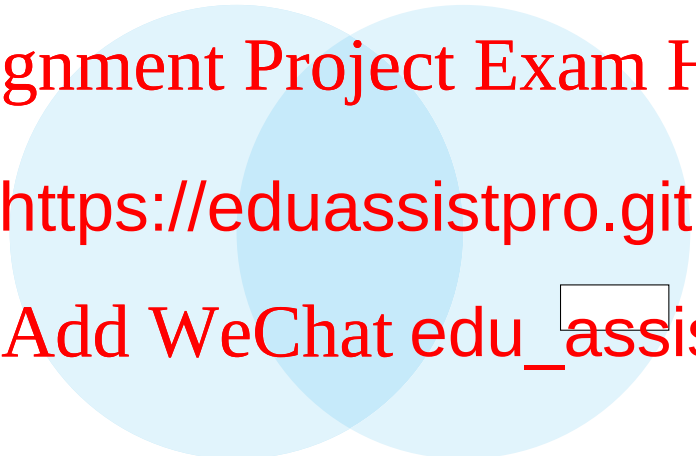
COMPSCI4

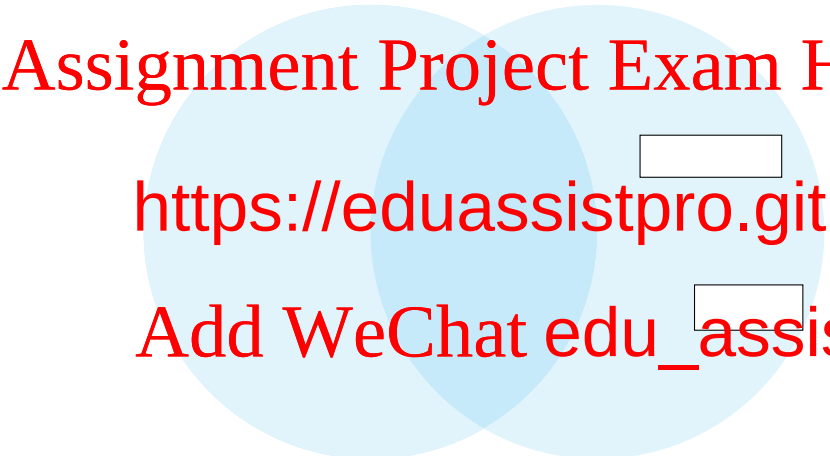
Liam O'Conn
University of Edinburgh, LFCS (ar

Term 2, 2020

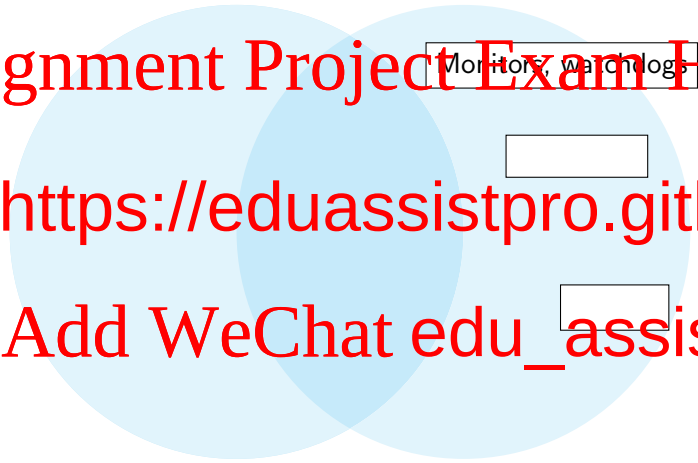
Methods of Assurance



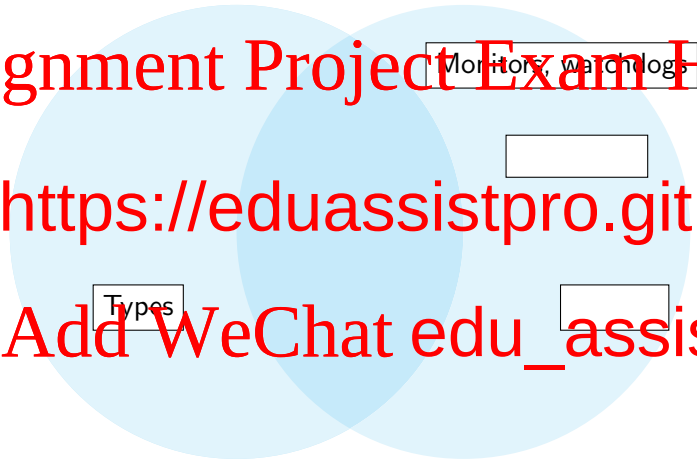
Methods of Assurance



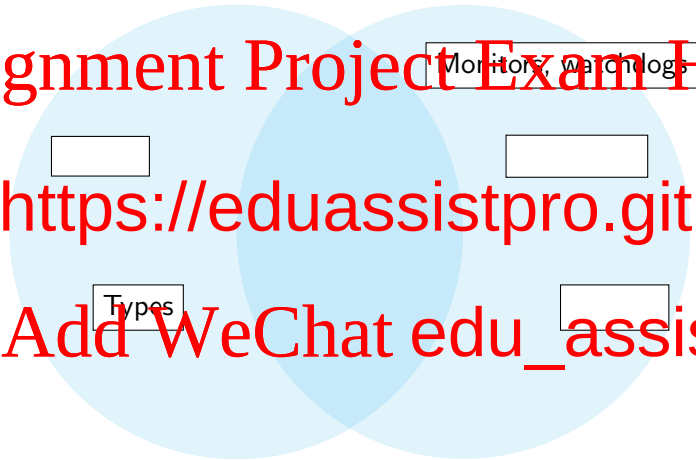
Methods of Assurance



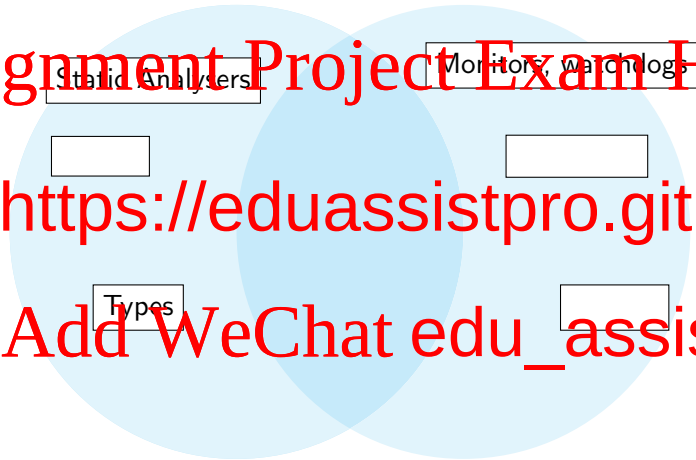
Methods of Assurance



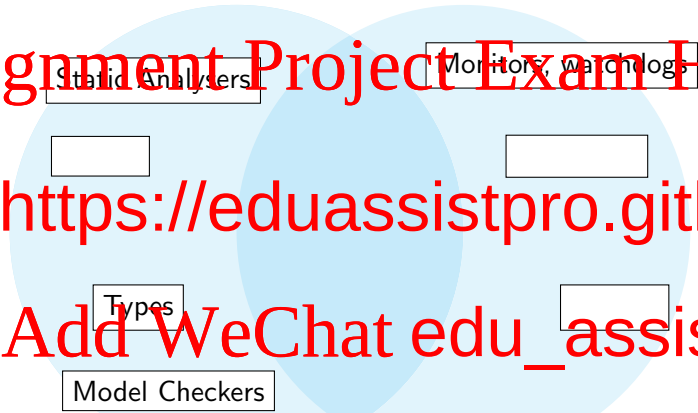
Methods of Assurance



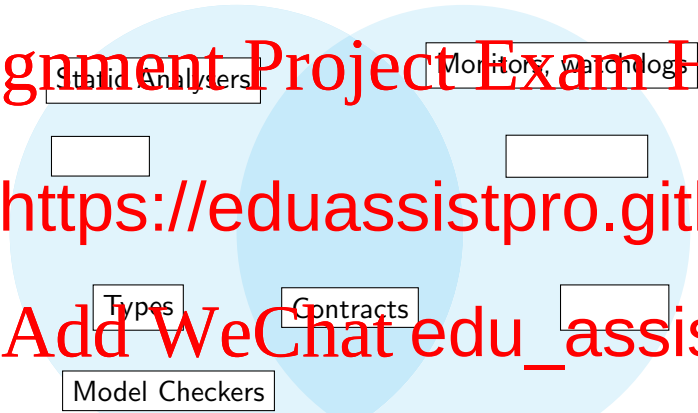
Methods of Assurance



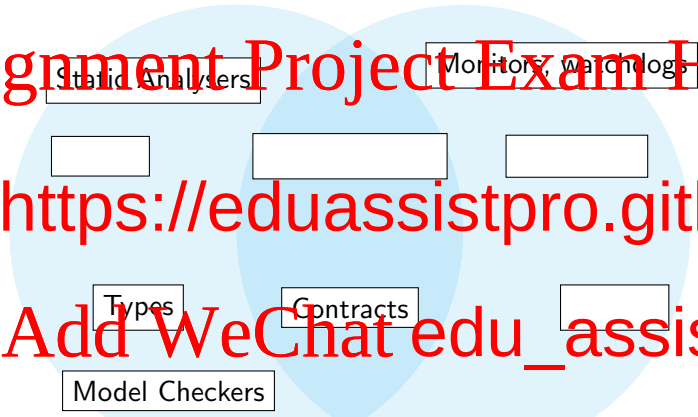
Methods of Assurance



Methods of Assurance



Methods of Assurance



Static means of assurance analyse a program without running it.

Static vs. Dynamic

- Static checks can be exhaustive.

Static vs. Dynamic

- Static checks can be exhaustive.

Exhaustivity

An exhaustive check...
program.

Static vs. Dynamic

- Static checks can be exhaustive.

Exhaustivity

An exhaustive check...
program.

- **However**, some properties cannot be checked statica
problem), or are intractable to feasibly check statically (
- Dynamic checks cannot be exhaustive, but can be used to c
where static methods are unsuitable.

Compiler Integration

Most static and all dynamic methods of assurance are not integrated into the compilation process.

Compiler Integration

Most static and all dynamic methods of assurance are not integrated into the compilation process.

- You can co

Compiler Integration

Most static and all dynamic methods of assurance are not integrated into the compilation process.

- You can co
- You can cha

Compiler Integration

Most static and all dynamic methods of assurance are not integrated into the compilation process.

- You can co
- You can cha
- Your proof

Compiler Integration

Most static and all dynamic methods of assurance are not integrated into the compilation process.

- You can co
- You can cha
- Your proof

Types

Because types are integrated into the compiler, they cannot di code. This means that type signatures are a kind of machine-che for your code.

Types

Types are the most widely used kind of formal verification in programming today.

- They are che
- They can be e
  expressivi
- They are an exhaustive analysis.

Types

Types are the most widely used kind of formal verification in programming today.

- They are che
- They can be e
  expressivi
- They are an exhaustive analysis.

This week, we'll look at techniques to encode various correctn
Haskell's type system.

Phantom Types

Definition

A type parameter is *phantom* if it does not appear in the right hand side of the type definition.

```
newtype Size x = S Int
```

Phantom Types

Definition

A type parameter is *phantom* if it does not appear in the right hand side of the type definition.


```
newtype Size x = S Int
```

Lets examine each one of the following use cases:

- We can use this parameter to track what data invariants h about a value.

Phantom Types

Definition

A type parameter is *phantom* if it does not appear in the right hand side of the type definition.

```
newtype Size x = S Int
```


Lets examine each one of the following use cases:

- We can use this parameter to track what data invariants h about a value.
- We can use this parameter to track information about the representation (e.g. units of measure).

Phantom Types

Definition

A type parameter is *phantom* if it does not appear in the right hand side of the type definition.

```
newtype Size x = S Int
```


Lets examine each one of the following use cases:

- We can use this parameter to track what data invariants h about a value.
- We can use this parameter to track information about the representation (e.g. units of measure).
- We can use this parameter to enforce an ordering of operations performed on these values (*type state*).

Validation

```
data UG -- empty type
data PG
data StudentID x = SID Int
```

Validation


```
data UG -- empty type
data PG
data StudentID x = SID Int
```

We can define a sma

```
sid :: Int -> Either
```


(Recalling the following definition of Either)

```
data Either a b = Left a | Right b
```

Validation


```
data UG -- empty type
data PG
data StudentID x = SID Int
```

We can define a sma


```
sid :: Int -> Either
```

(Recalling the following definition of Either)


```
data Either a b = Left a | Right b
```

And then define functions:

```
enrolInCOMP3141 :: StudentID UG -> IO ()
lookupTranscript :: StudentID x -> IO String
```

Units of Measure

In 1999, software confusing units of measure (pounds and newtons) caused a Mars orbiter to burn up on atmospheric entry.


```
data Kilometres
data Miles
data Value x = MkInt Int
sydneyToMel
losAngelesToSanFran = (U 383 :: Value Miles)
```

Units of Measure

In 1999, software confusing units of measure (pounds and newtons) caused a Mars orbiter to burn up on atmospheric entry.


```
data Kilometres
data Miles
data Value x = U Int
sydneyToMel
losAngelesToSanFran = (U 383 :: Value Miles)
```


In addition to tagging values, we can also enforce constraints on

```
data Square a
area :: Value m -> Value m -> Value (Square m)
area (U x) (U y) = U (x * y)
```

Note the arguments to `area` must have the same units.

Type State

Example

A Socket can either be ready to retrieve data, or busy. If the socket is busy, the user must first use the `wait` operation, which blocks until the socket is ready. If the socket is ready, the user ca ke the socket busy again.

Type State

Example

A Socket can either be ready to retrieve data, or busy. If the socket is busy, the user must first use the `wait` operation, which blocks until the socket is ready. If the socket is ready, the user ca ke the socket busy again.


```
data Busy
data Ready
newtype Socket s = Socket
```

```
wait :: Socket Busy -> IO (Socket Ready)
```

```
send :: Socket Ready -> String -> IO (Socket Busy)
```

What assumptions are we making here?

Linearity and Type State

The previous code assumed that we didn't re-use old Sockets:


```
send2 :: Socket Ready -> String -> String
      -> IO (Socket Busy)
send2 s x y = do s' <- send s x
```



```
                 pure s'''
```

Linearity and Type State

The previous code assumed that we didn't re-use old Sockets:

```
send2 :: Socket Ready -> String -> String
      -> IO (Socket Busy)
send2 s x y = do s' <- send s x
                 s'' <- send s' y
                 pure s'''
```

But we can just re-use old values to send without waiting:

```
send2' s x y = do _ <- send s x
                  s' <- send s y
                  pure s'
```

Linearity and Type State

The previous code assumed that we didn't re-use old Sockets:

```
send2 :: Socket Ready -> String -> String
      -> IO (Socket Busy)
send2 s x y = do s' <- send s x
                 pure s'''
```

But we can just re-use old values to send without waiting:

```
send2' s x y = do _  <- send s x
                  s' <- send s y
                  pure s'
```

Linear type systems can solve this, but not in Haskell (yet).

Datatype Promotion

```
data UG
data PG
data StudentId x = SID Int
```

Datatype Promotion

```
data UG
data PG
data StudentID x = SID Int
```

Defining empty d dentID UG, but also StudentI

Datatype Promotion

```
data UG
data PG
data StudentID x = SID Int
```

Defining empty d dentID UG, but also StudentI

Recall

Haskell types themselves have types, called kinds. Can we mak types more precise than *?

Datatype Promotion

```
data UG
data PG
data StudentID x = SID Int
```

Defining empty d dentID UG, but also StudentI

Recall

Haskell types themselves have types, called kinds. Can we mak types more precise than *?

The DataKinds language extension lets us use data types

```
{-# LANGUAGE DataKinds, KindSignatures #-}
data Stream = UG | PG
data StudentID (x :: Stream) = SID Int
-- rest as before
```

Motivation: Evaluation


```
data Expr = BConst Bool
          | IConst Int
          | T
          | L
          | A
          | I

data Value = BVal Bool | IVal Int
```

Example

Define an expression evaluator:

```
eval :: Expr -> Value
```

Motivation: Partiality

Unfortunately, the eval function is partial, undefined for input expressions that are not well-typed, like:

```
And (ICons 3) (BC
```

Motivation: Partiality

Unfortunately the eval function is partial, undefined for input expressions that are not well-typed, like:

```
And (ICons 3) (BC
```


Recall

With any partial function, we can make it total by either expanding the co-domain (e.g. with a `Maybe` type), or constraining the domain.

Motivation: Partiality

Unfortunately, the `eval` function is partial, undefined for input expressions that are not well-typed, like:

```
And (ICons 3) (BC
```


Recall

With any partial function, we can make it total by either expanding the co-domain (e.g. with a `Maybe` type), or constraining the domain.

Can we use phantom types to constrain the domain of `eval` to only accept well-typed expressions?

Attempt: Phantom Types

Let's try adding a phantom parameter to `Expr`, and defining typed constructors with precise types:

Attempt: Phantom Types

Let's try adding a phantom parameter to `Expr`, and defining typed constructors with precise types:


```
data Expr t = ...
bConst :: Bool -> Expr Bool
bConst = BConst
iConst :: Int -> Expr Int
iConst = IConst
times :: Expr Int -> Expr Int -> Expr Int
times = Times
less :: Expr Int -> Expr Int -> Expr Bool
less = Less
and :: Expr Bool -> Expr Bool -> Expr Bool
and = And
if'  :: Expr Bool -> Expr a -> Expr a -> Expr a
if' = If
```

Attempt: Phantom Types

This makes invalid expressions into type errors (yay!):

```
-- Couldn't match Int and Bool
and (iConst 3) (bConst True)
```

Attempt: Phantom Types

This makes invalid expressions into type errors (yay!):

```
-- Couldn't match Int and Bool
and (iConst 3) (bConst True)
```

How about our

Attempt: Phantom Types

This makes invalid expressions into type errors (yay!):

```
-- Couldn't match Int and Bool
and (iConst 3) (bConst True)
```

How about our

Attempt: Phantom Types

This makes invalid expressions into type errors (yay!):

```
-- Couldn't match Int and Bool
and (iCons 3) (bConst True)
```

How about our

Attempt: Phantom Types

This makes invalid expressions into type errors (yay!):

```
-- Couldn't match Int and Bool
and (iConst 3) (bConst True)
```

How about our

Bad News

Inside `eval`, the Haskell type checker cannot be sure that we us
constructors, so # e.g. the `IConst` case


```
eval :: Expr t -> t
eval (IConst i) = i -- type error
```

We are unable to tell that the type `t` is definitely `Int`.

Attempt: Phantom Types

This makes invalid expressions into type errors (yay!):

```
-- Couldn't match Int and Bool
and (iCons 3) (bConst True)
```


How about our

Bad News

Inside `eval`, the Haskell type checker cannot be sure that we us
constructors, so e.g. the `IConst` case


```
eval :: Expr t -> t
eval (IConst i) = i -- type error
```

We are unable to tell that the type `t` is definitely `Int`.

Phantom types aren't strong enough!

GADTs

Generalised Algebraic Datatypes (*GADTs*) is an extension to Haskell that, among other things, allows data types to be specified by writing the types of their constructors:


```
{-# LANGUAGE GA
-- Unary natura
data Nat = Z | S Nat
-- is the same as
data Nat :: * where
  Z :: Nat
  S :: Nat -> Nat
```

GADTs

Generalised Algebraic Datatypes (*GADTs*) is an extension to Haskell that, among other things, allows data types to be specified by writing the types of their constructors:


```
{-# LANGUAGE GA
-- Unary natura
data Nat = Z | S Nat
-- is the same as
data Nat :: * where
  Z :: Nat
  S :: Nat -> Nat
```


When combined with the *type indexing* trick of phantom types, this becomes very powerful!

Expressions as a GADT

```
data Expr :: * -> * where
   BConst :: Bool -> Expr Bool
   IConst :: Int -> Expr Int
   Times :: Expr Int -> Expr Int -> Expr Int
   Less :: Expr I
   And :: Expr Bo
   If  :: Expr Bool -
```

Expressions as a GADT

```
data Expr :: * -> * where
  BConst :: Bool -> Expr Bool
  IConst :: Int -> Expr Int
  Times :: Expr Int -> Expr Int -> Expr Int
  Less :: Expr I
  And :: Expr Bo
  If  :: Expr Bool -
```


Observation

There is now only one set of type-safe constructors

Expressions as a GADT

```
data Expr :: * -> * where
  BConst :: Bool -> Expr Bool
  IConst :: Int -> Expr Int
  Times :: Expr Int -> Expr Int -> Expr Int
  Less :: Expr I
  And :: Expr Bo
  If  :: Expr Bool -
```


Observation

There is now only one set of precisely-typed constructors.

Inside `eval` now, the Haskell type checker accepts our previously problematic case:

```
eval :: Expr t -> t
eval (IConst i) = i -- OK now
```

Expressions as a GADT

```
data Expr :: * -> * where
  BConst :: Bool -> Expr Bool
  IConst :: Int -> Expr Int
  Times :: Expr Int -> Expr Int -> Expr Int
  Less :: Expr I
  And :: Expr Bo
  If  :: Expr Bool -
```


Observation

There is now only one set of precisely-typed constructors.

Inside `eval` now, the Haskell type checker accepts our previously problematic case:

```
eval :: Expr t -> t
eval (IConst i) = i -- OK now
```

GHC now knows that if we have `IConst`, the type `t` must be `Int`.

Lists

We could define our own list type using GADT syntax as follows:

```
data List (a :: *) :: * w
  Nil  :: List a
  Cons :: a -> List a -
```

Lists

We could define our own list type using GADT syntax as follows:

```
data List (a :: *) :: * w
  Nil  :: List a
  Cons :: a -> List a -
```


But, if we define hea

```
hd (Cons x xs) = x
tl (Cons x xs) = xs
```

Lists

We could define our own list type using GADT syntax as follows:

```
data List (a :: *) :: * w
  Nil  :: List a
  Cons :: a -> List a -
```


But, if we define hea

```
hd (Cons x xs) = x
tl (Cons x xs) = xs
```


We will constrain the domain of these functions by tracking the l
the type level.

Vectors

As before, define a natural number kind to use on the type level:

```
data Nat = Z | S Nat
```

Vectors

As before, define a natural number kind to use on the type level:

```
data Nat = Z | S Nat
```

Now our length-i

```
data Vec (a :: *) :: Na
  Nil  :: Vec a Z
  Cons :: a -> Vec a n -> Vec a (S n)
```

Vectors

As before, define a natural number kind to use on the type level:

```
data Nat = Z | S Nat
```

Now our length-i

```
data Vec (a :: *) :: Na
  Nil  :: Vec a Z
  Cons :: a -> Vec a n -> Vec a (S n)
```


Now `hd` and `tl` can be total:


```
hd :: Vec a (S n) -> a
hd (Cons x xs) = x
tl :: Vec a (S n) -> Vec a n
tl (Cons x xs) = xs
```

Vectors, continued

Our `map` for vectors is as follows:

```
mapVec :: (a -> b) -> Vec a n -> Vec b n
mapVec f Nil = Nil
mapVec f (Cons x xs
```

Vectors, continued

Our `map` for vectors is as follows:

```
mapVec :: (a -> b) -> Vec a n -> Vec b n
mapVec f Nil = Nil
mapVec f (Cons x xs
```


Properties

Using this type, it's impossible to write a `mapVec` that changes the length of the vector.

Properties are verified by the compiler!

Tradeoffs

The benefits of this extra static checking are obvious, however:

- It can be difficult to convince the Haskell type checker that your code is correct, even when it is.

Tradeoffs

The benefits of this extra static checking are obvious, however:

- It can be difficult to convince the Haskell type checker that your code is correct, even when it is.
- Type-leve
  understan

Tradeoffs

The benefits of this extra static checking are obvious, however:

- It can be difficult to convince the Haskell type checker that your code is correct, even when it is.
- Type-leve
  understan
- Sometime
  productivi

ing

Tradeoffs

The benefits of this extra static checking are obvious, however:

- It can be difficult to convince the Haskell type checker that your code is correct, even when it is.
- Type-leve
  understan
- Sometime
  productivi

Pragmatism

We should use type-based encodings only when the assuranc
clarity disadvantages.

Tradeoffs

The benefits of this extra static checking are obvious, however:

- It can be difficult to convince the Haskell type checker that your code is correct, even when it is.
- Type-leve
  understan
- Sometime
  productivi

Pragmatism

We should use type-based encodings only when the assuranc
clarity disadvantages.
The typical use case for these richly-typed structures is to eliminate partial functions from our code base.

Tradeoffs

The benefits of this extra static checking are obvious, however:

- It can be difficult to convince the Haskell type checker that your code is correct, even when it is.
- Type-leve
  understan
- Sometime
  productivi

Pragmatism

We should use type-based encodings only when the assuranc
clarity disadvantages.
The typical use case for these richly-typed structures is to eliminate partial functions from our code base.
If we never use partial list functions, length-indexed vectors are not particularly useful.

Appending Vectors

Example (Problem)

```
appendV :: Vec a m -> Vec a n -> Vec a ???
```

Appending Vectors

Example (Problem)

```
appendV :: Vec a m -> Vec a n -> Vec a ???
```

We want to write for kind `Nat`.

Appending Vectors

Example (Problem)

```
appendV :: Vec a m -> Vec a n -> Vec a ???
```

We want to write for kind `Nat`.

We can define a nor

```
plus :: Nat -> Nat -> Nat
plus Z y = y
plus (S x) y = S (plus x y)
```

Appending Vectors

Example (Problem)

```
appendV :: Vec a m -> Vec a n -> Vec a ???
```

We want to write for kind `Nat`.

We can define a nor

```
plus :: Nat -> Nat -> Nat
plus Z y = y
plus (S x) y = S (plus x y)
```


This function is not applicable to type-level `Nats`, though.

Appending Vectors

Example (Problem)

```
appendV :: Vec a m -> Vec a n -> Vec a ???
```

We want to write for kind `Nat`.

We can define a nor

```
plus :: Nat -> Nat -> Nat
plus Z y = y
plus (S x) y = S (plus x y)
```


This function is not applicable to type-level `Nats`, though.

$\Rightarrow$ we need a type level function.

Type Families

Type-level functions, also called *type families*, are defined in Haskell like so:


```
{-# LANGUAGE Ty
type family Plu
  Plus Z     y = y
  Plus (S x) y = S (Plu
```

We can use our type family to define appendV:


```
appendV :: Vec a n -> Vec a m -> Vec a (Plus n m)
appendV Nil         ys = ys
appendV (Cons x xs) ys = Cons x (appendV xs ys)
```

Recursion

If we had implemented `Plus` by recursing on the second argument instead of the first:

```
{-# LANGUAGE TypeFamilies #-}
type family Plus' (x :: Nat) (y :: Nat) :: Nat where
  Plus' x Z     = x
  Plus' x (S y) = S (Pl
```

Recursion

If we had implemented `Plus` by recursing on the second argument instead of the first:

```
{-# LANGUAGE TypeFamilies #-}
type family Plus' (x :: Nat) (y :: Nat) :: Nat where
  Plus' x Z     = x
  Plus' x (S y) = S (Pl
```

Then our `app`

```
appendV :: Vec a n -> V
appendV Nil         ys = ys
appendV (Cons x xs) ys = Cons x (appendV xs ys)
```

Why?

Recursion

If we had implemented `Plus` by recursing on the second argument instead of the first:

```
{-# LANGUAGE TypeFamilies #-}
type family Plus' (x :: Nat) (y :: Nat) :: Nat where
  Plus' x Z     = x
  Plus' x (S y) = S (Pl
```

Then our `app`

```
appendV :: Vec a m -> V
appendV Nil         ys = ys
appendV (Cons x xs) ys = Cons x (appendV xs ys)
```

Why?

Answer

Consider the `Nil` case. We know `m = Z`, and must show that our desired return type `Plus' Z n` equals our given return type `n`, but that fact is not immediately apparent from the equations.

Type-driven development

- This lecture is only a taste of the full power of type-based specifications.
- Language e
  and value lev s.
- Haskell is als

Next week: Fancy theory about types!

- Deep connections between types, logic and proof
- Algebraic type structure for generic algorithms and ref
- Using polymorphic types to infer properties for free.

Homework

1. Assignme
2. The last pro
3. This week's quiz is also up, due in Friday of Week 9.