# Proof assistants, dependent types, and modeling...?

George McNinch

2025-04-27 16:22:31 EDT (george@valhalla)

# Outline

# Python

- I want to try to quickly something called *dependent types* which enable *proofs* in the context of computer code.
- The language we have used in this course – `python` – is *dynamically typed*:
  *Python is called a dynamically typed language because you do not need to declare the type of a variable when you create it; the type is determined automatically based on the value assigned to it.*

# A python example

- So for example we can write

```
import numpy as np

def f(a):
    return a + np.array([1,1,1])
```

without first declaring that `a` is an `np.array` of length 3. We just get a *runtime error* if `a` isn't of the correct form.

- we get

```
f([1,0,-1])
```

```
array([2, 1, 0])
```

## continued

```python
import numpy as np

def f(a):
    return a + np.array([1,1,1])
```

```python
try:
    f([1,0,-1,0])
except:
    print("runtime error...")
```

```
runtime error...
```

# Statically typed language

- In contrast, in a statically typed language you have to be more explicit about things.
- Because my plan for this talk is ultimately to describe a little bit about the `Lean` language/proof-assistant, I'm going to discuss typing for `Lean`, but until I discuss dependent types, my remarks mostly describe typing for any language in the `ML` family (`Haskell`, `OCaml`, ...).

# An add function

- Let's define a function `add` that that adds 2 lists of natural numbers: we view the arguments as "vectors" and we want the function to add these vectors.
- The intent is that `add [1, 1, 1] [1, −2, 1]` should return something like `[2, −1, 2]`. In this case, the type system will only permit you to call the function `add` with two arguments, both required to be lists of natural numbers.
- So e.g. `add ["a"] [1]` should fail, but not with a *runtime error* – the language "knows" this invocation is prohibited because it can infer the type of `["a"]` as `List String` instead of `List ℕ`.

# Error handling via Option

- But we have to worry about `add [1,1] [1,1,1]`.
- One way to deal with this is to have a type for error handling. Here if `a` is a type, then `Option a` is the type which can have values either `none` or `some a`.
- Our function can return a `Option` value.
  The *signature* of our function will be

```
def add (a: List ℕ) (b: List ℕ)
  : Option (List ℕ)
```

So an invocation of `add` can either return `none` or it can return `some [ ... ]`.

# The add function

```
def add (a: List ℕ) (b: List ℕ) :
    Option (List ℕ) :=
    if a.length == b.length then
      match a,b with
      | [],_ => some []
      | _,[] => some []
      | (c::cs), (d::ds) => do
        let rest ← add cs ds
        pure $ (c+d)::rest   -- this returns
                             -- a `some`-value.
    else
      none
```

# add function results

- So for example

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

evaluates to `some [4,6]` while

- while

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

evaluates to `none`.

- The main drawback with this approach is that after using it, one is then committed to carrying around values of the `Option` type

# A dependent type

- But `Lean` – and other dependently typed languages such as `Idris`, `Agda`, ... – offers us something more.
- We can encode the statement "a and b have the same length" using a piece of data. We view this data as a *proof*, or as *evidence* of the equality.
- If `x y : ℕ` then `x = y` is a *type*; more precisely, `x = y` is a *Proposition* in `Lean`.
- In contrast, `x == y` is really a boolean valued procedure, with signature something like

```
def (==) {a : Type} (x y : a) : Boolean
```

# equality types, continued

- For example, `Lean` knows statements like

```
theorem eq_succ { x y : ℕ } :
  x = y → (x+1) = (y+1)
```

which we read as "if `x` and `y` are equal, then so are `x+1` and `y+1`".
(more precisely: it is easy to prove such statements using `Lean`)

# Type-safe add

- One way to use this equality type is to require such evidence as an argument to a function
- For example, we can require the user provide an equality proof to *invoke* our addition function.
- Here is possible type-signature for such a function

```
def add_safe (a:List ℕ) (b:List ℕ)
  (p:a.length = b.length) : List ℕ
```

- Thus, one can make a call

```
add_safe l1 l2 p
```

where `l1 l2 : List ℕ` are lists of natural numbers, and where `p : l1.length = l2.length` is a proof that the lists have the same length.

# Type-safe add (implemented)

- here is the code

```
def add_safe (a:List ℕ) (b:List ℕ)
    (p:a.length = b.length) : List ℕ :=
  match a,b with
  | [],[] => []
  | z::zs, w::ws =>  by
    have h : zs.length = ws.length := by
       repeat rw [List.length_cons]  at p
       linarith
    exact (z+w)::add_safe zs ws h
```

- note that we needed to construct the proof `h` from the hypothesis `p` in order to be able to recursively invoke the function `add_safe` on the shorter lists `zs` and `ws`.

# Type-safe add, continued

- Now

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

  evaluates to `[2,4,7]`. Here `rfl` is a proof that `[1,2,3].length = [1,2,4].length` – this proof amounts to the "reflexive law of equality".

- in contrast

```
add_safe [ 1,2,3] [1,2,4,5] rfl
```

  doesn't type-check in `Lean`.

# Vectors

- a basic example of a dependent type is that of a *vector*
- the idea is that the type itself indicates how many entries the vector has. This is like saying that $\mathbb{R}^3$ is a type
- of course, you can make a type for "3-tuples of floats" in more-or-less any typed language. But dependently typed languages let you make a type for "`n`-tuples of floats" where `n` is a variable natural number.

# vectors continued

- here is a definition of a vector (this isn't actually the definition used in `Lean`, which is more complicated for reasons that aren't really relevant to our discussion).

```
inductive vect : Type → ℕ →  Type where
| vnil : vect a 0
| vcons (x:a) (v:vect a n)
    : vect a (Nat.succ n)
```

- `vect` is an *inductive type*. There are two *constructors*: `vnil` is the "empty vector" (of length 0) and `vcons` constructs a vector of length `n` from an element and a vector of length `n−1`
- thus we can create a vector of length two of natural numbers

```
vect.vcons 1 (vect.vcons 2 vect.vnil)
```

which "is" the vector `[1,2]`

# Vectors (notation)

- we can simplify notation a bit using

```
infixr:67 " ::: " => vect.vcons
```

- now our vector representing `[1,2]` above can be entered as

```
1 ::: 2 ::: vnil
```

# vectors as dependent type

- `vect a n` is a *dependent type*
- the type `vect a n` has a type parameter - in this case, `a`, which is an arbitrary type. But this doesn't make it a dependent type. E.g. this is essentially the same as the `Option` or `List` type constructors we have seen before, and which many non-dependently typed languages have.
  e.g. the definition of `Option` is as follows. The type doesn't depend on a value

```
inductive Option (α:Type) where
| none : Option α  -- no value
| some (val:α) : Option α
```

- what makes `vect a n` dependent is the value parameter `n`, a natural number

# code for adding our vectors

- rather than giving our `add_safe` function a proof that its arguments are equal-length lists, we can instead define an `add_vect` function with signature

```
def add_vect {n :ℕ} (av : vect ℕ n)
                    (bv : vect ℕ n)
    : vect ℕ n
```

- thus `add_vect` will only accept as arguments vectors of the *same length*

# code for adding our vectors

The code is actually simpler than that of our earlier `add_safe`:

```
def add_vect {n :ℕ} (av : vect ℕ n)
                     (bv : vect ℕ n)
   : vect ℕ n :=
  match av,bv with
  | vect.vnil,vect.vnil => vect.vnil
  | a ::: ar, b ::: br =>
         (a+b) ::: add_vect ar br
```

# adding some vectors

```
add_vect (1 ::: 2 ::: 3 ::: vect.vnil)
         (1 ::: 1 ::: 1 ::: vect.vnil)
```

evaluates to `2 ::: 3 ::: 4 ::: vect.vnil`

# Proving statements about constructions

- `List` in lean is another type constructor

```
inductive List (α:Type) where
| nil : List α       -- empty list
| cons (x:α) (xs:List α) : List α
```

  where we define notation `[]` for `nil` and `x :: xs` for `cons x xs`.
- thus e.g.

```
1 :: 2 :: 3 :: nil
```

  is the list `[1,2,3]`
- the main difference between `List` and `Vector` of course is that `Vector` s have a fixed length, while `List` s don't

# appending lists

- Here is some `Lean` code that *appends two lists*.

```
def append {a:Type} (xs ys : List a)
  : List a :=
  match xs with
  | [] => ys
  | z :: zs => z :: append zs ys
```

- e.g.

```
append ["a", "b", "c"] ["d", "e"]
```

  evaluates to `["a", "b", "c", "d", "e"]`

- Now, we are going to prove a property about this `append` function: namely, that the length of the appended lists is the sum of their lengths.

# the proof

- here is the proof in `Lean`

```
theorem append_length {a:Type}
      (xs ys : List a)
      : (append xs ys).length =
        xs.length + ys.length := by
  induction xs with
  | nil => simp [append]
  | cons z zs ih =>
      simp [append, ih]
      linarith
```

- you can view this theorem `append_length` as a function of `xs` and `ys`, whose value is the indicated equality `Proposition`.

# the proof continued

- here is the proof in `Lean`

```
theorem append_length {a:Type}
      (xs ys : List a)
      : (append xs ys).length =
        xs.length + ys.length := by
  induction xs with
  | nil => simp [append]
  | cons z zs ih =>
      simp [append, ih]
      linarith
```

- The proof is by induction on the length of the first list.
- In the base case where the first list is empty, the proof boils down to the observation that `append [] ys` is equal to `ys`. We are able to produce the pf using the *simplifier tactic* `simp`.

# proof continued 2

- here is the proof in `Lean`

```
theorem append_length {a:Type}
      (xs ys : List a)
      : (append xs ys).length =
        xs.length + ys.length := by
  induction xs with
  | nil => simp [append]
  | cons z zs ih =>
      simp [append, ih]
      linarith
```

- when the first list is non-empty, it must the form `z::zs` and we then have the inductive hypotheses that `append zs ys` has length equal to `zs.length + ys.length`. Using this, the required result is again provided by `simp`.

# Relevance to modeling?

Some thoughts, remarks, and questions:

- would more type-safe linear algebra functions be useful in the setting of math modeling?
  - advantages: types help prevent certain types of errors, and proofs provide assurance of correctness
  - disadvantages: more complex to create code
- one can imagine proofs of properties of results obtained from modeling software; how useful would this be?
- I'm aware of a fair amount of recent formalization activity in *pure mathematics*, but I know less about its adoption in math modeling settings
- having a machine-usable language for mathematical proofs is a pre-requisite for doing machine-learning *about* mathematical statements