Math Notation

James Wilcox and Kevin Zatloukal

May 2023

It is important for us to have tools that we can use to reason about code outside the context of a specific programming language. This article defines the mathematical tools that we will use for this purpose, starting first with data ("types") and then moving to code (" functions").

Data Types

Each data type defines a collection of allowed values. If "x" is a variable and "T" is a data type, then the statement "$x : T$", read "x has type T", says that the value of x is something included in T.

In math terms, data types are "sets". The most basic sets that we will use for data types are the following:

$\mathbb{N}$	all non-negative integers ("natural" numbers)
$\mathbb{Z}$	all integers
$\mathbb{R}$	all real numbers
$\mathbb{B}$	the boolean values (T and F)
$\mathbb{S}$	any character
$\mathbb{S}^*$	any sequence of characters ("strings")

Compound Types

We can construct new types from existing types A and B using the following operators:

- **Union**: $A \cup B$ is the set that includes every value in either A or B (or both)
- **Tuple**: $A \times B$ is the set of all pairs of the form (a, b) with $a : A$ and $b : B$
- **Record**: $\{x : A, y : B\}$ is the set of all records containing fields called "x" and "y" of types A and B, respectively

Both this tuple and record type represent values that include **both** a value of type A and a value of B, together in one object. The only difference is that the record identifies to the two parts by giving them names (x and y), whereas the parts of a tuple are identified by order. In the tuple $(a, b) : A \times B$, it is the first part "a" that has type A.

If $f : \{x : A, y : B\}$ is a record, then we can refer to the part of type A as "$f.x$". On the other hand, if $t : A \times B$, then we don't have any immediate way to refer to the parts. To talk about them, we have to define new variable names. Specifically, we can say "define $(a, b) := t$", and then "a" refers to the A part of t and "b" refers to the B part.[1]

Both tuples and records can have more than two components, if desired.

[1]The symbol ":=" indicates a definition, rather than an equation. In general, the equation "$(a, b) = t$" could be true or false, depending on the values of these variables, whereas the definition "$(a, b) := t$" tells us that the previous equation certainly holds because we are defining "a" and "b" to make it so.

Inductive Types

The most powerful way to define new types combines the power of union and tuple / record and a critical element missing above, namely, *recursion.* We call these "inductive types".

In general, an inductive type definition looks like the following:

$$\textbf{type } T := \mathsf{A} \mid \mathsf{B}(u : \mathbb{N}) \mid \mathsf{C}(v : \mathbb{N},\ w : T) \mid \mathsf{D}(x : \mathbb{N},\ y : T,\ z : T)$$

The "|" here is like union ($\cup$) except that the different possibilities are known to be *distinct*, whereas union can be formed between data types that include some common values.

The names "A", "B", "C", and "D" are "constructors". Each of them describes a distinct way to create values of type T. Constructors can optionally take arguments ("A" has no arguments, but the rest do). The argument types can be any known type including "T" itself. The latter case is called a "recursive argument".

Values of type T are written by describing how the value was created via constructors. For example, "A", "$\mathsf{B}(1)$", and "$\mathsf{C}(2, \mathsf{A})$" are all values of type T. (Notice the recursion in the last example.)

It is important to note that these are not function calls. Values of type T are descriptions of the sequence of constructors used to created them. Two values of type T are equal if and only if they were created by applying the *exact same* sequence of constructors, with all the same arguments. Hence, $\mathsf{A} = \mathsf{A}$ and $\mathsf{B}(3) = \mathsf{B}(3)$, but $\mathsf{A} \neq \mathsf{B}(u)$, no matter the value of $u : \mathbb{N}$. Similarly, we have $\mathsf{C}(1, \mathsf{B}(2)) \neq \mathsf{C}(1, \mathsf{B}(3))$ because the former passed a 2 to the B constructor and the latter a 3.

It is useful to have names for different sorts of inductive types depending on the maximum number of recursive arguments amongst its constructors. For reasons we will see in class, if this number is 0 (i.e., it has no recursive arguments), we call it "enum-like"; if this number is 1 (i.e., some constructor has one recursive argument), we call it "list-like"; and if it is 2 or more, we will call it "tree-like".

An inductive type with no recursive arguments is called enum-*like*, but if the constructors have no arguments at all, then it can actually be called an "enum" type. As an example, the boolean type could be defined as an enum like this:

$$\textbf{type } \mathbb{B} := \mathsf{T} \mid \mathsf{F}$$

This says that the ways to directly create boolean values are by writing "T" or "F".

Natural Numbers

It is also possible to define the natural numbers inductively, albeit with different notation, as follows:

$$\textbf{type } \mathbb{N} := \mathsf{zero} \mid \mathsf{succ}(n : \mathbb{N})$$

The number 2, for example, would then be written "$\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))$". ("succ" is short for successor.)

That notation is obviously inconvenient, so we will stick to the usual algebraic numerals, but it is important for us to recognize that the natural numbers really are an inductive type, even if we don't write them with inductive notation. In particular, every natural number is either 0 or it was created by adding one to another natural number, i.e., it is either 0 or $n+1$ for some natural number n. Since $\mathbb{N}$ is an inductive type with just these two constructors, the two cases just described are exclusive and exhaustive.

Functions

Our basic notation for defining functions looks like this:

$$\textbf{func } \mathsf{double}(n : \mathbb{N}) := 2n$$

Here, the type restriction that n must be a natural number is included on the left-hand side. The right-hand side can be any valid mathematical expression in the declared variables. In this case, for any natural number n that is passed in, the function returns twice that number.

We are free to use any types that can be defined as described in the previous section. For example, the following is a function that takes a point in the plane as a record and returns its distance from the origin.

$$\textbf{func } \mathsf{dist}(p : \{x : \mathbb{R},\ y : \mathbb{R}\}) := (p.x^2 + p.y^2)^{1/2}$$

If we define a name for this record type, then we can shorten the function declaration as follows:

$$\begin{aligned}&\textbf{type}\ \mathsf{Point} := \{x : \mathbb{R},\ y : \mathbb{R}\}\\ &\textbf{func}\ \mathsf{dist}(p : \mathsf{Point}) := (p.x^2 + p.y^2)^{1/2}\end{aligned}$$

Pattern Matching

The functions above were very simple, calculating the return value using the same expression for all inputs. To write more complex functions, we need to be able to break the inputs up into different cases and give each their own return value expression.

Our primary way of splitting the inputs into cases will be via **pattern matching**. For each case, we write a function definition where, rather than declaring an argument like "$a : A$" that could have any value of type A, we have a set of cases and write a function definition for each with an expression that describes the inputs that fall into that case.

As a simple example, consider the enum type $\mathbb{B}$, and suppose that we want to define the function "not" that flips the value between true and false. Rather than defining the function as "$\mathsf{not}(b : \mathbb{B}) := \dots$", we can split the true and false cases using pattern matching as follows:

$$\begin{aligned}\textbf{func}\ &\mathsf{not}(\mathsf{T}) := \mathsf{F}\\ &\mathsf{not}(\mathsf{F}) := \mathsf{T}\end{aligned}$$

Since every input is either T or F, exactly one of these rules applies. I.e., they are exclusive and exhaustive. (We will leave off the type on a literal value when its type is clear, e.g., we write "T" not "$\mathsf{T} : \mathbb{B}$".)

We can likewise use pattern matching on $\mathbb{N}$. For example, instead of defining "double" as we did above, we could instead define it like this:

$$\begin{aligned}\textbf{func}\ &\mathsf{double}(0) && := 0\\ &\mathsf{double}(n+1) && := \mathsf{double}(n) + 2 \qquad \text{for any } n : \mathbb{N}\end{aligned}$$

(Here, there wasn't a natural place in the expression "$n + 1$" to indicate that n has type $\mathbb{N}$, so I wrote that on the right side, just to be safe. However, in this case, it was probably already clear that we must have $n : \mathbb{N}$ in order for $n + 1$ to have type $\mathbb{N}$, so this was possibly unnecessary.)

It is important that **exactly one rule** applies for any valid input. As we discussed above, the cases 0 and $n + 1$ are exclusive and exhaustive for $\mathbb{N}$, so the previous example is a valid way to define the function. If we wanted, we could instead split the inputs into cases 0, 1, and $n + 2$ (for any $n : \mathbb{N}$) because those are also exclusive and exhaustive.

For inductive types, it is natural to have exactly one rule for each constructor. Each value was created by using a constructor, so exactly one rule will apply. However, we can match more general patterns as well.

For example, suppose that we have the following record type, which stores a real number and a boolean:

$$\textbf{type}\ \mathsf{R} := \{x : \mathbb{R},\ b : \mathbb{B}\}$$

In this case, we could split the inputs into cases based on just the value of the boolean field like this:

$$\begin{aligned}\textbf{func}\ &f(\{x : \mathbb{R}, \mathsf{T}\}) := x\\ &f(\{x : \mathbb{R}, \mathsf{F}\}) := -x\end{aligned}$$

Exactly one pattern matches any record of type R, so our rules are exclusive and exhaustive as required.

Side Conditions

In some cases, it will be necessary to write side conditions that restrict when a given pattern is allowed to match. For example, we could define the "not" example from before instead like this:

$$\begin{aligned}\textbf{func}\ &\mathsf{not}(b : \mathbb{B}) := \mathsf{F} \qquad \text{if } b = \mathsf{T}\\ &\mathsf{not}(b : \mathbb{B}) := \mathsf{T} \qquad \text{if } b = \mathsf{F}\end{aligned}$$

As before, we must be careful to make sure that the conditions are exclusive and exhaustive.

In general, side conditions are harder to work with when reasoning. (Unlike pattern matching, they require an explanation of why that side condition holds before the definition can be applied.) For that reason, we **always prefer** pattern matching.

We will only use side conditions when pattern matching notation is inapplicable. As an example, if $x : \mathbb{R}$ is an argument and we want to define our function differently based on whether $x > 1$, then we would need a side condition because we do not have a pattern to describe real numbers greater than 1.