down and left can be accomplished via a rounding operation such as floor or ceiling; moving down and right can be accomplished via absolute value. Listing 8 demonstrates these ideas by requesting the types of various expressions. In the last example, in particular, notice how DISCO infers the type $\mathbb{Q}$ for the elements of the list, since that is the only type that supports both negation and division.

DISCO has no floating-point type, because floating-point numbers are *the worst* [Gol91] and there is no particular need for real numbers in a discrete mathematics course.

3 Discussion and Future Work

3.1 Syntax

For the most part, DISCO tries to use syntax as close to standard mathematical syntax as possible. However, there are a few notable cases where this was deemed impossible, typically because standard mathematical syntax is particularly ambiguous or overloaded. Thinking explicitly about these cases is a worthwhile exercise, since they are likely to be confusing to students anyway.

- Mathematicians are very fond of using vertical bars for multiple unrelated things, and DISCO actually does well to allow them in many cases: absolute value, set cardinality, and the separator between expression and guards in a comprehension all can be written in DISCO with vertical bars. However, the "is a divisor of" relation is also traditionally written with a vertical bar, as in $3 \mid 21$, but DISCO does not support this notation. Including it makes the grammar extremely ambiguous. (And besides, Dijkstra would tell us that we should not use a visually symmetric operator symbol for a nonsymmetric relation!) Instead, DISCO provides `divides` as an infix operator. In my experience students have no problem remembering the difference.
- In mathematics, the equality symbol $=$ is also typically overloaded to denote both definition ("*let* $x = 3$*, and consider...*") and equality testing ("*if* $x = 3$*, then...*"). DISCO cannot use the same symbol for both, since otherwise it would be impossible to tell whether the user is writing a definition or entering a Boolean test to be evaluated. This is confusing for students but it seems like it can't be helped, and in any case I would argue that trying to gloss over the difference is not really doing students any favors, but simply allowing them to persist in some fundamental misunderstandings.
- DISCO allows juxtaposition to denote both function application, as in `f(3)`, and multiplication, as in `2x`. It uses a simple syntax-directed approach to tell them apart: if the expression on the left-hand side of a juxtaposition is a numeric literal, or a parenthesized expression with an operator, then it is interpreted as multiplication; otherwise it is interpreted as function application. However, this does not always get it right, and there are times when an explicit multiplication operator must be written (one notable example is when scaling the output of a function call, as in $2f(x)$; in DISCO this must sadly be written `2*f(x)`). It might be worth exploring a more type-directed approach, although that would be considerably more complex. It seems like to really get this "right" requires general intelligence: for example, does the expression $f(x+2)$ denote multiplication or function application? Are you sure? How do you know? What about in the expression $x(y+2)$? Or how about "Let x be the function which doubles its argument, and consider $x(y+2)$ ..."?

3.2 Student experience

So far, I have used DISCO in my discrete mathematics course once, in the spring of 2022, and I plan to use it again in the spring of 2023. Anecdotally, student experience seems mostly positive, although I do not have any data to back this up. One of my main tasks for the spring of 2023 will be to develop a solid sequence of exercises that can serve as a tutorial for the language, and commit these exercises back into the disco repository itself, to make it easier for other instructors or independent learners to make use of the language. I plan to also start collecting some very informal, qualitative data about students' experience using the language and its effect on their learning.

3.3 Type system

DISCO and its type system were designed to be intuitive for students and to corresponding closely to mathematical practice, but this has not always led to the simplest type system from an implementation point of view!

- As previously mentioned, DISCO has *subtyping* in order to accommodate typical mathematical practice. DISCO's subtyping is *structural*, meaning that we only really need concern ourselves with subtyping relationships between primitive types; a subtyping relation between complex types (for example, sum, product, or function types) can always ultimately be broken down into subtyping relations between simpler types. Subtyping complicates the type system since, for example, when typechecking the application of a function to an argument, we cannot just check that the types match via unification, but we must instead emit a subtyping constraint which we check later.

- DISCO also has parametric polymorphism, since a language without polymorphism would not really give students a good idea of the expressive power of statically typed functional programming. Of course, this means that typing constraints can involve unification variables as well as *skolem variables* (when *checking* a polymorphic type).

- DISCO's type system must actually support *qualified* polymorphism (similar to Haskell's type classes, but with only a specific set of built-in classes) in order to be able to infer types in a setting where some types support certain operations (*e.g.* subtraction or division) and some do not. For example, what is the type of `\x. x - 2`? Most generally, this function has a type like $\forall a.\ (sub(a), \mathbb{Z} <: a) \Rightarrow a \to a$, that is, is a polymorphic function with type $a \to a$ for any type a which supports subtraction and has $\mathbb{Z}$ as a subtype, *i.e.* either $\mathbb{Z}$ or $\mathbb{Q}$. (As a nice exercise, you might like to convince yourself that none of $\mathbb{Z} \to \mathbb{Z}$, $\mathbb{Z} \to \mathbb{Q}$, $\mathbb{Q} \to \mathbb{Z}$, or $\mathbb{Q} \to \mathbb{Q}$ will work—some of them are invalid types for the function, and some of them, although valid, are not general enough.)

  Such types are currently only allowed internally, during type inference, but must be monomorphized away before showing types to users. This is sound, but can be rather confusing. For example, DISCO will report that the type of `\x. x - 2` is $\mathbb{Z} \to \mathbb{Z}$, but will also happily allow it to be applied to a fractional input such as $5/2$, which would be a type error if its most general type were really $\mathbb{Z} \to \mathbb{Z}$.

  ```
  Disco> :type \x. x - 2
  λx. x - 2 : ℤ → ℤ
  Disco> (\x. x - 2) (5/2)
  1/2
  ```

  One interesting idea to improve the situation would be to show the user *multiple* potential monomorphic instantiations of a general type scheme, something like this, perhaps:

  ```
  Disco> :type \x. x - 2
  λx. x - 2
    : ℤ → ℤ
    : ℚ → ℚ
  ```

- As mentioned before, DISCO has equirecursive types. The big wrinkle this adds to the type system is that simple structural equality (or unification) no longer suffices; when recursive type synonyms are involved, two types can be the same even though they look different.

The combination of qualified parametric polymorphism, subtyping, and equirecursive types makes for an overall system which seems only barely on the edge of tractability. For the implementation of type inference and checking I relied heavily on Traytel *et al.* [TBN11] who describe the implementation of a similar type system for Isabelle/HOL. There are almost certainly bugs, but overall I am fairly confident in the soundness of the type system.

3.4 Types vs sets

Axiomatic set theory is usually taken as the *de facto* foundation for mathematics. On the other hand, in practice, mathematicians usually behave more as if they were working in some kind of type-theoretic foundation, which makes a statically typed functional language a good match for mathematics as it is practiced (for example, see HoTT [Uni13] and Lean [MU21]). However, one area where there seems to be a big mismatch is in the distinction between *types* and *sets*.

To most mathematicians and every discrete mathematics textbook ever, $\{2,4,7\}$ and $\mathbb{N}$ are both examples of *sets*. The former is finite and the latter (countably) infinite, but they are both fundamentally the same kind of thing, and it makes sense to talk about (for example) their difference, $\mathbb{N} - \{2,4,7\}$, which is also a set. In DISCO, however, $\{2,4,7\}$ and $\mathbb{N}$ are very different things: the former is a value of type `Set(N)`, whereas the latter is a type. $\mathbb{N} - \{2,4,7\}$ is so nonsensical that it is a *syntax* error. One might reasonably wonder: why the mismatch? Why not try to make DISCO more closely align with common mathematical practice, in accordance with DISCO's stated philosophy?

Although conflating sets and types might be fine on a theoretical level (at least, as long as one does not worry about deeper foundational issues), on a practical level it introduces several big problems:

- The ability to use arbitrary finite sets as types would lead to what is essentially a system of *refinement types*. Although this is well-studied and has many practical motivations, it quickly leads to undecidable typechecking, the need for tools like SMT solvers, and the requirement for users to provide annotations to help the system understand why a given type is valid. For example, to typecheck `f : N -> {2,3,7}` would require somehow checking that for any natural number input, the function `f` will always return a either 2, 3, or 7, which could depend on complex reasoning about the behavior of the function. This sort of *refinement type* system is well-studied, but calling out to an SMT solver in order to typecheck a teaching language to be used by students seems like a non-starter.
- Conversely, the ability to use types as value-level sets introduces all sorts of difficulties, chief among which is the fact that most types correspond to *infinite* sets. Set values would have to be represented as some kind of abstract set expressions rather than simply as sets of values, and operations like membership checking become only *semi*-decidable at best. What's more, these infinite set values would not really correspond to their supposed mathematical counterparts in some subtle ways. For example, as everyone knows, the power set of the natural numbers is uncountable; but if `N` were usable as a value of type `Set(N)` in DISCO, then `power(N)` would actually represent the set of *computable* subsets of $\mathbb{N}$, which is countable!

The one slight blurring of categories which seems both feasible and desirable would be the ability to use finite set values as domains for $\forall$ and $\exists$ property quantifiers, so one could write, for example, `forall x in [0..10]. x^2 <= 100`. It should be easy to incorporate such finite sets into the existing machinery for property checking.

In any case, how should we present and explain the relevant distinctions to students? Honestly, I'm not entirely sure. My best approach at the moment revolves around two ideas:

- First, explain to students that DISCO can only represent *finite* sets. This is easy enough to understand: if we allowed infinite sets, then certain operations might require infinitely long computations.
- We can then explain that types can be thought of as a particular collection of "ur-sets" out of which we can build and carve out all other sets. For particularly keen students, we can explain that types are sets with particularly nice structural properties. For example, $\mathbb{N}$ is the unique set that includes 0 and is closed under the successor operation; in contrast, there is nice structural way to define $\{2,4,7\}$ other than just listing its elements. These nice structural properties are precisely what enable decidable typechecking without having to resort to SMT solvers.

3.5 Error messages

When the DISCO project first started, I had grand designs for the way the system would interact with the user in the case of type errors [YEEI18]. Unfortunately, partly because I was intimidated by my own grand designs, and partly because error messages are hard, the system currently does not have very good error messages! For example, here is a terribly uninformative one:

```
Disco> each(3, [1,2,3])
Error: the shape of two types does not match.
https://disco-lang.readthedocs.io/en/latest/reference/shape-mismatch.html
```

In practice, I just tell students to ask me for help when they run into errors they can't figure out, but this obviously limits wider adoption. Improving error messages will be another big focus for work in the upcoming year.

4 Acknowledgements

DISCO was originally born out of a conversation with Harley Eades at TFPIE in 2016, and I'm very grateful to Harley for those initial conversations and many good ideas. I am also grateful to the many people who have contributed to the DISCO codebase over the years, including my students Callahan Hirrel, Bosco Ndemeye, Sanjit Kalapatapu, Jacob Hines, Eric Pinter, and Daniel Burnett, as well as other collaborators including Shay Lewis, Ryan Yates, Tristan de Cacqueray, and Chris Smith.

References

[CH00] Koen Claessen & John Hughes (2000): *QuickCheck: a lightweight tool for random testing of Haskell programs*. In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pp. 268–279.

[CM13] Association for Computing Machinery (2013): *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Available at `http://www.acm.org/education/CS2013-final-report.pdf`.

[DE04] Kees Doets & Jan van Eijck (2004): *The Haskell Road to Logic, Maths, and Programming*. Texts in Computing. King's College Publications.

[Gol91] David Goldberg (1991): *What every computer scientist should know about floating-point arithmetic*. *ACM computing surveys (CSUR)* 23(1), pp. 5–48.

[Hen02] P. B. Henderson (2002): *Functional and declarative languages for learning discrete mathematics*. Technical Report 0210, University of Kiel.

[Lib07] Liberal Arts Computer Science Consortium (2007): *A 2007 Model Curriculum for a Liberal Arts Degree in Computer Science*. *J. Educ. Resour. Comput.* 7(2), doi:10.1145/1240200.1240202. Available at `http://doi.acm.org/10.1145/1240200.1240202`.

[MU21] Leonardo de Moura & Sebastian Ullrich (2021): *The lean 4 theorem prover and programming language*. In: *International Conference on Automated Deduction*, Springer, pp. 625–635.

[OEI22] OEIS Foundation Inc. (2022): *The On-Line Encyclopedia of Integer Sequences*. Available at `http://oeis.org/`.

[OHP06] John O'Donnell, Cardelia Hall & Rex Page (2006): *Discrete Mathematics Using a Computer*. Springer-Verlag London.

[SW02] C. Scharff & A. Wildenberg (2002): *Teaching discrete structures with SML*. Technical Report, University of Kiel.

[TBN11] Dmitriy Traytel, Stefan Berghofer & Tobias Nipkow (2011): *Extending Hindley-Milner type inference with coercive structural subtyping*. In: *Asian Symposium on Programming Languages and Systems*, Springer, pp. 89–104.

[Uni13] The Univalent Foundations Program (2013): *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study.

[Van11] Thomas VanDrunen (2011): *The Case for Teaching Functional Programming in Discrete Math*. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, ACM, New York, NY, USA, pp. 81–86, doi:10.1145/2048147.2048180. Available at `http://doi.acm.org/10.1145/2048147.2048180`.

[Wai92] Roger L. Wainwright (1992): *Introducing Functional Programming in Discrete Mathematics*. In: *Proceedings of the Twenty-third SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '92, ACM, New York, NY, USA, pp. 147–152.

[Xin08] Cong-Cong Xing (2008): *Enhancing the Learning and Teaching of Functions Through Programming in ML*. *J. Comput. Sci. Coll.* 23(4), pp. 97–104. Available at `http://dl.acm.org/citation.cfm?id=1352079.1352096`.

[YEEI18] Brent A Yorgey, Richard A Eisenberg & Harley D Eades III (2018): *Explaining Type Errors*. In: *Off The Beaten Track*.