A Journey Through Types

Beck, Calvin
hobbes@ualberta.ca

April 1, 2017

What is this Talk about?

Types! This presentation hopes to address the following:

- How types make things easier to write.
- How types can help you write correct software.

What is this Talk about?

Types! This presentation hopes to address the following:

- How types make things easier to write.
- How types can help you write correct software.

Somewhat of a whirlwind introduction. Let me know if you're lost, because this talk is all over the place!

What are we trying to solve?

You think that this is normal...

What are we trying to solve?

You think that this is normal...

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not subscriptable
```

What are we trying to solve?

You think that this is normal...

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not subscriptable
```

... It's not!

What is a type?

A type describes what a value “is”.

What is a type?

A type describes what a value “is”.

You have probably heard of this as “how the bits are stored in memory.”

What is a type?

A type describes what a value “is”.
You have probably heard of this as “how the bits are stored in memory.”
It’s a bit more than that!

What is a type?

A type describes what a value "is".

You have probably heard of this as "how the bits are stored in memory."

It's a bit more than that!

- Tell us how to use values.
  - Tells us what operations are defined on the types.
  - Can you add things of this type?
  - Can a function take a value of this type as an argument?
  - What kind of stuff does this function return?

What is a type?

A type describes what a value "is".

You have probably heard of this as "how the bits are stored in memory."

It's a bit more than that!

- Tell us how to use values.
  - Tells us what operations are defined on the types.
  - Can you add things of this type?
  - Can a function take a value of this type as an argument?
  - What kind of stuff does this function return?
- Documentation

What is a type?

A type describes what a value "is".

You have probably heard of this as "how the bits are stored in memory."

It's a bit more than that!

- Tell us how to use values.
  - Tells us what operations are defined on the types.
  - Can you add things of this type?
  - Can a function take a value of this type as an argument?
  - What kind of stuff does this function return?
- Documentation
- Rejection of general nonsense: 357^{circles}

What is a type?

A type describes what a value “is”.
You have probably heard of this as “how the bits are stored in memory.”
It’s a bit more than that!

- Tell us how to use values.
  - Tells us what operations are defined on the types.
  - Can you add things of this type?
  - Can a function take a value of this type as an argument?
  - What kind of stuff does this function return?
- Documentation
- Rejection of general nonsense: 357^{circles}
  - # NO MORE NULL REFERENCE EXCEPTIONS!

The types you may have seen: Python

```python
def my_sort(xs):
    if xs == []:
        return xs
    else:
        first_elem = xs[0]
        rest = xs[1:]

        smaller = my_sort([x for x in rest if x <= first_elem])
        larger = my_sort([x for x in rest if x > first_elem])

        return smaller + [first_elem] + larger


def my_factorial(n):
    if n == 0:
        return 1
    else:
        return n * my_factorial(n-1)
```

- No types to help document functions.
- No types to catch errors at compile time.
  - Tests can help...
  - But it's nice to not have to worry about certain errors at all.

The types you may have seen: Python

- What if we could force functions to be compartmentalized?
  - No sneaky IO
  - No hidden global states

The types you may have seen: Python

- What if we could force functions to be compartmentalized?
  - No sneaky IO
  - No hidden global states
- Wouldn't it be nice to have a description of what a function can and can't do in a concise format?

The types you may have seen: Python

- What if we could force functions to be compartmentalized?
  - No sneaky IO
  - No hidden global states
- Wouldn't it be nice to have a description of what a function can and can't do in a concise format?
- Could the compiler tell us when our function deviates from these descriptions?
  - Why wait until runtime to find your mistakes?

The types you may have seen: Java

```
Integer factorial(Integer n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

ArrayList<Integer> my_sort(ArrayList<Integer> xs) {
    if (xs.size() == 0) {
        return new ArrayList<Integer>();
    }
    else {
        // ...
        // Calvin is too lazy to write Java
        // ...
    }
}
```

The types you may have seen: Java

```java
Integer factorial(Integer n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

ArrayList<Integer> my_sort(ArrayList<Integer> xs) {
    if (xs.size() == 0) {
        return new ArrayList<Integer>();
    }
    else {
        // ...
        // Calvin is too lazy to write Java
        // ...
    }
}
```

- Can see what functions accept and return!

The types you may have seen: Java

```java
Integer factorial(Integer n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

ArrayList<Integer> my_sort(ArrayList<Integer> xs) {
    if (xs.size() == 0) {
        return new ArrayList<Integer>();
    }
    else {
        // ...
        // Calvin is too lazy to write Java
        // ...
    }
}
```

- Can see what functions accept and return!
- Null references... :c

The types you may have seen: Java

```
Integer factorial(Integer n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

ArrayList<Integer> my_sort(ArrayList<Integer> xs) {
    if (xs.size() == 0) {
        return new ArrayList<Integer>();
    }
    else {
        // ...
        // Calvin is too lazy to write Java
        // ...
    }
}
```

- Can see what functions accept and return!
- Null references... :c
- Very verbose. Lots of additional syntactic cruft.

The types you may have seen: Java

```java
Integer factorial(Integer n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

ArrayList<Integer> my_sort(ArrayList<Integer> xs) {
    if (xs.size() == 0) {
        return new ArrayList<Integer>();
    }
    else {
        // ...
        // Calvin is too lazy to write Java
        // ...
    }
}
```

- Can see what functions accept and return!
- Null references... :c
- Very verbose. Lots of additional syntactic cruft.

Types aren't bad...

The types you may have seen: Java

```java
Integer factorial(Integer n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

ArrayList<Integer> my_sort(ArrayList<Integer> xs) {
    if (xs.size() == 0) {
        return new ArrayList<Integer>();
    }
    else {
        // ...
        // Calvin is too lazy to write Java
        // ...
    }
}
```

- Can see what functions accept and return!
- Null references... :c
- Very verbose. Lots of additional syntactic cruft.

Types aren't bad... Java is bad.

Greener pastures and better ideas

What do we want in our types?

Greener pastures and better ideas

What do we want in our types?

- Catch errors at compile time!
  - If something is "wrong", then why wait for runtime to tell us?

Greener pastures and better ideas

What do we want in our types?

- Catch errors at compile time!
  - If something is "wrong", then why wait for runtime to tell us?
- Ease reading and writing of programs.
  - Act as a kind of documentation.
  - Guide us when writing programs.
  - Stop us from making mistakes.

Greener pastures and better ideas

What do we want in our types?

- Catch errors at compile time!
  - If something is "wrong", then why wait for runtime to tell us?
- Ease reading and writing of programs.
  - Act as a kind of documentation.
  - Guide us when writing programs.
  - Stop us from making mistakes.
- Not too much verbosity.
  - Nice, clean syntax!

Introducing Haskell!

```
def my_sort(xs):
    if xs == []:
        return xs
    else:
        first_elem = xs[0]
        rest = xs[1:]

        smaller = my_sort([x for x in rest if x <= first_elem])
        larger = my_sort([x for x in rest if x > first_elem])

        return smaller + [first_elem] + larger


def my_factorial(n):
    if n == 0:
        return 1
    else:
        return n * my_factorial(n-1)
```

Introducing Haskell!

```python
def my_sort(xs):
    if xs == []:
        return xs
    else:
        first_elem = xs[0]
        rest = xs[1:]

        smaller = my_sort([x for x in rest if x <= first_elem])
        larger = my_sort([x for x in rest if x > first_elem])

        return smaller + [first_elem] + larger


def my_factorial(n):
    if n == 0:
        return 1
    else:
        return n * my_factorial(n-1)
```

```haskell
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]


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

Introducing Haskell!

```haskell
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]


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

- Type inference: compiler can figure out the types of things.
- Nice, relatively specific types.

```haskell
-- Causes a type error, because it doesn't make sense.
mySort [factorial, (*2)]
```

Something similar in Python would only be caught at runtime

We need Ord!

You might think you could do this:

```
-- Instead of: Ord a => [a] -> [a]
mySort :: [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]
```

We need Ord!

You might think you could do this:

```
-- Instead of: Ord a => [a] -> [a]
mySort :: [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]
```

... But this actually causes a type error!

We need `Ord`!

You might think you could do this:

```
-- Instead of: Ord a => [a] -> [a]
mySort :: [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]
```

... But this actually causes a type error!

- *a* could be *any type*
- This type could be unorderable
    - Like a function, or a picture
- Need the constraint so we know we can perform comparisons!

We need Ord!

You might think you could do this:

```
-- Instead of: Ord a => [a] -> [a]
mySort :: [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]
```

... But this actually causes a type error!

- *a* could be *any type*
- This type could be unorderable
  - Like a function, or a picture
- Need the constraint so we know we can perform comparisons!

Haskell makes sure we can only perform operations that are defined on values of a given type, but allows us to be general about it. This function works with any orderable element still, and not just a fixed type.

More care in a type

Haskell is somewhat careful about what values inhabit a type.

More care in a type

Haskell is somewhat careful about what values inhabit a type.

- No “null” values which inhabit every type.

More care in a type

Haskell is somewhat careful about what values inhabit a type.

- No “null” values which inhabit every type.
- This keeps it so that, for the most part, elements of a type act the same way.

More care in a type

Haskell is somewhat careful about what values inhabit a type.

- No “null” values which inhabit every type.
- This keeps it so that, for the most part, elements of a type act the same way.
- Operations on elements of a type work on all values, so no runtime exceptions are raised!

More care in a type

Haskell is somewhat careful about what values inhabit a type.

- No "null" values which inhabit every type.
- This keeps it so that, for the most part, elements of a type act the same way.
- Operations on elements of a type work on all values, so no runtime exceptions are raised!

This helps to keep everything sane!

Maybe maybe!

Sometimes you need something *like* a null. Maybe a function can't always compute an answer!

Maybe maybe!

Sometimes you need something *like* a null. Maybe a function can't always compute an answer!

Enter maybe types:

```
data Maybe a = Just a | Nothing
```

Maybe maybe!

Sometimes you need something *like* a null. Maybe a function can't always compute an answer!

Enter maybe types:

```
data Maybe a = Just a | Nothing
```

- Not just null!

Maybe maybe!

Sometimes you need something *like* a null. Maybe a function can't always compute an answer!

Enter maybe types:

```
data Maybe a = Just a | Nothing
```

- Not just null!
- Type checker can tell us when we need to handle null.

Maybe maybe!

Sometimes you need something *like* a null. Maybe a function can't always compute an answer!

Enter maybe types:

```
data Maybe a = Just a | Nothing
```

- Not just null!
- Type checker can tell us when we need to handle null.
- Compile time errors if we don't handle null!

Maybe examples

```
-- Find out where a value is in a function.
getIndex :: Eq a => a -> [a] -> Maybe Integer
getIndex = getIndexAcc 0

-- Helper function that remembers our position in the list.
getIndexAcc :: Eq a => Integer -> a -> [a] -> Maybe Integer
getIndexAcc pos value [] = Nothing
getIndexAcc pos value (x::xs) = if x == value
                                then Just pos
                                else getIndexAcc (pos+1) xs

-- A dictionary of all the important words.
dictionary :: [String]
dictionary = ["cats", "sandwiches", "hot chocolate"]

main :: IO ()
main = do entry <- getLine
          case getIndex entry dictionary of
               (Just pos) => putStrLn "Your entry is at position " ++ show
                    pos ++ " in the dictionary."
               Nothing => putStrLn "Your entry does not appear in the
                    dictionary."
```

Maybe examples

```
-- Find out where a value is in a function.
getIndex :: Eq a => a -> [a] -> Maybe Integer
getIndex = getIndexAcc 0

-- Helper function that remembers our position in the list.
getIndexAcc :: Eq a => Integer -> a -> [a] -> Maybe Integer
getIndexAcc pos value [] = Nothing
getIndexAcc pos value (x::xs) = if x == value
                                then Just pos
                                else getIndexAcc (pos+1) xs

-- A dictionary of all the important words.
dictionary :: [String]
dictionary = ["cats", "sandwiches", "hot chocolate"]

main :: IO ()
main = do entry <- getLine
          case getIndex entry dictionary of
               (Just pos) => putStrLn "Your entry is at position " ++ show
                    pos ++ " in the dictionary."
               Nothing => putStrLn "Your entry does not appear in the
                    dictionary."
```

- You know getIndex can yield a “null” value (Nothing). Just from type.

Maybe examples

```
-- Find out where a value is in a function.
getIndex :: Eq a => a -> [a] -> Maybe Integer
getIndex = getIndexAcc 0

-- Helper function that remembers our position in the list.
getIndexAcc :: Eq a => Integer -> a -> [a] -> Maybe Integer
getIndexAcc pos value [] = Nothing
getIndexAcc pos value (x::xs) = if x == value
                                then Just pos
                                else getIndexAcc (pos+1) xs

-- A dictionary of all the important words.
dictionary :: [String]
dictionary = ["cats", "sandwiches", "hot chocolate"]

main :: IO ()
main = do entry <- getLine
          case getIndex entry dictionary of
               (Just pos) => putStrLn "Your entry is at position " ++ show
                    pos ++ " in the dictionary."
               Nothing => putStrLn "Your entry does not appear in the
                    dictionary."
```

- You know getIndex can yield a “null” value (Nothing). Just from type.
- Could also be a Just <Integer>, such as Just 3.

Maybe examples

```
-- Find out where a value is in a function.
getIndex :: Eq a => a -> [a] -> Maybe Integer
getIndex = getIndexAcc 0

-- Helper function that remembers our position in the list.
getIndexAcc :: Eq a => Integer -> a -> [a] -> Maybe Integer
getIndexAcc pos value [] = Nothing
getIndexAcc pos value (x::xs) = if x == value
                                then Just pos
                                else getIndexAcc (pos+1) xs

-- A dictionary of all the important words.
dictionary :: [String]
dictionary = ["cats", "sandwiches", "hot chocolate"]

main :: IO ()
main = do entry <- getLine
          case getIndex entry dictionary of
               (Just pos) => putStrLn "Your entry is at position " ++ show
                    pos ++ " in the dictionary."
               Nothing => putStrLn "Your entry does not appear in the
                    dictionary."
```

- You know `getIndex` can yield a “null” value (Nothing). Just from type.
- Could also be a `Just <Integer>`, such as `Just 3`.
- You have to explicitly unwrap these values (see main) to get at the possible value!

Maybe more!

Seems tedious? It's not! Good syntax makes this easy!

```
-- Look up a word in the same position in a different dictionary.
dictionary :: [String]
dictionary = ["cats", "sandwiches", "hot chocolate"]


synonyms :: [String]
synonyms = ["meows", "bread oreos", "sweet nectar"]


moreSynonyms :: [String]
moreSynonyms = ["floofs", "subs", "hot coco"]


getIndex :: Integer -> [a] -> Maybe a
getIndex _ [] = Nothing
getIndex 0 (x:xs) = Just x
getIndex n (_:xs) = getIndex (n-1) xs
```

More on next slide...

Seems tedious? It's not! Good syntax makes this easy!

```
lookupSynonyms :: String -> Maybe (String, String)
lookupSynonyms word = do index <- getIndex word dictionary

                         -- Lookup my synonyms, if anything fails return Nothing
                              .
                         firstSynonym <- getIndex index synonyms
                         secondSynonym <- getIndex index moreSynonyms

                         -- Success! Return Just the synonyms.
                         Just (firstSynonym, secondSynonym)

-- lookupSynonyms essentially desugars to this.
-- The compiler can help avoid this tedium!
painfulLookupSynonyms :: String -> Maybe (String, String)
painfulLookupSynonyms word = case getIndex word dictionary of
                                  Nothing -> Nothing
                                  (Just index) ->
                                    case getIndex index synonyms of
                                         Nothing -> Nothing
                                         (Just first) ->
                                           case getIndex index moreSynonyms of
                                                Nothing -> Nothing
                                                (Just second) -> Just (first,
                                                     second)

main :: IO ()
main = do word <- getLine
          case lookupSynonym word of
            Nothing -> putStrLn ("Hmmm, I don't know a synonym for " ++ word)
            (Just synonym) -> putStrLn ("I think " ++ word ++ "'s are a lot like
                  " ++ synonym ++ "'s!")
```

Last words on Maybe

If you're a JavaScript programmer you've probably encountered promises. In a language like Haskell you could also have a promise type, which is similar to `Maybe`. Imagine having:

Last words on Maybe

If you're a JavaScript programmer you've probably encountered promises. In a language like Haskell you could also have a promise type, which is similar to `Maybe`. Imagine having:

- The type checker tell you when you forgot to "unwrap" a promise.

Last words on Maybe

If you're a JavaScript programmer you've probably encountered promises. In a language like Haskell you could also have a promise type, which is similar to `Maybe`. Imagine having:

- The type checker tell you when you forgot to "unwrap" a promise.
- Do notation which lets you string promises together with no syntactic overhead.

Last words on Maybe

If you're a JavaScript programmer you've probably encountered promises. In a language like Haskell you could also have a promise type, which is similar to `Maybe`. Imagine having:

- The type checker tell you when you forgot to "unwrap" a promise.
- Do notation which lets you string promises together with no syntactic overhead.
- Not having to write JavaScript ;)

Having a good type system in an expressive language, like Haskell, can really help ease a lot of the pain you currently suffer.

Last words on Maybe

If you're a JavaScript programmer you've probably encountered promises. In a language like Haskell you could also have a promise type, which is similar to `Maybe`. Imagine having:

- The type checker tell you when you forgot to "unwrap" a promise.
- Do notation which lets you string promises together with no syntactic overhead.
- Not having to write JavaScript ;)

Having a good type system in an expressive language, like Haskell, can really help ease a lot of the pain you currently suffer.

Programming can be good?

IO types

```
-- putStrLn :: IO ()
-- getLine :: IO String

main :: IO ()
main = do putStrLn "What is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name)
```

IO types

```haskell
-- putStrLn :: IO ()
-- getLine :: IO String

main :: IO ()
main = do putStrLn "What is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name)
```

- () is “void” — no return value.

IO types

```haskell
-- putStrLn :: IO ()
-- getLine :: IO String

main :: IO ()
main = do putStrLn "What is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name)
```

- `()` is "void" — no return value.
- `IO` means a function performs input / output.

IO types

```haskell
-- putStrLn :: IO ()
-- getLine :: IO String

main :: IO ()
main = do putStrLn "What is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name)
```

- () is “void” — no return value.
- IO means a function performs input / output.
  - Reads from disk, or stdin

IO types

```haskell
-- putStrLn :: IO ()
-- getLine :: IO String

main :: IO ()
main = do putStrLn "What is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name)
```

- `()` is "void" — no return value.
- `IO` means a function performs input / output.
  - Reads from disk, or stdin
  - Writes to disk, prints to screen

IO types

```haskell
-- putStrLn :: IO ()
-- getLine :: IO String

main :: IO ()
main = do putStrLn "What is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name)
```

- `()` is "void" — no return value.
- `IO` means a function performs input / output.
  - Reads from disk, or stdin
  - Writes to disk, prints to screen
  - etc...
- No escaping `IO`. Taints anything using it, so you know if something does input / output.

IO types

```haskell
-- putStrLn :: IO ()
-- getLine :: IO String

main :: IO ()
main = do putStrLn "What is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name)
```

- () is "void" — no return value.
- IO means a function performs input / output.
  - Reads from disk, or stdin
  - Writes to disk, prints to screen
  - etc...
- No escaping IO. Taints anything using it, so you know if something does input / output.
- Can help avoid unexpected behaviour, similar to global state changing a functions behaviour.

Haskell in summary: what does it buy us?

- We can catch errors at compile time!
  - Type system lets us describe values in a fair amount of detail, which removes a lot of obviously incorrect programs from the set of programs that compile.
  - Types don't contain nulls. Very few values which cause explosions at runtime.
- Easier to read and write programs. Types of functions are very descriptive!
  - Types help in much the same way as test driven development (but they're always there, unlike tests!)
    - Makes you think about arguments a function takes, and what it returns
  - Types point out errors when developing, such as forgetting to unwrap a Maybe value.

What more does it buy us?

- Types can be very general, allowing us to reuse functions with any type that makes sense.
  - `mySort` works with any list of orderable elements!
- It allows us to specify properties and guarantees within our programs.
  - "This function does not alter global state, or read from a file".
  - Functions are "pure".
  - Special actions, like IO, are clearly labeled.

Enter dependent types

There are some things that we just can't do with Haskell's types.

Enter dependent types

There are some things that we just can't do with Haskell's types.

Can write this:

```
index :: Integer -> [a] -> Maybe a
index 0 [] = Nothing
index 0 (x::xs) = Just x
index n (x::xs) = index (n-1) xs
```

Enter dependent types

There are some things that we just can't do with Haskell's types.

Can write this:

```
index :: Integer -> [a] -> Maybe a
index 0 [] = Nothing
index 0 (x::xs) = Just x
index n (x::xs) = index (n-1) xs
```

But can't just avoid calling an index function when the index is out of range:

```
-- Want the integer argument to always be in range so we don't need Maybe!
index :: Integer -> [a] -> a
index 0 [] = error "Uh... Whoops, walking off the end of the list!"
index 0 (x :: xs) = x
index n (x :: xs) = index (n-1) xs
```

Enter dependent types

There are some things that we just can't do with Haskell's types.

Can write this:

```
index :: Integer -> [a] -> Maybe a
index 0 [] = Nothing
index 0 (x::xs) = Just x
index n (x::xs) = index (n-1) xs
```

But can't just avoid calling an index function when the index is out of range:

```
-- Want the integer argument to always be in range so we don't need Maybe!
index :: Integer -> [a] -> a
index 0 [] = error "Uh... Whoops, walking off the end of the list!"
index 0 (x :: xs) = x
index n (x :: xs) = index (n-1) xs
```

- Need to encode length of the list into the type.
- Can't do this in Haskell because a type can not depend upon a value.
  - Length in the type must depend upon the length of the list value.

More motivation...

Also not possible to encode specific properties which depend upon values in types.

```
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]
```

More motivation...

Also not possible to encode specific properties which depend upon values in types.

```
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]
```

Would be nice to encode into the type of `mySort` that...

More motivation...

Also not possible to encode specific properties which depend upon values in types.

```
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]
```

Would be nice to encode into the type of `mySort` that...

- Output list must be in ascending order.

More motivation...

Also not possible to encode specific properties which depend upon values in types.

```
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]
```

Would be nice to encode into the type of `mySort` that...

- Output list must be in ascending order.
- Output list must contain the same values as the input list.

More motivation...

Also not possible to encode specific properties which depend upon values in types.

```
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]
```

Would be nice to encode into the type of `mySort` that...

- Output list must be in ascending order.
- Output list must contain the same values as the input list.

This would prove that the program works! `mySort` would be guaranteed to sort a list in ascending order if the program type checks!

More motivation...

Also not possible to encode specific properties which depend upon values in types.

```
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (first_elem::rest) = smaller ++ [first_elem] ++ larger
  where smaller = mySort [x | x <- rest, x <= first_elem]
        larger = mySort [x | x <- rest, x > first_elem]
```

Would be nice to encode into the type of `mySort` that...

- Output list must be in ascending order.
- Output list must contain the same values as the input list.

This would prove that the program works! `mySort` would be guaranteed to sort a list in ascending order if the program type checks!

We can do this kind of thing with dependent types. We'll look at some basic examples in Idris, a programming language like Haskell, but with dependent types.

Dependent types in Idris

Vectors are a classic example of dependent types!

Dependent types in Idris

Vectors are a classic example of dependent types!

- Like lists, but...

Dependent types in Idris

Vectors are a classic example of dependent types!

- Like lists, but...
- They include the length of the list in the type.

```
two_little_piggies : Vect 2 String
two_little_piggies = ["Oinkers", "Snorkins"]

-- This would be a type error, caught at compilation:
three_little_piggies : Vect 3 String
three_little_piggies = two_little_piggies
```

Append and the type level computations

Computations at the type level allow us to make some more complicated, generalized functions.

Append and the type level computations

Computations at the type level allow us to make some more complicated, generalized functions.

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
```

Append and the type level computations

Computations at the type level allow us to make some more complicated, generalized functions.

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
```

`n`, `m`, and `(n + m)` are all natural numbers, and `elem` is any type.

Append and the type level computations

Computations at the type level allow us to make some more complicated, generalized functions.

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
```

`n`, `m`, and `(n + m)` are all natural numbers, and `elem` is any type.

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 a
  (::) : (x : a) -> Vect k a -> Vect (S k) a
```

Append and the type level computations

Computations at the type level allow us to make some more complicated, generalized functions.

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
```

`n`, `m`, and `(n + m)` are all natural numbers, and `elem` is any type.

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 a
  (::) : (x : a) -> Vect k a -> Vect (S k) a
```

- The full `Vect` type constructed from natural number value for length, and a type for the elements.

Append and the type level computations

Computations at the type level allow us to make some more complicated, generalized functions.

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
```

`n`, `m`, and `(n + m)` are all natural numbers, and `elem` is any type.

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 a
  (::) : (x : a) -> Vect k a -> Vect (S k) a
```

- The full `Vect` type constructed from natural number value for length, and a type for the elements.
- Two constructors define the type recursively (called an inductive type – we'll see why later).
  - One for the empty vector.
  - Single value concatenated to another vector to make a vector with 1 more element. S is successor of natural numbers, $+1$.

Types and automation

Idris can help us generate programs based on the types. (All the steps you will see are done automatically by Idris).

Types and automation

Idris can help us generate programs based on the types. (All the steps you will see are done automatically by Idris).

We can ask idris to start our function definition based on the type:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append xs ys = ?append_rhs
```

Types and automation

Idris can help us generate programs based on the types. (All the steps you will see are done automatically by Idris).

We can ask idris to start our function definition based on the type:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append xs ys = ?append_rhs
```

`?append_rhs` is a hole. It's a stand in for a value we need to provide. Idris can tell us the type of a hole, and potentially fill it in for us. It also tells us types of what's in scope for the hole. This looks like this:

Types and automation

Idris can help us generate programs based on the types. (All the steps you will see are done automatically by Idris).

We can ask idris to start our function definition based on the type:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append xs ys = ?append_rhs
```

?append_rhs is a hole. It's a stand in for a value we need to provide. Idris can tell us the type of a hole, and potentially fill it in for us. It also tells us types of what's in scope for the hole. This looks like this:

```
- + Main.append_rhs [P]
 `--              elem : Type
                     m : Nat
                    ys : Vect m elem
                     n : Nat
                    xs : Vect n elem
     ----------------------------------------
      Main.append_rhs : Vect (n + m) elem
```

Types and automation

We can get Idris to do case split on the first argument...

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append xs ys = ?append_rhs
```

Types and automation

We can get Idris to do case split on the first argument...

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append xs ys = ?append_rhs
```

Which leads to a pattern match on constructors, and two holes:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ?append_rhs_1
append (x :: xs) ys = ?append_rhs_2
```

Types and automation

We can get Idris to do case split on the first argument...

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append xs ys = ?append_rhs
```

Which leads to a pattern match on constructors, and two holes:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ?append_rhs_1
append (x :: xs) ys = ?append_rhs_2
```

```
- + Main.append_rhs_1 [P]
 `--               elem : Type
                      m : Nat
                     ys : Vect m elem
     ------------------------------------------
      Main.append_rhs_1 : Vect (0 + m) elem

- + Main.append_rhs_2 [P]
 `--               elem : Type
                      x : elem
                      m : Nat
                     ys : Vect m elem
                    len : Nat
                     xs : Vect len elem
     ------------------------------------------------
      Main.append_rhs_2 : Vect ((S len) + m) elem
```

Types and automation

Once broken into cases, Idris can search for values which satisfy the types of the holes. Let's look at the first one...

Types and automation

Once broken into cases, Idris can search for values which satisfy the types of the holes. Let's look at the first one...

```
    append : Vect n elem -> Vect m elem -> Vect (n + m) elem
    append [] ys = ?append_rhs_1
```

```
- + Main.append_rhs_1 [P]
 `--               elem : Type
                      m : Nat
                     ys : Vect m elem
     ------------------------------------------
      Main.append_rhs_1 : Vect (0 + m) elem
```

Types and automation

Once broken into cases, Idris can search for values which satisfy the types of the holes. Let's look at the first one...

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ?append_rhs_1
```

```
- + Main.append_rhs_1 [P]
 `--               elem : Type
                      m : Nat
                     ys : Vect m elem
     -----------------------------------------
      Main.append_rhs_1 : Vect (0 + m) elem
```

Idris actually evaluates the type `Vect (0 + m) elem`, so this is really...

Types and automation

Once broken into cases, Idris can search for values which satisfy the types of the holes. Let's look at the first one...

```
    append : Vect n elem -> Vect m elem -> Vect (n + m) elem
    append [] ys = ?append_rhs_1
```

```
- + Main.append_rhs_1 [P]
 `--                elem : Type
                       m : Nat
                      ys : Vect m elem
     ------------------------------------------
      Main.append_rhs_1 : Vect (0 + m) elem
```

Idris actually evaluates the type `Vect (0 + m) elem`, so this is really...

```
- + Main.append_rhs_1 [P]
 `--                elem : Type
                       m : Nat
                      ys : Vect m elem
     ------------------------------------------
      Main.append_rhs_1 : Vect m elem
```

Types and automation

Once broken into cases, Idris can search for values which satisfy the types of the holes. Let's look at the first one...

```
    append : Vect n elem -> Vect m elem -> Vect (n + m) elem
    append [] ys = ?append_rhs_1
```

```
- + Main.append_rhs_1 [P]
 `--              elem : Type
                     m : Nat
                    ys : Vect m elem
     ------------------------------------------
      Main.append_rhs_1 : Vect (0 + m) elem
```

Idris actually evaluates the type `Vect (0 + m) elem`, so this is really...

```
- + Main.append_rhs_1 [P]
 `--              elem : Type
                     m : Nat
                    ys : Vect m elem
     ------------------------------------------
      Main.append_rhs_1 : Vect m elem
```

Only `ys` satisfies this type. Remember `m` could be any natural.

So, Idris can tell that it can use `ys` for this case, giving us:

So, Idris can tell that it can use ys for this case, giving us:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = ?append_rhs_2
```

So, Idris can tell that it can use ys for this case, giving us:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = ?append_rhs_2
```

The second hole is a bit more interesting:

```
- + Main.append_rhs_2 [P]
 `--                  elem : Type
                         x : elem
                         m : Nat
                        ys : Vect m elem
                         k : Nat
                        xs : Vect k elem
     ----------------------------------------------
      Main.append_rhs_2 : Vect ((S k) + m) elem
```

So, Idris can tell that it can use ys for this case, giving us:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = ?append_rhs_2
```

The second hole is a bit more interesting:

```
- + Main.append_rhs_2 [P]
 `--                 elem : Type
                        x : elem
                        m : Nat
                       ys : Vect m elem
                        k : Nat
                       xs : Vect k elem
     ----------------------------------------------
      Main.append_rhs_2 : Vect ((S k) + m) elem
```

... Idris can also fill this in.

So, Idris can tell that it can use ys for this case, giving us:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = ?append_rhs_2
```

The second hole is a bit more interesting:

```
- + Main.append_rhs_2 [P]
 `--                elem : Type
                       x : elem
                       m : Nat
                      ys : Vect m elem
                       k : Nat
                      xs : Vect k elem
     ----------------------------------------------
      Main.append_rhs_2 : Vect ((S k) + m) elem
```

... Idris can also fill this in.

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

So, Idris can tell that it can use ys for this case, giving us:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = ?append_rhs_2
```

The second hole is a bit more interesting:

```
- + Main.append_rhs_2 [P]
 `--              elem : Type
                     x : elem
                     m : Nat
                    ys : Vect m elem
                     k : Nat
                    xs : Vect k elem
     ----------------------------------------------
      Main.append_rhs_2 : Vect ((S k) + m) elem
```

... Idris can also fill this in.

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

This seems bonkers, so let's look at how Idris did this.

Idris the smart and brave

To get `append (x :: xs) ys = x :: append xs ys` Idris realized a couple things:

```
data Nat : Type where
  0 : Nat -- Zero
  S : Nat -> Nat -- Successor (+1)


(+) : Nat -> Nat -> Nat
(+) 0 m = m
(+) (S k) m = S (k + m)
```

Idris the smart and brave

To get `append (x :: xs) ys = x :: append xs ys` Idris realized a couple things:

```
data Nat : Type where
  0 : Nat -- Zero
  S : Nat -> Nat -- Successor (+1)


(+) : Nat -> Nat -> Nat
(+) 0 m = m
(+) (S k) m = S (k + m)
```

Remember our hole:

```
Main.append_rhs_2 : Vect (( S k ) + m ) elem
```

Idris the smart and brave

To get `append (x :: xs) ys = x :: append xs ys` Idris realized a couple things:

```
data Nat : Type where
  0 : Nat -- Zero
  S : Nat -> Nat -- Successor (+1)


(+) : Nat -> Nat -> Nat
(+) 0 m = m
(+) (S k) m = S (k + m)
```

Remember our hole:

```
Main.append_rhs_2 : Vect (( S k ) + m ) elem
```

By definition `(S k) + m = S (k + m)`, so we have:

Idris the smart and brave

To get `append (x :: xs) ys = x :: append xs ys` Idris realized a couple things:

```
data Nat : Type where
  0 : Nat -- Zero
  S : Nat -> Nat -- Successor (+1)


(+) : Nat -> Nat -> Nat
(+) 0 m = m
(+) (S k) m = S (k + m)
```

Remember our hole:

```
Main.append_rhs_2 : Vect (( S k ) + m ) elem
```

By definition `(S k) + m = S (k + m)`, so we have:

```
Main.append_rhs_2 : Vect (S (k + m)) elem
```

Idris the smart and brave

So we really have this goal:

```
Main.append_rhs_2 : Vect (S (k + m)) elem
```

Idris the smart and brave

So we really have this goal:

```
Main.append_rhs_2 : Vect (S (k + m)) elem
```

Idris looks at how to construct a `Vect (S blah) elem`...

Idris the smart and brave

So we really have this goal:

```
Main.append_rhs_2 : Vect (S (k + m)) elem
```

Idris looks at how to construct a `Vect (S blah) elem`...

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 a
  (::) : (x : a) -> Vect k a -> Vect (S k) a
```

Idris the smart and brave

So we really have this goal:

```
Main.append_rhs_2 : Vect (S (k + m)) elem
```

Idris looks at how to construct a `Vect (S blah) elem`...

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 a
  (::) : (x : a) -> Vect k a -> Vect (S k) a
```

To construct a `Vect (S blah) elem` we need `::`!

Idris the smart and brave

So we really have this goal:

```
Main.append_rhs_2 : Vect (S (k + m)) elem
```

Idris looks at how to construct a `Vect (S blah) elem`...

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 a
  (::) : (x : a) -> Vect k a -> Vect (S k) a
```

To construct a `Vect (S blah) elem` we need `::`!

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = ?elem_to_concat :: ?rest_of_vect
```

Idris the smart and brave

So we really have this goal:

```
Main.append_rhs_2 : Vect (S (k + m)) elem
```

Idris looks at how to construct a `Vect (S blah) elem`...

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 a
  (::) : (x : a) -> Vect k a -> Vect (S k) a
```

To construct a `Vect (S blah) elem` we need `::`!

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = ?elem_to_concat :: ?rest_of_vect
```

These are the holes:

```
Main.elem_to_concat : elem

Main.rest_of_vect : Vect (plus k m) elem
```

Idris the smart and brave

The first hole is easy for Idris.

```
- + Main.elem_to_concat [P]
 `--                  elem : Type
                         x : elem
                         m : Nat
                        ys : Vect m elem
                         k : Nat
                        xs : Vect k elem
     -----------------------------------
      Main.elem_to_concat : elem
```

Idris the smart and brave

The first hole is easy for Idris.

```
- + Main.elem_to_concat [P]
 `--                  elem : Type
                         x : elem
                         m : Nat
                        ys : Vect m elem
                         k : Nat
                        xs : Vect k elem
     -----------------------------------
      Main.elem_to_concat : elem
```

We need something with the arbitrary type `elem`. Only `x` fits!

Idris the smart and brave

The first hole is easy for Idris.

```
- + Main.elem_to_concat [P]
 `--                  elem : Type
                         x : elem
                         m : Nat
                        ys : Vect m elem
                         k : Nat
                        xs : Vect k elem
     -----------------------------------
      Main.elem_to_concat : elem
```

We need something with the arbitrary type `elem`. Only `x` fits!

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = x :: ?rest_of_vect
```

Idris the smart and brave

The first hole is easy for Idris.

```
- + Main.elem_to_concat [P]
 `--                  elem : Type
                         x : elem
                         m : Nat
                        ys : Vect m elem
                         k : Nat
                        xs : Vect k elem
     -----------------------------------
      Main.elem_to_concat : elem
```

We need something with the arbitrary type `elem`. Only `x` fits!

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = x :: ?rest_of_vect
```

Second hole involves a bit more work...

Daily dose of recursion

```
  - + Main.rest_of_vect [P]
   `--               elem : Type
                        x : elem
                        m : Nat
                       ys : Vect m elem
                        k : Nat
                       xs : Vect k elem
     ------------------------------------------
      Main.rest_of_vect : Vect (k + m) elem
```

Daily dose of recursion

```
 - + Main.rest_of_vect [P]
  `--                  elem : Type
                          x : elem
                          m : Nat
                         ys : Vect m elem
                          k : Nat
                         xs : Vect k elem
     ------------------------------------------
      Main.rest_of_vect : Vect (k + m) elem
```

Idris knows it can call append recursively with a `Vect k elem` and a `Vect m elem` to get a `Vect (k + m) elem`.

Daily dose of recursion

```
    - + Main.rest_of_vect [P]
     `--               elem : Type
                          x : elem
                          m : Nat
                         ys : Vect m elem
                          k : Nat
                         xs : Vect k elem
         ------------------------------------------
          Main.rest_of_vect : Vect (k + m) elem
```

Idris knows it can call append recursively with a `Vect k elem` and a `Vect m elem` to get a `Vect (k + m) elem`.

Looking at our goal we have such vectors, `ys` and `xs`.

Daily dose of recursion

```
- + Main.rest_of_vect [P]
 `--              elem : Type
                     x : elem
                     m : Nat
                    ys : Vect m elem
                     k : Nat
                    xs : Vect k elem
     ------------------------------------------
      Main.rest_of_vect : Vect (k + m) elem
```

Idris knows it can call append recursively with a `Vect k elem` and a `Vect m elem` to get a `Vect (k + m) elem`.

Looking at our goal we have such vectors, `ys` and `xs`.

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

Daily dose of recursion

```
  - + Main.rest_of_vect [P]
   `--                elem : Type
                         x : elem
                         m : Nat
                        ys : Vect m elem
                         k : Nat
                        xs : Vect k elem
       ------------------------------------------
        Main.rest_of_vect : Vect (k + m) elem
```

Idris knows it can call append recursively with a `Vect k elem` and a `Vect m elem` to get a `Vect (k + m) elem`.

Looking at our goal we have such vectors, `ys` and `xs`.

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

Idris wrote this function automatically based on a small spec!

Daily dose of recursion

```
- + Main.rest_of_vect [P]
 `--                  elem : Type
                         x : elem
                         m : Nat
                        ys : Vect m elem
                         k : Nat
                        xs : Vect k elem
     ------------------------------------------
      Main.rest_of_vect : Vect (k + m) elem
```

Idris knows it can call append recursively with a `Vect k elem` and a `Vect m elem` to get a `Vect (k + m) elem`.

Looking at our goal we have such vectors, `ys` and `xs`.

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

Idris wrote this function automatically based on a small spec!
It's cool that Idris even gets the order correct because of the types!

Less precise types

Doesn't work well with the less precise list type!

Less precise types

Doesn't work well with the less precise list type!

```
append : List elem -> List elem -> List elem
append [] ys = ?append_rhs1
append (x :: xs) ys = ?append_rhs2
```

What do you think will happen when we try the same thing?

Less precise types

Doesn't work well with the less precise list type!

```
append : List elem -> List elem -> List elem
append [] ys = ?append_rhs1
append (x :: xs) ys = ?append_rhs2
```

What do you think will happen when we try the same thing?

```
append : List elem -> List elem -> List elem
append [] ys = []
append (x :: xs) ys = []
```

Huh... I guess it doesn't realize that append should make a list as long as the inputs combined!

Less precise types

Doesn't work well with the less precise list type!

```
append : List elem -> List elem -> List elem
append [] ys = ?append_rhs1
append (x :: xs) ys = ?append_rhs2
```

What do you think will happen when we try the same thing?

```
append : List elem -> List elem -> List elem
append [] ys = []
append (x :: xs) ys = []
```

Huh... I guess it doesn't realize that append should make a list as long as the inputs combined!

Idris just finds the first possible function. Length isn't encoded in the type to enforce length of output.

A careful index

In Haskell we couldn't guarantee that an index was in range of a list...

A careful index

In Haskell we couldn't guarantee that an index was in range of a list... In Idris we can!

A careful index

In Haskell we couldn't guarantee that an index was in range of a list... In Idris we can!

```
index : Fin len -> Vect len elem -> elem
index FZ (x :: xs) = x
index (FS n) (_ :: xs) = myIndex n xs
```

`Fin len` is a type for natural numbers strictly less than `len`

A careful index

In Haskell we couldn't guarantee that an index was in range of a list... In Idris we can!

```
index : Fin len -> Vect len elem -> elem
index FZ (x :: xs) = x
index (FS n) (_ :: xs) = myIndex n xs
```

`Fin len` is a type for natural numbers strictly less than `len`

```
data Fin : Nat -> Type where
  FZ : Fin (S k)
  FS : Fin k -> Fin (S k)
```

A careful index

In Haskell we couldn't guarantee that an index was in range of a list... In Idris we can!

```
index : Fin len -> Vect len elem -> elem
index FZ (x :: xs) = x
index (FS n) (_ :: xs) = myIndex n xs
```

`Fin len` is a type for natural numbers strictly less than `len`

```
data Fin : Nat -> Type where
  FZ : Fin (S k)
  FS : Fin k -> Fin (S k)
```

A tale of cautious indexing

```
cats : Vect 2 String
cats = ["The Panther", "Smoke Smoke"]

-- "The Panther" : String
index 0 cats -- This type checks.


-- (input):1:9:When checking argument prf to function Data.Fin.fromInteger:
--          When using 2 as a literal for a Fin 2
--                  2 is not strictly less than 2
index 2 cats -- This is out of bounds, so the program won't even compile!
```

A tale of cautious indexing

```
cats : Vect 2 String
cats = ["The Panther", "Smoke Smoke"]

-- "The Panther" : String
index 0 cats -- This type checks.


-- (input):1:9:When checking argument prf to function Data.Fin.fromInteger:
--         When using 2 as a literal for a Fin 2
--                 2 is not strictly less than 2
index 2 cats -- This is out of bounds, so the program won't even compile!
```

Lots of cool guarantees that we can make with dependent types!

Logic primer

What are propositions?

Logic primer

What are propositions?

- A statement: "the sky is blue", "$2 + 2$ is 4"

Logic primer

What are propositions?

- A statement: "the sky is blue", "$2 + 2$ is 4"
- Not necessarily true: "$2 + 2$ is 27"

Logic primer

What are propositions?

- A statement: "the sky is blue", "$2 + 2$ is 4"
- Not necessarily true: "$2 + 2$ is 27"

Logical proofs are used to determine the validity of a proposition.

Logic primer

What are propositions?

- A statement: "the sky is blue", "$2 + 2$ is 4"
- Not necessarily true: "$2 + 2$ is 27"

Logical proofs are used to determine the validity of a proposition.

We could show that "$2 + 2$ is 27" is false with a logical proof.

Variables

Propositions are often represented by variables, for instance:

$$p$$

Variables

Propositions are often represented by variables, for instance:

$$p$$

p is a proposition. It could be anything, really...

Variables

Propositions are often represented by variables, for instance:

$$p$$

p is a proposition. It could be anything, really...

$$p = \text{“ducks are fantastic”}$$

Variables

Propositions are often represented by variables, for instance:

$$p$$

p is a proposition. It could be anything, really...

$$p = \text{“ducks are fantastic''}$$

And I might have another proposition:

$$q = \text{“ducks are truly the worst''}$$

Variables

Propositions are often represented by variables, for instance:

$$p$$

p is a proposition. It could be anything, really...

$$p = \text{"ducks are fantastic"}$$

And I might have another proposition:

$$q = \text{"ducks are truly the worst"}$$

We can build up more complicated propositions with logical connectives. In this case we might have:

$$p \to \neg q$$

Which means that if p is true, then q is not true. We'll see more of this shortly.

More logic

We've mostly been using plain English to convey these propositions, but often they'll be more mathematical statements, such as:

$$\forall n \in \mathrm{N}, \exists m \in \mathrm{N} \text{ such that } m > n$$

More logic

We've mostly been using plain English to convey these propositions, but often they'll be more mathematical statements, such as:

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N} \text{ such that } m > n$$

Propositions...

- Are built up from a set of axioms
  - Just rules which describe your mathematical objects

More logic

We've mostly been using plain English to convey these propositions, but often they'll be more mathematical statements, such as:

$$\forall n \in \mathrm{N}, \exists m \in \mathrm{N} \text{ such that } m > n$$

Propositions...

- Are built up from a set of axioms
  - Just rules which describe your mathematical objects
- Can be combined with logical connectives.

More logic

We've mostly been using plain English to convey these propositions, but often they'll be more mathematical statements, such as:

$$\forall n \in \mathrm{N}, \exists m \in \mathrm{N} \text{ such that } m > n$$

Propositions...

- Are built up from a set of axioms
  - Just rules which describe your mathematical objects
- Can be combined with logical connectives.

Logic is a sort of metalanguage which describes how you can make judgements about your mathematical objects.

Logical connectives

In a logical system you might have these logical connectives:

Logical connectives

In a logical system you might have these logical connectives:

- Implication:
  - $p \to q$, meaning "if p is true, then q must be true."

Logical connectives

In a logical system you might have these logical connectives:

- Implication:
  - $p \to q$, meaning "if p is true, then q must be true."
- Conjunction:
  - $p \land q$, meaning "both p and q are true."

Logical connectives

In a logical system you might have these logical connectives:

- Implication:
  - $p \rightarrow q$, meaning "if p is true, then q must be true."
- Conjunction:
  - $p \wedge q$, meaning "both p and q are true."
- Disjunction:
  - $p \vee q$, meaning "at least one of p or q is true."

Logical connectives

In a logical system you might have these logical connectives:

- Implication:
  - $p \to q$, meaning "if p is true, then q must be true."
- Conjunction:
  - $p \wedge q$, meaning "both p and q are true."
- Disjunction:
  - $p \vee q$, meaning "at least one of p or q is true."
- Negation:
  - $\neg p$, meaning "not p", "p is false."

Quantification

You might also have quantifiers:

Quantification

You might also have quantifiers:

- Universal quantification:
  - $\forall x \in S, p(x)$, meaning "for every x (in S), $p(x)$ is true."

Quantification

You might also have quantifiers:

- Universal quantification:
  - $\forall x \in S, p(x)$, meaning "for every x (in S), $p(x)$ is true."
- Existential quantification:
  - $\exists x \in S, p(x)$, meaning "there's at least on x (in S), which makes $p(x)$ true."

Proof rules

You also have some rules for how you can combine these things to form proofs. E.g., Modus ponens

Curry-Howard Isomorphism: Propositions as Types!

Things get interesting when you start thinking about types as propositions...

Curry-Howard Isomorphism: Propositions as Types!

Things get interesting when you start thinking about types as propositions...

$$p \rightarrow q$$

Curry-Howard Isomorphism: Propositions as Types!

Things get interesting when you start thinking about types as propositions...

$$p \rightarrow q$$

This looks an awful lot like...

```
hmmm1 : p -> q
```

Curry-Howard Isomorphism: Propositions as Types!

Things get interesting when you start thinking about types as propositions...

$$p \to q$$

This looks an awful lot like...

```
hmmm1 : p -> q
```

Similarly...

$$p \wedge q$$

Curry-Howard Isomorphism: Propositions as Types!

Things get interesting when you start thinking about types as propositions...

$$p \rightarrow q$$

This looks an awful lot like...

```
hmmm1 : p -> q
```

Similarly...

$$p \wedge q$$

Is kind of similar to:

```
hmmm2 : (p, q)
```

Curry-Howard Isomorphism: Propositions as Types!

Conjunction elimination corresponds to destructing a product...

```
-- P /\ Q -> P
fst : (p, q) -> p
fst (a, b) = a

-- P /\ Q -> Q
snd : (p, q) -> q
snd (a, b) = b
```

Curry-Howard Isomorphism: Propositions as Types!

Conjunction elimination corresponds to destructing a product...

```
-- P /\ Q -> P
fst : (p, q) -> p
fst (a, b) = a

-- P /\ Q -> Q
snd : (p, q) -> q
snd (a, b) = b
```

Conjuction introduction corresponds to constructing a product...

```
-- P -> Q -> (P, Q)
and : p -> q -> (p, q)
and a b = (a, b)
```

Curry-Howard Isomorphism: Propositions as Types!

Conjunction elimination corresponds to destructing a product...

```
-- P /\ Q -> P
fst : (p, q) -> p
fst (a, b) = a

-- P /\ Q -> Q
snd : (p, q) -> q
snd (a, b) = b
```

Conjuction introduction corresponds to constructing a product...

```
-- P -> Q -> (P, Q)
and : p -> q -> (p, q)
and a b = (a, b)
```

Similar things for other logical connectives. E.g., $p \vee q$ corresponds to a sum type `Either p q`.

Curry-Howard Isomorphism: Propositions as Types!

Conjunction elimination corresponds to destructing a product...

```
-- P /\ Q -> P
fst : (p, q) -> p
fst (a, b) = a

-- P /\ Q -> Q
snd : (p, q) -> q
snd (a, b) = b
```

Conjuction introduction corresponds to constructing a product...

```
-- P -> Q -> (P, Q)
and : p -> q -> (p, q)
and a b = (a, b)
```

Similar things for other logical connectives. E.g., $p \vee q$ corresponds to a sum type `Either p q`.

$\neg p$ corresponds to `p -> Void` where `Void` is an uninhabited type.

Curry-Howard Isomorphism: Propositions as Types!

Conjunction elimination corresponds to destructing a product...

```
-- P /\ Q -> P
fst : (p, q) -> p
fst (a, b) = a

-- P /\ Q -> Q
snd : (p, q) -> q
snd (a, b) = b
```

Conjuction introduction corresponds to constructing a product...

```
-- P -> Q -> (P, Q)
and : p -> q -> (p, q)
and a b = (a, b)
```

Similar things for other logical connectives. E.g., $p \vee q$ corresponds to a sum type `Either p q`.

$\neg p$ corresponds to `p -> Void` where `Void` is an uninhabited type.

Dependent types are needed for quantifiers.

Valuable proofs

If types are propositions, then what are the vaues?

Valuable proofs

If types are propositions, then what are the vaues?

Well, they're a sort of "existence proof" of a proposition.

Valuable proofs

If types are propositions, then what are the vaues?

Well, they're a sort of "existence proof" of a proposition.

```
const : p -> q -> p
const a b = a
```

Valuable proofs

If types are propositions, then what are the vaues?

Well, they're a sort of "existence proof" of a proposition.

```
const : p -> q -> p
const a b = a
```

The value `const` can be seen as a proof of the proposition (type) `p -> q -> p`... This is what `const` says:

Valuable proofs

If types are propositions, then what are the vaues?

Well, they're a sort of "existence proof" of a proposition.

```
const : p -> q -> p
const a b = a
```

The value `const` can be seen as a proof of the proposition (type) `p -> q -> p`... This is what `const` says:

- given a proof of `p`, a

Valuable proofs

If types are propositions, then what are the vaues?

Well, they're a sort of "existence proof" of a proposition.

```
const : p -> q -> p
const a b = a
```

The value `const` can be seen as a proof of the proposition (type) `p -> q -> p`... This is what `const` says:

- given a proof of `p`, `a`
- and a proof of `q`, `b`

Valuable proofs

If types are propositions, then what are the vaues?

Well, they're a sort of "existence proof" of a proposition.

```
const : p -> q -> p
const a b = a
```

The value `const` can be seen as a proof of the proposition (type) `p -> q -> p`... This is what `const` says:

- given a proof of `p`, `a`
- and a proof of `q`, `b`
- I can provide `a` as a proof of `p`

Valuable proofs

If types are propositions, then what are the vaues?

Well, they're a sort of "existence proof" of a proposition.

```
const : p -> q -> p
const a b = a
```

The value `const` can be seen as a proof of the proposition (type) `p -> q -> p`... This is what `const` says:

- given a proof of `p`, `a`
- and a proof of `q`, `b`
- I can provide `a` as a proof of `p`

Which oddly enough makes a lot of sense as proof!

Invaluable proofs

Type checker can prevent bogus proofs! Stops us from proving false propositions!

Invaluable proofs

Type checker can prevent bogus proofs! Stops us from proving false propositions!

```
bogus : p -> q
bogus p = -- What can I put here that would type check? :(
```

Invaluable proofs

Type checker can prevent bogus proofs! Stops us from proving false propositions!

```
bogus : p -> q
bogus p = -- What can I put here that would type check? :(
```

Can't find a value of type `q`, since we only have a value of type `p`!

Proofs in practice

Idris has a type for equality between two things.

Proofs in practice

Idris has a type for equality between two things.

```
equality_good : 2+3 = 5 -- Equality as a proposition in the type!
equality_good = Refl

-- This fails to type check
equality_bad : 2+3 = 7
equality_bad = Refl
```

Proofs in practice

Idris has a type for equality between two things.

```
equality_good : 2+3 = 5 -- Equality as a proposition in the type!
equality_good = Refl

-- This fails to type check
equality_bad : 2+3 = 7
equality_bad = Refl
```

Equality has only one constructor, `Refl`. This is roughly defined as:

Proofs in practice

Idris has a type for equality between two things.

```
equality_good : 2+3 = 5 -- Equality as a proposition in the type!
equality_good = Refl

-- This fails to type check
equality_bad : 2+3 = 7
equality_bad = Refl
```

Equality has only one constructor, `Refl`. This is roughly defined as:

```
data (=) : a -> b -> Type where
  Refl : x = x
```

Proofs in practice

Idris has a type for equality between two things.

```
equality_good : 2+3 = 5 -- Equality as a proposition in the type!
equality_good = Refl

-- This fails to type check
equality_bad : 2+3 = 7
equality_bad = Refl
```

Equality has only one constructor, `Refl`. This is roughly defined as:

```
data (=) : a -> b -> Type where
  Refl : x = x
```

Looks a little obtuse... But if we need a `something = blah` type, we use `Refl`.

Proofs in practice

Idris has a type for equality between two things.

```
equality_good : 2+3 = 5 -- Equality as a proposition in the type!
equality_good = Refl

-- This fails to type check
equality_bad : 2+3 = 7
equality_bad = Refl
```

Equality has only one constructor, `Refl`. This is roughly defined as:

```
data (=) : a -> b -> Type where
  Refl : x = x
```

Looks a little obtuse... But if we need a `something = blah` type, we use `Refl`.

Idris will try to determine if they are equal from the definitions it knows about.

Proofs in practice

Idris has a type for equality between two things.

```
equality_good : 2+3 = 5 -- Equality as a proposition in the type!
equality_good = Refl

-- This fails to type check
equality_bad : 2+3 = 7
equality_bad = Refl
```

Equality has only one constructor, `Refl`. This is roughly defined as:

```
data (=) : a -> b -> Type where
  Refl : x = x
```

Looks a little obtuse... But if we need a `something = blah` type, we use `Refl`.

Idris will try to determine if they are equal from the definitions it knows about.
E.g., 2+3 = 5, since Idris can evaluate 2+3 to 5, and see that they are identical.

Congruency

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f prf = ?cong_rhs
```

Congruency

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f prf = ?cong_rhs
```

Seems a bit scary! Equality in the types!

Congruency

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f prf = ?cong_rhs
```

Seems a bit scary! Equality in the types!

Remember that this just means we need to construct a type using `Refl`. Idris just needs to show that the left and right hand side are equal.

Congruency

This is our goal:

```
  - + Main.cong_rhs [P]
 `--              b : Type
                  a : Type
                  x : a
                  f : a -> b
                  y : a
                prf : x = y
     ---------------------------
      Main.cong_rhs : f x = f y
```

Congruency

This is our goal:

```
 - + Main.cong_rhs [P]
 `__              b : Type
                  a : Type
                  x : a
                  f : a -> b
                  y : a
                prf : x = y
     ---------------------------
      Main.cong_rhs : f x = f y
```

We can get some help by pattern matching!

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f Refl = ?cong_rhs_1
```

Congruency

This is our goal:

```
  - + Main.cong_rhs [P]
 `--               b : Type
                   a : Type
                   x : a
                   f : a -> b
                   y : a
                 prf : x = y
     ---------------------------
      Main.cong_rhs : f x = f y
```

We can get some help by pattern matching!

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f Refl = ?cong_rhs_1
```

Looks unimpressive, but it changed our goal:

```
 - + Main.cong_rhs_1 [P]
  `--                b : Type
                     a : Type
                     x : a
                     f : a -> b
     -----------------------------
      Main.cong_rhs_1 : f x = f x
```

Congruency

`Refl` lets us construct equalities.

```
Refl : x = x

-- So, if we just replace the general "x" above with our "f x" we
-- would get...
Refl : f x = f x
```

Congruency

`Refl` lets us construct equalities.

```
Refl : x = x

-- So, if we just replace the general "x" above with our "f x" we
-- would get...
Refl : f x = f x
```

`Refl` uses implicit arguments, can infer from context:

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f Refl = Refl
```

Congruency

`Refl` lets us construct equalities.

```
Refl : x = x

-- So, if we just replace the general "x" above with our "f x" we
-- would get...
Refl : f x = f x
```

`Refl` uses implicit arguments, can infer from context:

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f Refl = Refl
```

Can also provide the argument explicitly:

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f (Refl {x}) = Refl {x = f x}
```

Congruency

`Refl` lets us construct equalities.

```
Refl : x = x

-- So, if we just replace the general "x" above with our "f x" we
-- would get...
Refl : f x = f x
```

`Refl` uses implicit arguments, can infer from context:

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f Refl = Refl
```

Can also provide the argument explicitly:

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f (Refl {x}) = Refl {x = f x}
```

Bit confusing because `x` is in both places, but `x` in the definition of `Refl` is in a different scope, and we substitute `f x` for `x`.

Congruency

`Refl` lets us construct equalities.

```
Refl : x = x

-- So, if we just replace the general "x" above with our "f x" we
-- would get...
Refl : f x = f x
```

`Refl` uses implicit arguments, can infer from context:

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f Refl = Refl
```

Can also provide the argument explicitly:

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f (Refl {x}) = Refl {x = f x}
```

Bit confusing because `x` is in both places, but `x` in the definition of `Refl` is in a different scope, and we substitute `f x` for `x`.

This concludes the proof of `cong`!

Slightly more complicated proofs

Let's prove the associativity of addition on natural numbers!

Slightly more complicated proofs

Let's prove the associativity of addition on natural numbers!

Here's our nice unary representation of natural numbers:

```
data Nat : Type where
  0 : Nat
  S : Nat -> Nat -- Successor, +1

-- 0 = 0
-- S 0 = 1
-- S (S 0) = 2
-- etc...
```

Slightly more complicated proofs

Let's prove the associativity of addition on natural numbers!

Here's our nice unary representation of natural numbers:

```
data Nat : Type where
  0 : Nat
  S : Nat -> Nat -- Successor, +1

-- 0 = 0
-- S 0 = 1
-- S (S 0) = 2
-- etc...
```

Addition looks like this:

```
(+) : Nat -> Nat -> Nat
(+) 0 y = y
(+) (S x) y = S (x + y)
```

Associativity

```
plus_assoc : (x, y, z : Nat) -> x + (y + z) = (x + y) + z
plus_assoc x y z = ?plus_assoc_rhs
```

Associativity

```
plus_assoc : (x, y, z : Nat) -> x + (y + z) = (x + y) + z
plus_assoc x y z = ?plus_assoc_rhs
```

Case split on `x`...

```
plus_assoc : (x, y, z : Nat) -> x + (y + z) = (x + y) + z
plus_assoc Z y z = ?plus_assoc_rhs_1
plus_assoc (S k) y z = ?plus_assoc_rhs_2
```

Associativity

```
plus_assoc : (x, y, z : Nat) -> x + (y + z) = (x + y) + z
plus_assoc x y z = ?plus_assoc_rhs
```

Case split on x...

```
plus_assoc : (x, y, z : Nat) -> x + (y + z) = (x + y) + z
plus_assoc Z y z = ?plus_assoc_rhs_1
plus_assoc (S k) y z = ?plus_assoc_rhs_2
```

Gives us some interesting holes...

```
- + Main.plus_assoc_rhs_1 [P]
 `--                      y : Nat
                          z : Nat
    ---------------------------------------------------------------
     Main.plus_assoc_rhs_1 : 0 + (y + z) = (0 + y) + z

- + Main.plus_assoc_rhs_2 [P]
 `--                      k : Nat
                          y : Nat
                          z : Nat
    --
         --------------------------------------------------------------------
     Main.plus_assoc_rhs_2 : (S k) + (y + z) = ((S k) + y) + z
```

First case...

We have:

```
     - + Main.plus_assoc_rhs_1 [P]
      `--                      y : Nat
                               z : Nat
          ----------------------------------------------------------------
           Main.plus_assoc_rhs_1 : 0 + (y + z) = (0 + y) + z
```

First case...

We have:

```
    - + Main.plus_assoc_rhs_1 [P]
     `--                        y : Nat
                                z : Nat
         ----------------------------------------------------------------
          Main.plus_assoc_rhs_1 : 0 + (y + z) = (0 + y) + z
```

Idris will evaluate expressions in an equality type when we use `Refl`, so this hole is really more like:

```
    - + Main.plus_assoc_rhs_1 [P]
     `--                        y : Nat
                                z : Nat
         ----------------------------------------------------------------
          Main.plus_assoc_rhs_1 : y + z = y + z
```

First case...

We have:

```
- + Main.plus_assoc_rhs_1 [P]
 `--                       y : Nat
                           z : Nat
    --------------------------------------------------------------
     Main.plus_assoc_rhs_1 : 0 + (y + z) = (0 + y) + z
```

Idris will evaluate expressions in an equality type when we use `Refl`, so this hole is really more like:

```
- + Main.plus_assoc_rhs_1 [P]
 `--                       y : Nat
                           z : Nat
    --------------------------------------------------------------
     Main.plus_assoc_rhs_1 : y + z = y + z
```

Which is just satisfied with reflexivity...

```
plus_assoc : (x, y, z : Nat) -> x + (y + z) = (x + y) + z
plus_assoc Z y z = Refl
plus_assoc (S k) y z = ?plus_assoc_rhs_2
```

Second case...

Here's our goal:

```
     - + Main.plus_assoc_rhs_2 [P]
      `--                      k : Nat
                               y : Nat
                               z : Nat
          ------------------------------------------------------------
           Main.plus_assoc_rhs_2 : (S k) + (y + z) = ((S k) + y) + z
```

Second case...

Here's our goal:

```
     - + Main.plus_assoc_rhs_2 [P]
      `--                       k : Nat
                                y : Nat
                                z : Nat
          ------------------------------------------------------------
           Main.plus_assoc_rhs_2 : (S k) + (y + z) = ((S k) + y) + z
```

Idris can evaluate this some to simplify as well.

```
     - + Main.plus_assoc_rhs_2 [P]
      `--                       k : Nat
                                y : Nat
                                z : Nat
          ------------------------------------------------------------
           Main.plus_assoc_rhs_2 : S (k + (y + z)) = S ((k + y) + z)
```

It loooks like we need to prove associativity again...

Second case...

Here's our goal:

```
     - + Main.plus_assoc_rhs_2 [P]
      `--                     k : Nat
                              y : Nat
                              z : Nat
          -----------------------------------------------------------
           Main.plus_assoc_rhs_2 : (S k) + (y + z) = ((S k) + y) + z
```

Idris can evaluate this some to simplify as well.

```
     - + Main.plus_assoc_rhs_2 [P]
      `--                     k : Nat
                              y : Nat
                              z : Nat
          -----------------------------------------------------------
           Main.plus_assoc_rhs_2 : S (k + (y + z)) = S ((k + y) + z)
```

It loooks like we need to prove associativity again...

```
    k + (y + z) = (k + y) + z
```

Recursion is induction!

Idris knows about recursion, so we can actually call `plus_assoc` on `k`, `y`, and `z` to get something with type...

```
attempt : (k y z : Nat) -> k + (y + z) = (k + y) + z
attempt k y z = plus_assoc k y z
```

Recursion is induction!

Idris knows about recursion, so we can actually call `plus_assoc` on `k`, `y`, and `z` to get something with type...

```
attempt : (k y z : Nat) -> k + (y + z) = (k + y) + z
attempt k y z = plus_assoc k y z
```

So, now we just need to add `S` to both sides of this... Hmmm...

Enter cong

This is what we use `cong` for, if you remember...

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f (Refl {x}) = Refl {x = f x}
```

Enter cong

This is what we use cong for, if you remember...

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f (Refl {x}) = Refl {x = f x}
```

If we give cong a function, and an equality type, it will apply the function to both sides!

Enter cong

This is what we use cong for, if you remember...

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f (Refl {x}) = Refl {x = f x}
```

If we give cong a function, and an equality type, it will apply the function to both sides!

```
plus_assoc : (x, y, z : Nat) -> x + (y + z) = (x + y) + z
plus_assoc Z y z = Refl
plus_assoc (S k) y z = cong S (plus_assoc k y z)
```

Enter cong

This is what we use cong for, if you remember...

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f (Refl {x}) = Refl {x = f x}
```

If we give cong a function, and an equality type, it will apply the function to both sides!

```
plus_assoc : (x, y, z : Nat) -> x + (y + z) = (x + y) + z
plus_assoc Z y z = Refl
plus_assoc (S k) y z = cong S (plus_assoc k y z)
```

This completes the proof!

Enter cong

This is what we use cong for, if you remember...

```
cong : (f : a -> b) -> x = y -> f x = f y
cong f (Refl {x}) = Refl {x = f x}
```

If we give cong a function, and an equality type, it will apply the function to both sides!

```
plus_assoc : (x, y, z : Nat) -> x + (y + z) = (x + y) + z
plus_assoc Z y z = Refl
plus_assoc (S k) y z = cong S (plus_assoc k y z)
```

This completes the proof!
Neat how applying a theorem is just applying a function. Also neat how recursion and induction are really just the same thing.

Tactics

Can use a different meta-language like Coq's tactics to aid proofs...

Tactics

Can use a different meta-language like Coq's tactics to aid proofs...

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.


Fixpoint plus (n m : nat) : nat :=
  match n with
    | O => m
    | S n' => S (plus n' m)
  end.


Theorem plus_assoc : forall (x y z : nat), plus x (plus y z) = plus (plus x
     y) z.
Proof.
  intros x y z. induction x as [| k].
  - reflexivity.
  - simpl. (* Simplify with evaluation *)
    rewrite IHk. (* Use induction hypothesis to rewrite terms *)
    reflexivity.
Qed.
```

Tactics

Can use a different meta-language like Coq's tactics to aid proofs...

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.


Fixpoint plus (n m : nat) : nat :=
  match n with
    | O => m
    | S n' => S (plus n' m)
  end.


Theorem plus_assoc : forall (x y z : nat), plus x (plus y z) = plus (plus x
     y) z.
Proof.
  intros x y z. induction x as [| k].
  - reflexivity.
  - simpl. (* Simplify with evaluation *)
    rewrite IHk. (* Use induction hypothesis to rewrite terms *)
    reflexivity.
Qed.
```

Can allow for very succinct and easy proof development, since meta-language can perform large automated steps!

Conclusion! Questions?

Whirlwind introduction, so you probably have many!

References

- Propositions as Types
- Type-Driven Development
- Software Foundations

These are all good resources! You should look at them!