Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 10, Part 1
Haskell extensions

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Haskell extensions

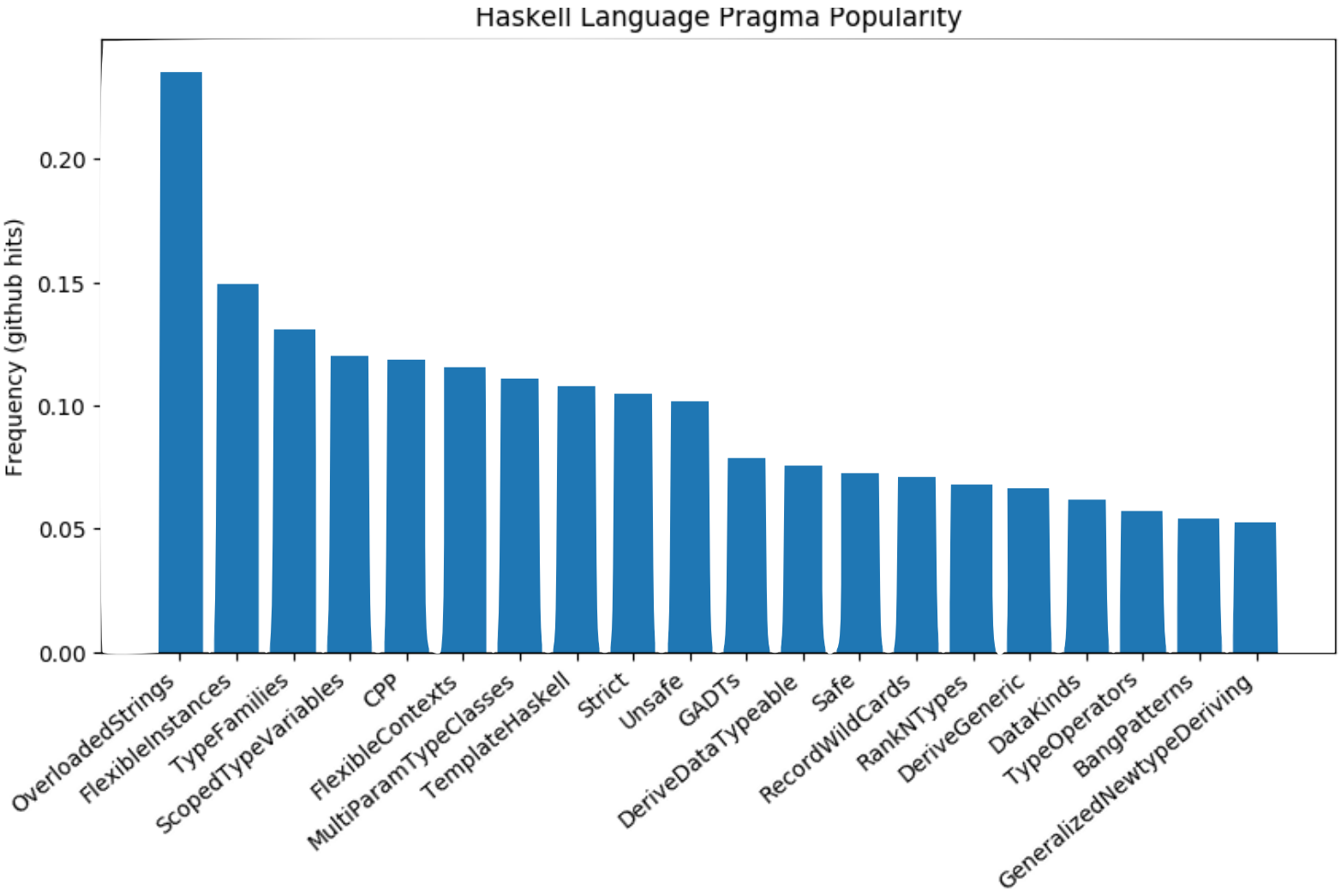
Haskell is standardised in two versions (Haskell98 and Haskell2010). But research into functional languages has not stopped!

GHC implements many "extensions". Some are simple and fairly uncontroversial (OverloadedStrings, for example). Some are outright known to be a bad idea (NPlusKPatterns) but are retained as options in the compiler

- **I counted 129 language extensions in my version of GHC.**
- **Not going to try to discuss them all.**
- **Everyone has their own idea about what is worthwhile, unwise to use, etc.**

Haskell extensions

atondwal on Github:



Haskell extensions

How to use

You can turn on a specific extension either with a command line directive to the compiler (which could be specified in your .cabal file).

To turn on an extension called “Foo” you specify “-XFoo” on the command line.

You can also specify this with a specially formatted comment at the top of a module, which will turn on extensions for the module.

```
{-# LANGUAGE Foo, Bar, NoBaz -#}
```

this will turn on Foo and Bar, and turn *off* Baz

Haskell extensions

Fairly uncontroversial extensions

Some extensions are little more than simple syntactic cleanups.

TupleSections

```
(x,,) = \y z -> (x, y, z)
```

LambdaCase

```
\x -> case x of 1 -> ...
                2 -> ...
```

Becomes

```
\case
  1 -> ...
  2 -> ...
```

Haskell extensions

Fairly uncontroversial extensions

Some extensions are little more than simple syntactic cleanups.

MultiWayIf

```
if val1 == val2
  then e1
  else if val1 == val3
    then e2
    else e3
```

Becomes

```
if | val1 == val2 -> e1
   | val1 == val2 -> e2
   | otherwise    -> e3
```

Haskell extensions

Fairly uncontroversial extensions

Some extensions are little more than simple syntactic cleanups.

NumericUnderscores enables numeric constants like this:

```
123_456
```

OverloadedStrings creates

```
class IsString a where
  fromString :: String -> a
```

OverloadedLists creates a similar “isList” and “fromList” arrangement for list literals, which can allow, for example, a literal of type Map String Int with this definition:

```
[("default",0), (k1,v1)]
```

Haskell extensions

Richer pattern matching

ViewPatterns lets you apply a function as part of a pattern.

Instead of:

```
f x = case toList x of
  []     -> 0
  (x:xs) -> x
```

You can write:

```
f (toList -> [])     = 0
f (toList -> (x:xs)) = x
```

Haskell extensions

Small type system extensions

MultiParamTypeClasses

Allows type classes with more than one type variable

FlexibleInstances

Allows type class declarations to mention arbitrary nested types.

FlexibleContexts

Lifts the Haskell98 restriction that contexts in class constraints must be a simple class applied to type variables.

TypeSynonymInstances

Allows class instances to be declared for type synonyms

Haskell extensions

Large system extensions

ExistentialQuantification

GADTs

DataKinds and KindSignatures

TemplateHaskell

These are big enough to need their own lectures; we'll look at the most important of them next.

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 1

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 10, Part 2
Type system extensions (1)

CS4012

Topics in Functional Programming

Michaelmas Term 2020

Glenn Strong <glenn.Strong@scss.tcd.ie>

Type System Extensions

The `classic' Haskell type system is expressive and quite powerful.

But it is hardly the last word - research in making type systems more expressive is a big thing!

Haskell programmers often treat the type of a function as a kind of static specification

- **Lightweight**
- **Machine checked**
- **Ubiquitous**

Type System Extensions

Generally, type system designers are interested in:

- **Making more *correct* programs get through the type checker**
- **Making fewer *incorrect* programs get through the type checker**

We will examine some of the more interesting extensions to the Haskell type system

Phantom Types

Recall that types are declared by a "data" or "newtype" declaration that includes some parameters

```
data T a b c = TC stuff
```

It works very much as you'd imagine.

- **"a", "b", and "c" are the type parameters to the new type being declared**
- **Implicitly these are *universally quantified***

Phantom Types

The same is true for function types. When you write:

```
length :: [a] -> Int
```

You are implicitly saying something like:

```
length :: ∀a. [a] -> Int
```

In fact you can even write this (or something like this) if you like

```
length :: forall a. [a] -> Int
```

(use the -XRankNTypes option to GHC to enable this syntax)

Phantom Types

One last obvious thing to say is that data types can have multiple constructors, and not every parameter to the type need be mentioned in every constructor.

For example:

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

the data constructors define functions which create values of the type Maybe…

```
Nothing :: ∀a .      Maybe a
Just    :: ∀a . a -> Maybe a
```

Phantom Types

So, given all that, consider this type:

```
data Lis a = Lis [Int]
```

What type does the constructor have here?

```
Lis :: ∀a. Int -> Lis a
```

We refer to the type parameter ‘a’ as a “Phantom” type parameter, since it doesn’t appear anywhere on the right-hand side.

Phantom Types

Let's add these somewhat odd looking (but perfectly legal) types:

```
data Even = Even
data Odd  = Odd
```

Using these we can teach the compiler to distinguish between lists of even and odd length!

Phantom Types

```
consE :: Int -> Lis Even -> Lis Odd
consE i (Lis l) = Lis (i:l)

consO :: Int -> Lis Odd -> Lis Even
consO i (Lis l) = Lis (i:l)
```

With these definitions, if I write:

```
myList = consO 10 (consE 5 nil)
```

The compiler can deduce:

```
myList :: Lis Even
```

Phantom Types

And if I write:

```
consO 1 myList
```

The compiler concludes:

```
Couldn't match expected type `Odd' with actual
type `Even'
Expected type: Lis Odd
  Actual type: Lis Even
In the second argument of `consO', namely `aList'
In the expression: consO 1 myList
```

Phantom Types

It's a neat stunt, I suppose, but is it useful for anything *practical*?

How would you feel about *typed pointers?*

```
data Ptr a = MkPtr Addr
```

With operations:

```
peek :: Ptr a      -> IO a
poke :: Ptr a -> a -> IO ()
```

This kind of bad code can be eliminated at compile time:

```
do ptr <- allocate
   poke ptr (42::Int)
   bad::Float <- peek ptr
```

Phantom Types

We can write functions that are polymorphic in those "phantom" type parameters

```
lisMap f (Lis x) = Lis (map f x)
```

The compiler deduces this type:

```
lisMap :: (Int -> Int) -> (Lis a) -> (Lis b)
```

Actually, we can do better manually here; we know the parity of the list won't change under mapping, but the type inference system cannot prove that by itself:

```
lisMap :: (Int -> Int) -> (Lis a) -> (Lis a)
```

Phantom Types

Phantom types represent a lightweight (since they don't require any extensions to the type system) way to capture certain kinds of invariants.

Some typical uses of Phantom Types include:

- **Tracking types in embedded languages**
  - **We can say "Exp a" be the type of expressions evaluating to a value a**
  - **(we can extend this idea to do even better, actually)**
- **Object Hierarchy models (this happens in the Haskell "GTK" library)**

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 2

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 10, Part 3
Type system extensions (2)

CS4012

Topics in Functional Programming

Michaelmas Term 2020

Glenn Strong <glenn.Strong@scss.tcd.ie>

Existential Types

Lists are normally *homogenous*. By which we mean:

```
data List a = Nil | Cons a (List a)
```

Or, in traditional notation:

```
data [a]    = []  | (:)  a [a]
```

Which means values like this are impossible:

```
 [ "foo", "bar", 12.3 ]
```

There is (obviously) no single 'a' that can be universally quantified over the body of the list.
The type checker has to reject this expression (and rightly so!)

Existential Types

But if we were allowed to perform a bit of sleight-of-hand and shuffle the quantifier *inside* the declaration:

```
data forall a . HList a = ...
```

To:

```
data HList = HNil
           | forall a . HCons a HList
```

Then the quantification could be though of as being over each application of the HCons constructor.

Existential Types

The ExistentialTypes extension allows exactly this kind of declaration (and yes, we say ∀/forall, not ∃/exists. See later).

```
f = HCons "foo" (HCons "bar" (HCons 12.3 HNil))
```

Sadly, this seems completely useless

```
printHList :: HList -> IO ()
printHList HNil = return ()
printHList (HCons x xs)
  = do putStrLn (show x)
       printHList xs
```

```
No instance for (Show a)
      arising from a use of `show'
...
```

Existential Types

There is no direct way through this problem; the value of the existentially quantified type can never escape.

But we can go *around* the problem, if we know what the operation is we wanted to perform.

What if we changed the definition of HList a bit so that it had the operations on our "hidden" data values bundled within it?

Wrap the operations and the values in a way that allow us to confirm they are safe

Existential Types

```
data HList = HNil
           | forall a . HCons (a, a -> String) HList
```

```
f = HCons ("foo",id) (HCons ("bar",id)
          (HCons (12.3,show) HNil))
```

```
printHList :: HList -> IO ()
printHList HNil              = return ()
printHList (HCons (x,s) xs) = do putStrLn (s x)
                                 printHList xs
```

Existential Types

Instead of explicitly supplying a function we can also say the hidden type must be an instance of some class, which can make the definitions simpler.

```
data HList = HNil
           | forall a . Show a => HCons a HList
```

```
f = HCons "foo" (HCons "bar" (HCons 12.3 HNil))
```

```
printHList :: HList -> IO ()
printHList HNil          = return ()
printHList (HCons x xs) = do putStrLn (show x)
                             printHList xs
```

Existential Types

This approach has often been used to model the notion of *Private fields* in object-like type systems

- **Create a data type with named fields and existential values**
- **Create a type class for the methods**

You can't have "." for structure access but lots of people define an infix "#" operation instead.

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 3

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 10, Part 4
Type system extensions (3)

CS4012

Topics in Functional Programming
Michaelmas Term 2020

Glenn Strong <glenn.Strong@scss.tcd.ie>

Generalised Algebraic Datatypes

We have seen several extensions to the Hindley-Milner system in the past week or two.

- **One more, which subsumes some of these approaches, is the notion of a *Generalized Algebraic Data Type***
- **This is really just a data type where we declare the types of the constructors directly, giving us freedom to vary the constructor function signatures.**

Sometimes called indexed data types

Generalised Algebraic Datatypes

When you declare a data type you normally write:

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

Creating constructor functions:

```
Left  :: a -> Either a b
Right :: b -> Either a b
```

Generalised Algebraic Datatypes

Using the GADTs extension we can declare the same structure like this:

```
data Either a b where
     Left  :: a -> Either a b
     Right :: b -> Either a b
```

So what?

Apart from illustrating that data constructors really are just functions this doesn't seem to offer much

Generalised Algebraic Datatypes

The power comes from being able to *restrict* the type variables in the constructors.

As an example I will develop a type for lists that track their length at the type level.

First, I will need a type to represent numbers

```
data Zero
data Succ n
```

I mean, of course, Peano numbers.

Since I only plan to use these in types, and will never make values, I need not define constructors!

Generalised Algebraic Datatypes

Now, if I create a list type, declared as a GADT, with a length parameter which I will restrict…

```
data List a n where
  Nil  :: List a Zero
  Cons :: a -> List a n -> List a (Succ n)
```

With this definition I can write a "safe" function to take the head of a list:

```
hd :: List a (Succ n) -> a
hd (Cons a _) = a
```

Generalised Algebraic Datatypes

There is a price to pay for this ability to restrict our functions. Every function must have an expressible type!

```
f 0 = Nil
f 1 = Cons 1 Nil
```

There is no possible type for this list that correctly captures the possible lengths!

And this is what we asked for when we defined the List type the way we did.

Generalised Algebraic Datatypes

Motivating example

The classic small GADT example is a type-safe interpreter

Earlier this year we built a small interpreter. Stripping down the expression language a little, it looked a bit like this:

```
data Val = I Int | B Bool

data Expr = Const Val
     | Add Expr Expr | Sub Expr Expr …
     | Eq Expr Expr | Gt Expr Expr | …
   deriving (Eq, Show)
```

The expression evaluator had to have a kind of dynamic typing build in

Generalised Algebraic Datatypes

Motivating example

The expression evaluator had to have a kind of dynamic typing build in. Something like:

```
eval (Add e0 e1) =
    do e0' <- eval e0
       e1' <- eval e1
       case (e0', e1') of
         (I i0, I i1) -> return $ I (i0 + i1)
         _            -> throwError "type error“
```

Boo, hiss

Generalised Algebraic Datatypes

Motivating example

One way to eliminate that would be to lock down the types in the construction of the expression, something like:

```
data Expr a where
   N :: Int -> Expr Int
   B :: Bool -> Expr Bool
   Add  :: Expr Int ->  Expr Int ->  Expr Int
   Mult :: Expr Int ->  Expr Int ->  Expr Int
```

Expressions of fixed types fit well into this approach:

```
   Or :: Expr Bool ->  Expr Bool ->  Expr Bool
```

We can add expressions with mixed types, and even polymorphic types

```
   If :: Expr Bool ->  Expr a -> Expr a -> Expr a
```

Generalised Algebraic Datatypes

Motivating example

One possible head-scratching moment is how to handle an "equality" test. Should it be

```
Eq :: Expr Int ->  Expr Int ->  Expr Bool
```

or

```
Eq :: Expr Bool ->  Expr Bool ->  Expr Bool
```

This can't be right (why?):

```
Eq :: Expr a ->  Expr a ->  Expr Bool
```

Using type classes is (yet again) the way to go:

```
Eq :: Eq a => Expr a -> Expr a ->  Expr Bool
```

Generalised Algebraic Datatypes

Motivating example

That would lead us to an evaluator that is type-safe!

```
eval :: Expr a -> a
eval (N x)          = x
eval (B x)          = x
eval (Add e0 e1)    = eval e0 + eval e1
eval (Mult e0 e1) = eval e0 * eval e1
eval (Eq a b) = eval a == eval b
eval (If c t e) = if (eval c) then (eval t) else (eval e)
```

Of course, we have only moved the problem of incorrect expressions around, now it's the case that we must *build* the expressions in a safe way.

The analogy here is moving problems from run-time to compile-time.

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 3

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 10, Part 5
Type system extensions (4)

CS4012

Topics in Functional Programming
Michaelmas Term 2020

Glenn Strong <glenn.Strong@scss.tcd.ie>

Data Kinds

There is one more extension in Haskell that I want to talk about

With this extension to the type system Haskell lets us talk about the *kind* of a type — that is, the type of a type.

Here are some examples of kinds for things we're familiar with:

```
Int     :: *
Char    :: *
[Int]   :: *
Maybe   :: * -> *
[]      :: * -> *
StateT :: * -> (* -> *) -> * -> *
```

In other words, the *kind* tells us how many type arguments need to be supplied to produce a type.
Haskell has the Kind "*" built in.

Data Kinds

When using the DataKinds extension Haskell has:

- **A new *type* for each data constructor**
- **A new *kind* for each data type**

We can use these with GADTs to further constrain the use data constructors in a type

Data Kinds

In our previous attempt to define a list type which tracked lengths we stopped with a "head" operation. Why not move on to produce a function that concatenated lists too?

The answer is that we would need to write a function with a type something like this:

```
append :: List a x -> List a y -> List a (x+y)
```

Data Kinds

Using data kinds and type families we can actually achieve this!

```
data Nat where
  Zero :: Nat
  Succ :: Nat -> Nat
```

I want a type that has constructors for this (remember, DataKinds means there is a new *type* for each *constructor).*

So this declaration introduces:

- **A type `Nat` (of Kind '*') and**
- **Two constructors: `Zero :: Nat` and `Succ :: Nat → Nat`**
- **a Kind `Nat` and two new types: `Zero` and `Succ`, of Kind `Nat`**

Data Kinds

Then use this in the same way that we did with GADTs

```
data List a (l :: Nat) where
  Nil :: List a Zero
  Cons :: a -> List a n -> List a (Succ n)
```

My type variable "l" now gets an annotation saying it must be of Kind "Nat". This is going to matter shortly!

Anyway, now I can create a sample list:

```
mylist ::  List Integer (Succ (Succ (Succ Zero)))
mylist = 1 `Cons` (2 `Cons` (3 `Cons` Nil))
```

Data Kinds

I know how to write my "Appending" function:

```
append (Cons x xs) ys = x `Cons` (append xs ys)
append Nil         ys = ys
```

But I need to supply a type. This is quite beyond the power of the inference system!

```
Main.hs:23:25: error:
    • Couldn't match expected type ‘p1’
                  with actual type ‘List a ('Succ n0)’
        ‘p1’ is untouchable
          inside the constraints: l ~ 'Succ n
          bound by a pattern with constructor:
                     Cons :: forall a (n :: Nat). a -> List a n -> List a ('Succ n),
                   in an equation for ‘append’
          at Main.hs:23:9-17
      ‘p1’ is a rigid type variable bound by
        the inferred type of append :: List a l -> p -> p1
        at Main.hs:(23,1)-(24,26)
      Possible fix: add a type signature for ‘append’
    • In the expression: x `Cons` (append xs ys)
      In an equation for ‘append’:
          append (Cons x xs) ys = x `Cons` (append xs ys)
    • Relevant bindings include
        xs :: List a n (bound at Main.hs:23:16)
        x :: a (bound at Main.hs:23:14)
        append :: List a l -> p -> p1 (bound at Main.hs:23:1)
```

Data Kinds

What we need is a way to *calculate* the type of the resulting list.

In theory this is easy; the length component is the *sum* of the input lengths.

The TypeFamilies extension allows us to tell the compiler how to produce new types of certain Kinds. This is type-level programming!

You can think of it as being akin to class definitions in Haskell; where classes define sets of functions associated with types, type families define sets of types associated with classes

Data Kinds

First we name the family of types we are declaring

```
type family Add (x :: Nat) (y :: Nat) :: Nat
```

Next we describe how to build a specific type according to the pattern described in the family (in this case, how to make a type of Kind "Nat" when given two other types of kind "Nat").

```
type instance Add Zero y = y
type instance Add (Succ x) y = Succ (Add x y)
```

That leaves us able to state a type for our append function:

```
append :: List a x -> List a y -> List a (Add x y)
```

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 5

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thank you

glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/