# DISCO: A Functional Programming Language for Discrete Mathematics

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

DISCO is a pure functional programming language designed to be used in a Discrete Mathematics course. [TODO: statically typed, math notation. features: property testing, arithmetic patterns, equirecursive types, subtyping.] [TODO: available on GitHub.]

## 1 Introduction

Many computer science curricula at the university level include *discrete mathematics* as a core requirement [Ass13]. 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 [Ass13] 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.
- 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

Submitted to:
TFPIE 2023

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.
- 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 schools 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. [TODO: examples in section 2]
- As much as possible, the language [TODO: exceptions in section 3]
- Although there is as yet no data to back this up, the language should be easy for instructors to learn, even mathematicians without much programming experience.

[TODO: say what Disco is. Say where to find it, where to find documentation. Note it is available via repl.it!] [TODO: Note this also contains my very opinionated ideas about how to teach these things, what language features we should want.]

## 2 DISCO by Example

In order to introduce the main features of the language, [TODO: 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 (regular comments start with `--`). This documentation can be later accessed with the `:doc` command at the REPL prompt:

[TODO: automatically typeset REPL interactions from just input?]

```
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. [TODO: using QuickCheck + enumeration library.] 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.

[TODO: tuples for products of more than 2 types]

## 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 pp. 4–11 [DE04], and has been transcribed from Haskell into DISCO. (DISCO also has a much more efficient built-in primality testing function, so this can be used as an example to teach students, but need not be used in practice.)

```
||| 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

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`. It is supposed to be reminiscent of mathematical notation like

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

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 curly brace in any case). The intention is that writing `{? ... ?}` 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 keyword `and` instead of `/\`. These are synonymous—in fact, `&&` and the Unicode character $\wedge$ are also accepted. In general, DISCO's philosophy is to allow multiple syntaxes for various things. 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 $\mathbb{N}$, `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; it's just that some functions return other functions (curried style) and some functions 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.

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.

This example again uses case expressions; it may seem odd to use case expressions with only one case, but this is done in order to be able to pattern-match on the result of the recursive call to `zOrder'`.

```
!!! 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

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 can be used as a pattern, as long as it has exactly one variable and uses only basic arithmetic operators; 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 the type `Set(A)` are finite sets with elements of type `A`. Disco supports the usual set operations (union, intersection, difference, power set), and sets can be created by writing a finite set literal (`{1,3,5,7}`), using ellipsis notation (`{1, 3 .. 7}`), or using a set comprehension (`{2x+1 | x in {0 .. 3}}`). Listing 4 shows a small 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 multiset 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.

One 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.

[TODO: Talk about sets vs types.]

## 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) [TODO: cite].

```
-- 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

```
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

```
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 : (a -> Bool) * List(a) -> Bool
all(P, as) = reduce(~/\~, true, each(P, as))

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

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

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

Listing 6: Counting trees

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. This makes the implementation of the type system more complex, but using equirecursive types 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. 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 Math class rarely have more than two or three summands, and working directly with primitive sum and product types helps students make connections to other things they have already seen, such as Cartesian product and disjoint union of sets.

In addition to `extendSequence`, the `oeis` module also provides a function `lookupSequence : List(N) -> Unit + List(Char)`, 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 parametric polymorphism. (Internally, it also supports qualified polymorphism, but this feature is not available through the surface syntax.)

## 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 very 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:

```
-------------------------------------------------------------------
-- 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
g2(x) = x - 1/2
-- FILL IN YOUR DEFINITION HERE
```

Listing 7: Defining and testing bijections

- 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

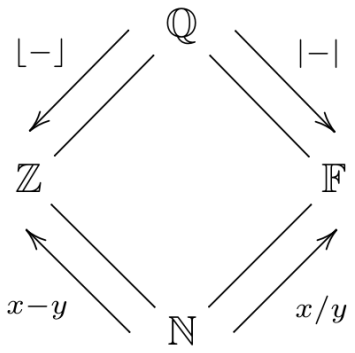


Figure 1: DISCO's numeric type lattice

$\mathbb{Z}$ 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 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 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 about these cases is worthwhile, since they are likely to confuse 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 "evenly divides" relation is also traditionally written with a vertical bar, as in $3 \mid 21$,

```
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

but DISCO does not support this notation. Including it makes the grammar extremely ambiguous. (And besides, Dijkstra tells us that we should not use a physically symmetric operator symbol for a nonsymmetric relation! [TODO: cite]) Instead, DISCO provides `divides` as an infix operator. In my experience students had no problem remembering the difference.

- The equality symbol $=$ is also overloaded to denote both definition and equality testing. 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.
- 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. 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 $x(y+2)\ldots$"?

# 4 Types

# 5 Tooling

# 6 Discussion

# 7 Acknowledgements

Harley Eades, Callahan Hirrel, Bosco Ndemeye, Sanjit Kalapatapu, Jacob Hines, Daniel Burnett

# 8 Bibliography

# References

[Ass13] 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`.

[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.

[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`.