Solving the expression problem in Haskell

Sean Seefried

Australian Government
Department of Communications, Information Technology and the Arts
Australian Research Council

NICTA Members
NICTA Partners

Hello all. This is some work I did while I was doing my PhD under the supervision of Manuel Chakravarty and Gabi Keller (who are both present in the audience today). But as you can see now I'm working at NICTA. The only background knowledge I'm assuming in this talk is a fairly rudimentary understanding of type classes and how they are implemented with dictionaries. I'll explain everything else in as much detail as is necessary.

What is the expression problem?

NICTA

- It's about ***extensible data types***

- Problem is to extend:
  - variants of a data type
  - methods on that data type
  - without modifying existing code
  - with separate compilation
  - It'd be nice if it was statically typed too

- Plug-in compilers need a solution

[3:00] (2:30)

So, what is the expression problem? Well, the term was first coined by Philip Wadler on the Java genericity mailing list. The problem came up in the context of an interpreter for a small expression-based language, hence the name "expression problem". The problem was fairly simple to express. Say we have a data type, in this case representing expressions. This data type will have a number of variants. In this talk, I'm going to use this word exclusively. In functional languages such as Haskell and OCaml, a variant refers to that thing you create with a specific constructor. In an OO language it's a subclass of the a base class. Naturally, you also have some methods that operate on this data type. (In this talk I'll use the word method and function interchangeably.)

The ***expression problem*** is this: how do we extend the variants *and* the methods without modifying existing code. This amounts to defining the extensions in a new module. Naturally, it should be possible to compile the extensions separately without requiring the source of the other modules. Back in 1998, static typing was something that was considered an added bonus but not entirely necessary. Since I'm solving this problem in Haskell, though, a solution will automatically have this property.

The expression problem is perhaps not the best name. When you think expression problem think "extensible data types".

Back when the expression problem was first posed the reason they wanted to solve this problem was primarily one of good software engineering practice. Going back and modifying source code is tedious and error prone. The old code may have been well-used and the worry is that you'll introduce new bugs. I ***needed*** to solve the expression problem. During my PhD I became interested in the idea of a plug-in compiler, a compiler constructed in such a manner that it could be extended by plug-ins. The basic idea was to expose parts of the compiler, such as the AST and accompanying functions, and allow users to write new functionality. But this is going to require that new functions and new variants can be added to the AST. The central idea of a plug-in compiler is that you don't ***have to*** touch the source of the compiler. In fact, it makes good commercial sense to keep this a trade secret. Solving the expression problem was a necessity.

Functional vs. Objected Oriented

NICTA

- Functional languages
  - easy to add new methods
    - they're just functions
  - impossible to add new variants
    - must modify data type declaration
- Object Oriented
  - easy to add new variants
    - Just subclass
  - impossible to add new methods
    - Must modify base class

[3:30] (0:30)

Functional languages and objected oriented languages actually solve one half of the expression problem, but complementary halves. In functional languages it is easy to add new methods. They're just functions. But to add new variants there is no choice but to go back and change the original data type declaration.

On the other hand in object oriented languages, it's easier to add new variants. Usually you'll define a data type by defining a base class and the sub-classing that base class for each new variant. You can see the problem. You have to go back and modify the base class to add new methods. It's not enough to sub-class the base class and add new methods since that does not allow you to add new methods to the ***existing*** variants.

I haven't written this on the slide but there is something called the Visitor Pattern. This allows you to write new methods on a class simply by passing in a visitor to an object. The downside is that the variants are hard coded inside the visitor method. Basically, you're in the same situation as with functional languages.

What would it look like in HaskellX?

NICTA

Need a way to *open* a data type

```
module Exp where

open data Exp = Var String
              | Lam String Exp
              | App Exp Exp

alpha :: Exp -> (String, String) -> Exp
alpha (Var v) s = swap s v
alpha (Lam name body) ...
alpha (App e1 e2) ...
```


[4:00] (0:30)

Let's imagine a new Haskell, HaskellX, that provides support for extensible data types. It has a new "open data" keyword that allows you to declare a data type which can later be extended.

We can then write a number a number of functions on this data, as usual.

Our running example in this talk is the pure, un-typed **lambda calculus**. These three constructors introduce our first three *variants*: variables, lambda abstractions and applications.

Our first function on this data type is alpha conversion -- taking a lambda expression and replacing all occurrences of one variable name with another. This one is dead simple and does not make a distinction between bound and free variables.

I'm only going to show the body of the Var equation here.

What would it look like in Haskell?

NICTA

And *extend* a data type

```
module Eval where

extend data Exp = Let String Exp Exp

alpha (Let name exp body) ...

eval :: Exp -> Env -> Exp
eval (Var x) = ...
eval (Lam name body) = ...
eval (App e1 e2) = ...
eval (Let name exp body) = ...

type Env = [(String, Exp)]
```


[4:30] (0:30)

In a new module, called Eval, we extend the lambda calculus with let-expressions using a new "extend data" keyword.

First we introduce the missing equation on the Let variant for the alpha function. In HaskellX it is an error to redefine an equation on an existing variant.

Next we introduce a function to evaluate lambda expressions using beta-reduction.

So that's it for HaskellX. In the rest of the talk I'm going to try to show you how we can encode this behaviour using existing features of Haskell. We could use this as the basis for an implementation of HaskellX, or more immediately, as an idiomatic way of programming extensible data types. I already have, in a plug-in compiler called PHRaC.

Now I wonder what feature of Haskell could be used to encode open data types?

Features of the solution

- Type classes
  - only candidate because they're ***open***
  - need to be ***multi-parameter***
- Existential types
  - Allow dynamic dispatch, *a la Läufer.*
- Recursive Dictionaries
  - Introducing ***retrospective super-classing***
- Plus:
  - scoped type variables
  - kind annotations

[5:30](1:00)

Okay, the answer is pretty obvious. It's Type Classes since they're the only declaration form in Haskell that is open. But equally as important are existential types which allow us to provide a kind of dynamic dispatch quite similar to the way it occurs in objected oriented languages. We also use an experimental feature introduced in GHC 6.4 called recursive dictionaries. I'll explain them in full when we come to them. We use a trick, due to John Hughes, but which I coined "retrospective super-classing". In short, they allow you to specify the super-class of a class *after (*i.e. in another module) you've declared the class!

So, these are the big three things that make my solution possible. We also need a few other things such as scoped type variables and kind annotations. But for fairly boring reasons.

Läufer's method

```
data Exp = forall b. Alpha b => MkExp b

data Exp_0 = Var String | ... | App Exp Exp

class Alpha b where
  alpha :: b -> (String, String) -> Exp

instance Alpha Exp_0 where
  alpha (Var v) = MkExp (Var (swap s v))
  ...
  alpha (App e1 e2) = ...

instance Alpha Exp where
  alpha (MkExp e) = alpha e
```


[9:00](2:00)

My technique is an extension of Konstantin Läufer's dynamic dispatch method. I'll spend a short time explaining how it works.

CLICK

This is called the wrapper type. It is an existentially quantified type that encapsulates functionality for *component types.* For historical reasons you introduce them with a ***forall*** keyword which is downright confusing. What's really cool about existential types is that you can encapsulate a class context within them, in this case "Alpha b". What actually happens is that the ***dictionary*** is also encapsulated. In fact, if we didn't include the class context then it would not be possible to apply any methods to values wrapped by the **MkExp** constructor.

CLICK

Exp_0 I call a component type. A uniform interface to all component types is kept by making them all instances of....

CLICK

... the *functionality class.* It defines the functionality for the wrapper type. Look carefully at the type here though. Notice that where we used to have the *first* occurrence of the Exp type we now have B. That makes sense. But notice that the second occurrence is the wrapper type. Hey, you might want to call some other function on the result, right?

CLICK

A component instance then provides the functionality for a given component type. I've decided again to just show the body of the equation on the Var variant. Notice that we've added the MkExp constructor in order to wrap it up.

CLICK

And now for the final piece of the puzzle -- the unwrapping instance. You're never going to see component types; they're always wrapped up. The right hand side makes sense because the dictionary for the component type accompanies it (inside the existential wrapper type).

Läufer's method

NICTA

```
data Exp = forall b. Alpha b => MkExp b

data Exp_0 = Var String | ... | App Exp Exp

class Alpha b where
  alpha :: b -> (String, String) -> Exp

instance Alpha Exp_0 where
  alpha (Var v) = MkExp (Var (swap s v))
  ...
  alpha (App e1 e2) = ...

instance Alpha Exp where
  alpha (MkExp e) = alpha e
```

Wrapper Type

[9:00](2:00)

My technique is an extension of Konstantin Läufer's dynamic dispatch method. I'll spend a short time explaining how it works.

CLICK

This is called the wrapper type. It is an existentially quantified type that encapsulates functionality for *component types.* For historical reasons you introduce them with a ***forall*** keyword which is downright confusing. What's really cool about existential types is that you can encapsulate a class context within them, in this case "Alpha b". What actually happens is that the ***dictionary*** is also encapsulated. In fact, if we didn't include the class context then it would not be possible to apply any methods to values wrapped by the **MkExp** constructor.

CLICK

Exp_0 I call a component type. A uniform interface to all component types is kept by making them all instances of....

CLICK

... the *functionality class.* It defines the functionality for the wrapper type. Look carefully at the type here though. Notice that where we used to have the *first* occurrence of the Exp type we now have B. That makes sense. But notice that the second occurrence is the wrapper type. Hey, you might want to call some other function on the result, right?

CLICK

A component instance then provides the functionality for a given component type. I've decided again to just show the body of the equation on the Var variant. Notice that we've added the MkExp constructor in order to wrap it up.

CLICK

And now for the final piece of the puzzle -- the unwrapping instance. You're never going to see component types; they're always wrapped up. The right hand side makes sense because the dictionary for the component type accompanies it (inside the existential wrapper type).

Läufer's method

```
data Exp = forall b. Alpha b => MkExp b

data Exp_0 = Var String | ... | App Exp Exp

class Alpha b where
  alpha :: b -> (String, String) -> Exp

instance Alpha Exp_0 where
  alpha (Var v) = MkExp (Var (swap s v))
  ...
  alpha (App e1 e2) = ...

instance Alpha Exp where
  alpha (MkExp e) = alpha e
```

Wrapper Type

Component Type

[9:00](2:00)

My technique is an extension of Konstantin Läufer's dynamic dispatch method. I'll spend a short time explaining how it works.

CLICK

This is called the wrapper type. It is an existentially quantified type that encapsulates functionality for *component types.* For historical reasons you introduce them with a ***forall*** keyword which is downright confusing. What's really cool about existential types is that you can encapsulate a class context within them, in this case "Alpha b". What actually happens is that the ***dictionary*** is also encapsulated. In fact, if we didn't include the class context then it would not be possible to apply any methods to values wrapped by the **MkExp** constructor.

CLICK

Exp_0 I call a component type. A uniform interface to all component types is kept by making them all instances of....

CLICK

... the *functionality class.* It defines the functionality for the wrapper type. Look carefully at the type here though. Notice that where we used to have the *first* occurrence of the Exp type we now have B. That makes sense. But notice that the second occurrence is the wrapper type. Hey, you might want to call some other function on the result, right?

CLICK

A component instance then provides the functionality for a given component type. I've decided again to just show the body of the equation on the Var variant. Notice that we've added the MkExp constructor in order to wrap it up.

CLICK

And now for the final piece of the puzzle -- the unwrapping instance. You're never going to see component types; they're always wrapped up. The right hand side makes sense because the dictionary for the component type accompanies it (inside the existential wrapper type).

Läufer's method

```
data Exp = forall b. Alpha b => MkExp b

data Exp_0 = Var String | ... | App Exp Exp

class Alpha b where
  alpha :: b -> (String, String) -> Exp

instance Alpha Exp_0 where
  alpha (Var v) = MkExp (Var (swap s v))
  ...
  alpha (App e1 e2) = ...

instance Alpha Exp where
  alpha (MkExp e) = alpha e
```

Wrapper Type

Component Type

Functionality Class

[9:00](2:00)

My technique is an extension of Konstantin Läufer's dynamic dispatch method. I'll spend a short time explaining how it works.

CLICK

This is called the wrapper type. It is an existentially quantified type that encapsulates functionality for *component types.* For historical reasons you introduce them with a ***forall*** keyword which is downright confusing. What's really cool about existential types is that you can encapsulate a class context within them, in this case "Alpha b". What actually happens is that the ***dictionary*** is also encapsulated. In fact, if we didn't include the class context then it would not be possible to apply any methods to values wrapped by the **MkExp** constructor.

CLICK

Exp_0 I call a component type. A uniform interface to all component types is kept by making them all instances of....

CLICK

... the *functionality class.* It defines the functionality for the wrapper type. Look carefully at the type here though. Notice that where we used to have the *first* occurrence of the Exp type we now have B. That makes sense. But notice that the second occurrence is the wrapper type. Hey, you might want to call some other function on the result, right?

CLICK

A component instance then provides the functionality for a given component type. I've decided again to just show the body of the equation on the Var variant. Notice that we've added the MkExp constructor in order to wrap it up.

CLICK

And now for the final piece of the puzzle -- the unwrapping instance. You're never going to see component types; they're always wrapped up. The right hand side makes sense because the dictionary for the component type accompanies it (inside the existential wrapper type).

Läufer's method

```
data Exp = forall b. Alpha b => MkExp b

data Exp_0 = Var String | ... | App Exp Exp

class Alpha b where
  alpha :: b -> (String, String) -> Exp

instance Alpha Exp_0 where
  alpha (Var v) = MkExp (Var (swap s v))
  ...
  alpha (App e1 e2) = ...

instance Alpha Exp where
  alpha (MkExp e) = alpha e
```

Wrapper Type

Component Type

Functionality Class

Functionality Instance

[9:00](2:00)

My technique is an extension of Konstantin Läufer's dynamic dispatch method. I'll spend a short time explaining how it works.

CLICK

This is called the wrapper type. It is an existentially quantified type that encapsulates functionality for *component types.* For historical reasons you introduce them with a ***forall*** keyword which is downright confusing. What's really cool about existential types is that you can encapsulate a class context within them, in this case "Alpha b". What actually happens is that the ***dictionary*** is also encapsulated. In fact, if we didn't include the class context then it would not be possible to apply any methods to values wrapped by the **MkExp** constructor.

CLICK

Exp_0 I call a component type. A uniform interface to all component types is kept by making them all instances of....

CLICK

... the *functionality class.* It defines the functionality for the wrapper type. Look carefully at the type here though. Notice that where we used to have the *first* occurrence of the Exp type we now have B. That makes sense. But notice that the second occurrence is the wrapper type. Hey, you might want to call some other function on the result, right?

CLICK

A component instance then provides the functionality for a given component type. I've decided again to just show the body of the equation on the Var variant. Notice that we've added the MkExp constructor in order to wrap it up.

CLICK

And now for the final piece of the puzzle -- the unwrapping instance. You're never going to see component types; they're always wrapped up. The right hand side makes sense because the dictionary for the component type accompanies it (inside the existential wrapper type).

Läufer's method

```
data Exp = forall b. Alpha b => MkExp b
```

Wrapper Type

```
data Exp_0 = Var String | ... | App Exp Exp
```

Component Type

```
class Alpha b where
  alpha :: b -> (String, String) -> Exp
```

Functionality Class

```
instance Alpha Exp_0 where
  alpha (Var v) = MkExp (Var (swap s v))
  ...
  alpha (App e1 e2) = ...
```

Functionality Instance

```
instance Alpha Exp where
  alpha (MkExp e) = alpha e
```

Unwrapping Instance

[9:00](2:00)

My technique is an extension of Konstantin Läufer's dynamic dispatch method. I'll spend a short time explaining how it works.

CLICK

This is called the wrapper type. It is an existentially quantified type that encapsulates functionality for *component types.* For historical reasons you introduce them with a ***forall*** keyword which is downright confusing. What's really cool about existential types is that you can encapsulate a class context within them, in this case "Alpha b". What actually happens is that the ***dictionary*** is also encapsulated. In fact, if we didn't include the class context then it would not be possible to apply any methods to values wrapped by the **MkExp** constructor.

CLICK

Exp_0 I call a component type. A uniform interface to all component types is kept by making them all instances of....

CLICK

... the *functionality class.* It defines the functionality for the wrapper type. Look carefully at the type here though. Notice that where we used to have the *first* occurrence of the Exp type we now have B. That makes sense. But notice that the second occurrence is the wrapper type. Hey, you might want to call some other function on the result, right?

CLICK

A component instance then provides the functionality for a given component type. I've decided again to just show the body of the equation on the Var variant. Notice that we've added the MkExp constructor in order to wrap it up.

CLICK

And now for the final piece of the puzzle -- the unwrapping instance. You're never going to see component types; they're always wrapped up. The right hand side makes sense because the dictionary for the component type accompanies it (inside the existential wrapper type).

Can we extend the variants?

```
data Exp_1 = Let String Exp Exp

instance Alpha Exp_1 where
  alpha (Let name exp body) = ...
```


[9:30](0:30)

Can we extend the variants? Yeah, sure. We just introduce a new component type and a new functionality instance.

Looks plausible doesn't it? Well it has some problems which we'll only really see when we consider methods.

Can we extend the methods?

[10:30](1:30)

Type classes have inheritance. It seems plausible that we should be able to add new methods this way. Here I've declared class Eval has inheriting the behaviour of class Alpha. Seems like it might work.

CLICK

But look carefully at the result type? It's an Exp and we know that the only context that's bound up inside that wrapper type is that of class Alpha. You might want to use the result of this in a further call to "eval" and that's clearly not going to work.

CLICK

We can try to get around this by introducing a new wrapper type, ExpE, which encapsulates the Eval class in its context. But unfortunately this doesn't work. I'll show you why.

CLICK

Here's an invocation of eval. **CLICK**. Notice it has a return type of ExpE. All well and good. **CLICK.** But now look what happens when we try to try to apply "alpha" to it. **CLICK.** It just doesn't work. alpha works on Exp`s not ExpE`s

CLICK

If anything like this is going to work we're going to need just one wrapper type for all time. Somehow we need to make do with just one class context inside the wrapper type.

Can we extend the methods?

```
class Alpha b => Eval b where
   eval :: b -> Env -> Exp
```


[10:30](1:30)

Type classes have inheritance. It seems plausible that we should be able to add new methods this way. Here I've declared class Eval has inheriting the behaviour of class Alpha. Seems like it might work.

CLICK

But look carefully at the result type? It's an Exp and we know that the only context that's bound up inside that wrapper type is that of class Alpha. You might want to use the result of this in a further call to "eval" and that's clearly not going to work.

CLICK

We can try to get around this by introducing a new wrapper type, ExpE, which encapsulates the Eval class in its context. But unfortunately this doesn't work. I'll show you why.

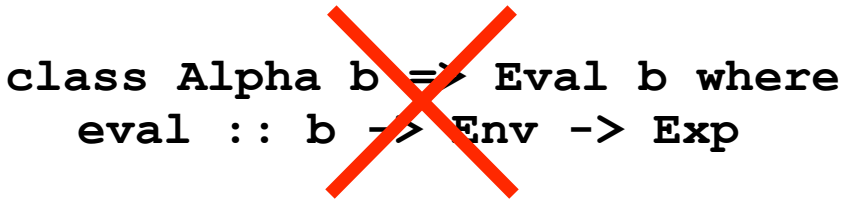
CLICK

Here's an invocation of eval. **CLICK**. Notice it has a return type of ExpE. All well and good. **CLICK.** But now look what happens when we try to try to apply "alpha" to it. **CLICK.** It just doesn't work. alpha works on Exp`s not ExpE`s

CLICK

If anything like this is going to work we're going to need just one wrapper type for all time. Somehow we need to make do with just one class context inside the wrapper type.

Can we extend the methods?



[10:30](1:30)

Type classes have inheritance. It seems plausible that we should be able to add new methods this way. Here I've declared class Eval has inheriting the behaviour of class Alpha. Seems like it might work.

CLICK

But look carefully at the result type? It's an Exp and we know that the only context that's bound up inside that wrapper type is that of class Alpha. You might want to use the result of this in a further call to "eval" and that's clearly not going to work.

CLICK

We can try to get around this by introducing a new wrapper type, ExpE, which encapsulates the Eval class in its context. But unfortunately this doesn't work. I'll show you why.

CLICK

Here's an invocation of eval. **CLICK**. Notice it has a return type of ExpE. All well and good. **CLICK.** But now look what happens when we try to try to apply "alpha" to it. **CLICK.** It just doesn't work. alpha works on Exp`s not ExpE`s

CLICK

If anything like this is going to work we're going to need just one wrapper type for all time. Somehow we need to make do with just one class context inside the wrapper type.

Can we extend the methods?

```
class Alpha b => Eval b where
   eval :: b -> Env -> ExpE

data ExpE = forall b. Eval b => MkExpE b
```


[10:30](1:30)

Type classes have inheritance. It seems plausible that we should be able to add new methods this way. Here I've declared class Eval has inheriting the behaviour of class Alpha. Seems like it might work.

CLICK

But look carefully at the result type? It's an Exp and we know that the only context that's bound up inside that wrapper type is that of class Alpha. You might want to use the result of this in a further call to "eval" and that's clearly not going to work.

CLICK

We can try to get around this by introducing a new wrapper type, ExpE, which encapsulates the Eval class in its context. But unfortunately this doesn't work. I'll show you why.

CLICK

Here's an invocation of eval. **CLICK**. Notice it has a return type of ExpE. All well and good. **CLICK.** But now look what happens when we try to try to apply "alpha" to it. **CLICK.** It just doesn't work. alpha works on Exp`s not ExpE`s

CLICK

If anything like this is going to work we're going to need just one wrapper type for all time. Somehow we need to make do with just one class context inside the wrapper type.

Can we extend the methods?

```
class Alpha b => Eval b where
   eval :: b -> Env -> ExpE

data ExpE = forall b. Eval b => MkExpE b
```

Let's see it in action

```
let exp = (MkExpE (Lam "x" (MkExp (Var "x"))))
in        eval exp []
```


[10:30](1:30)

Type classes have inheritance. It seems plausible that we should be able to add new methods this way. Here I've declared class Eval has inheriting the behaviour of class Alpha. Seems like it might work.

CLICK

But look carefully at the result type? It's an Exp and we know that the only context that's bound up inside that wrapper type is that of class Alpha. You might want to use the result of this in a further call to "eval" and that's clearly not going to work.

CLICK

We can try to get around this by introducing a new wrapper type, ExpE, which encapsulates the Eval class in its context. But unfortunately this doesn't work. I'll show you why.

CLICK

Here's an invocation of eval. **CLICK**. Notice it has a return type of ExpE. All well and good. **CLICK.** But now look what happens when we try to try to apply "alpha" to it. **CLICK.** It just doesn't work. alpha works on Exp`s not ExpE`s

CLICK

If anything like this is going to work we're going to need just one wrapper type for all time. Somehow we need to make do with just one class context inside the wrapper type.

Can we extend the methods?

```
class Alpha b => Eval b where
   eval :: b -> Env -> ExpE

data ExpE = forall b. Eval b => MkExpE b
```

Let's see it in action

```
let exp = (MkExpE (Lam "x" (MkExp (Var "x"))))
in        eval exp []  :: ExpE
```


[10:30](1:30)

Type classes have inheritance. It seems plausible that we should be able to add new methods this way. Here I've declared class Eval has inheriting the behaviour of class Alpha. Seems like it might work.

CLICK

But look carefully at the result type? It's an Exp and we know that the only context that's bound up inside that wrapper type is that of class Alpha. You might want to use the result of this in a further call to "eval" and that's clearly not going to work.

CLICK

We can try to get around this by introducing a new wrapper type, ExpE, which encapsulates the Eval class in its context. But unfortunately this doesn't work. I'll show you why.

CLICK

Here's an invocation of eval. **CLICK**. Notice it has a return type of ExpE. All well and good. **CLICK.** But now look what happens when we try to try to apply "alpha" to it. **CLICK.** It just doesn't work. alpha works on Exp`s not ExpE`s

CLICK

If anything like this is going to work we're going to need just one wrapper type for all time. Somehow we need to make do with just one class context inside the wrapper type.

Can we extend the methods?

```
class Alpha b => Eval b where
   eval :: b -> Env -> ExpE

data ExpE = forall b. Eval b => MkExpE b
```

Let's see it in action

```
let exp = (MkExpE (Lam "x" (MkExp (Var "x"))))
in alpha (eval exp [] ) ("x","y")
```


[10:30](1:30)

Type classes have inheritance. It seems plausible that we should be able to add new methods this way. Here I've declared class Eval has inheriting the behaviour of class Alpha. Seems like it might work.

CLICK

But look carefully at the result type? It's an Exp and we know that the only context that's bound up inside that wrapper type is that of class Alpha. You might want to use the result of this in a further call to "eval" and that's clearly not going to work.

CLICK

We can try to get around this by introducing a new wrapper type, ExpE, which encapsulates the Eval class in its context. But unfortunately this doesn't work. I'll show you why.

CLICK

Here's an invocation of eval. **CLICK**. Notice it has a return type of ExpE. All well and good. **CLICK.** But now look what happens when we try to try to apply "alpha" to it. **CLICK.** It just doesn't work. alpha works on Exp`s not ExpE`s

CLICK

If anything like this is going to work we're going to need just one wrapper type for all time. Somehow we need to make do with just one class context inside the wrapper type.

Can we extend the methods?

NICTA

```
class Alpha b => Eval b where
   eval :: b -> Env -> ExpE

data ExpE = forall b. Eval b => MkExpE b
```

Let's see it in action

```
let exp = (MkExpE (Lam "x" (MkExp (Var "x"))))
in alpha (eval exp [] ) ("x","y")
```

Type Error

[10:30](1:30)

Type classes have inheritance. It seems plausible that we should be able to add new methods this way. Here I've declared class Eval has inheriting the behaviour of class Alpha. Seems like it might work.

CLICK

But look carefully at the result type? It's an Exp and we know that the only context that's bound up inside that wrapper type is that of class Alpha. You might want to use the result of this in a further call to "eval" and that's clearly not going to work.

CLICK

We can try to get around this by introducing a new wrapper type, ExpE, which encapsulates the Eval class in its context. But unfortunately this doesn't work. I'll show you why.

CLICK

Here's an invocation of eval. **CLICK**. Notice it has a return type of ExpE. All well and good. **CLICK.** But now look what happens when we try to try to apply "alpha" to it. **CLICK.** It just doesn't work. alpha works on Exp`s not ExpE`s

CLICK

If anything like this is going to work we're going to need just one wrapper type for all time. Somehow we need to make do with just one class context inside the wrapper type.

Can we extend the methods?

NICTA

```
class Alpha b => Eval b where
   eval :: b -> Env -> ExpE

data ExpE = forall b. Eval b => MkExpE b
```

Let's see it in action

```
let exp = (MkExpE (Lam "x" (MkExp (Var "x"))))
in alpha (eval exp [] ) ("x","y")
```

Type Error

If only we could just have one wrapper type for all time!

[10:30](1:30)

Type classes have inheritance. It seems plausible that we should be able to add new methods this way. Here I've declared class Eval has inheriting the behaviour of class Alpha. Seems like it might work.

CLICK

But look carefully at the result type? It's an Exp and we know that the only context that's bound up inside that wrapper type is that of class Alpha. You might want to use the result of this in a further call to "eval" and that's clearly not going to work.

CLICK

We can try to get around this by introducing a new wrapper type, ExpE, which encapsulates the Eval class in its context. But unfortunately this doesn't work. I'll show you why.

CLICK

Here's an invocation of eval. **CLICK**. Notice it has a return type of ExpE. All well and good. **CLICK.** But now look what happens when we try to try to apply "alpha" to it. **CLICK.** It just doesn't work. alpha works on Exp`s not ExpE`s

CLICK

If anything like this is going to work we're going to need just one wrapper type for all time. Somehow we need to make do with just one class context inside the wrapper type.

Time to fantasise

NICTA

[12:00](1:30)

It's time to fantasise. What if it were possible to declare a class like so. **POINT TO IT.** These declarations look pretty similar to what we had before but they (**CLICK)** have a new "cxt" (short for ***context***) parameter that appears in a number of places. But look closer, this isn't valid Haskell. **CLICK.** This is not a type parameter, it's a ***class parameter****,* a way to abstract over classes, something that Haskell does not have. But let's pretend that it does.

CLICK

Look. This expression type checks. **[POINT TO IT]** The cxt parameter has been filled with Eval. This expression type checks because the Eval class is a super-class of Alpha now and hence the eval method is available.

CLICK

The inheritance can, and in fact should be, bi-directional. Inheritance in the forward direction is useful, because as usual, you might want to define extended functionality in terms of existing functionality. Notice it's present in the declaration of class Eval. **POINT TO IT.**

CLICK

Well it turns out this is not complete fantasy. John Hughes came up with a method that encodes what I've just shown you. I call it retrospective super-classing, because you decide what is a super-class after the fact.

Time to fantasise

NICTA

What if we could do this?

```
class cxt b => Alpha cxt b where
   alpha :: b -> (String, String) -> Exp cxt

data Exp cxt = forall b. Alpha cxt b => MkExp b

class Alpha cxt b => Eval b where
   eval :: b -> Env Eval -> Exp Eval
```


[12:00](1:30)

It's time to fantasise. What if it were possible to declare a class like so. **POINT TO IT.** These declarations look pretty similar to what we had before but they (**CLICK)** have a new "cxt" (short for ***context***) parameter that appears in a number of places. But look closer, this isn't valid Haskell. **CLICK.** This is not a type parameter, it's a ***class parameter****,* a way to abstract over classes, something that Haskell does not have. But let's pretend that it does.

CLICK

Look. This expression type checks. **[POINT TO IT]** The cxt parameter has been filled with Eval. This expression type checks because the Eval class is a super-class of Alpha now and hence the eval method is available.

CLICK

The inheritance can, and in fact should be, bi-directional. Inheritance in the forward direction is useful, because as usual, you might want to define extended functionality in terms of existing functionality. Notice it's present in the declaration of class Eval. **POINT TO IT.**

CLICK

Well it turns out this is not complete fantasy. John Hughes came up with a method that encodes what I've just shown you. I call it retrospective super-classing, because you decide what is a super-class after the fact.

Time to fantasise

NICTA

What if we could do this?

```
class cxt b => Alpha cxt b where
   alpha :: b -> (String, String) -> Exp cxt

data Exp cxt = forall b. Alpha cxt b => MkExp b

class Alpha cxt b => Eval b where
   eval :: b -> Env Eval -> Exp Eval
```


[12:00](1:30)

It's time to fantasise. What if it were possible to declare a class like so. **POINT TO IT.** These declarations look pretty similar to what we had before but they (**CLICK)** have a new "cxt" (short for ***context***) parameter that appears in a number of places. But look closer, this isn't valid Haskell. **CLICK.** This is not a type parameter, it's a ***class parameter****,* a way to abstract over classes, something that Haskell does not have. But let's pretend that it does.

CLICK

Look. This expression type checks. **[POINT TO IT]** The cxt parameter has been filled with Eval. This expression type checks because the Eval class is a super-class of Alpha now and hence the eval method is available.

CLICK

The inheritance can, and in fact should be, bi-directional. Inheritance in the forward direction is useful, because as usual, you might want to define extended functionality in terms of existing functionality. Notice it's present in the declaration of class Eval. **POINT TO IT.**

CLICK

Well it turns out this is not complete fantasy. John Hughes came up with a method that encodes what I've just shown you. I call it retrospective super-classing, because you decide what is a super-class after the fact.

Time to fantasise

NICTA

What if we could do this?

```
class cxt b => Alpha cxt b where
   alpha :: b -> (String, String) -> Exp cxt

data Exp cxt = forall b. Alpha cxt b => MkExp b

class Alpha cxt b => Eval b where
   eval :: b -> Env Eval -> Exp Eval
```


[12:00](1:30)

It's time to fantasise. What if it were possible to declare a class like so. **POINT TO IT.** These declarations look pretty similar to what we had before but they (**CLICK)** have a new "cxt" (short for ***context***) parameter that appears in a number of places. But look closer, this isn't valid Haskell. **CLICK.** This is not a type parameter, it's a ***class parameter****,* a way to abstract over classes, something that Haskell does not have. But let's pretend that it does.

CLICK

Look. This expression type checks. **[POINT TO IT]** The cxt parameter has been filled with Eval. This expression type checks because the Eval class is a super-class of Alpha now and hence the eval method is available.

CLICK

The inheritance can, and in fact should be, bi-directional. Inheritance in the forward direction is useful, because as usual, you might want to define extended functionality in terms of existing functionality. Notice it's present in the declaration of class Eval. **POINT TO IT.**

CLICK

Well it turns out this is not complete fantasy. John Hughes came up with a method that encodes what I've just shown you. I call it retrospective super-classing, because you decide what is a super-class after the fact.

Time to fantasise

NICTA

What if we could do this?

```
class cxt b => Alpha cxt b where
   alpha :: b -> (String, String) -> Exp cxt

data Exp cxt = forall b. Alpha cxt b => MkExp b

class Alpha cxt b => Eval b where
   eval :: b -> Env Eval -> Exp Eval
```

Then this would type check!

```
eval (MkExp (Var "x") :: Exp Eval) []
```


[12:00](1:30)

It's time to fantasise. What if it were possible to declare a class like so. **POINT TO IT.** These declarations look pretty similar to what we had before but they (**CLICK)** have a new "cxt" (short for ***context***) parameter that appears in a number of places. But look closer, this isn't valid Haskell. **CLICK.** This is not a type parameter, it's a ***class parameter****,* a way to abstract over classes, something that Haskell does not have. But let's pretend that it does.

CLICK

Look. This expression type checks. **[POINT TO IT]** The cxt parameter has been filled with Eval. This expression type checks because the Eval class is a super-class of Alpha now and hence the eval method is available.

CLICK

The inheritance can, and in fact should be, bi-directional. Inheritance in the forward direction is useful, because as usual, you might want to define extended functionality in terms of existing functionality. Notice it's present in the declaration of class Eval. **POINT TO IT.**

CLICK

Well it turns out this is not complete fantasy. John Hughes came up with a method that encodes what I've just shown you. I call it retrospective super-classing, because you decide what is a super-class after the fact.

Time to fantasise

NICTA

What if we could do this?

```
class cxt b => Alpha cxt b where
   alpha :: b -> (String, String) -> Exp cxt

data Exp cxt = forall b. Alpha cxt b => MkExp b

class Alpha cxt b => Eval b where
   eval :: b -> Env Eval -> Exp Eval
```

Then this would type check!

```
eval (MkExp (Var "x") :: Exp Eval) []
```

Inheritance is bi-directional

[12:00](1:30)

It's time to fantasise. What if it were possible to declare a class like so. **POINT TO IT.** These declarations look pretty similar to what we had before but they (**CLICK)** have a new "cxt" (short for ***context***) parameter that appears in a number of places. But look closer, this isn't valid Haskell. **CLICK.** This is not a type parameter, it's a ***class parameter****,* a way to abstract over classes, something that Haskell does not have. But let's pretend that it does.

CLICK

Look. This expression type checks. **[POINT TO IT]** The cxt parameter has been filled with Eval. This expression type checks because the Eval class is a super-class of Alpha now and hence the eval method is available.

CLICK

The inheritance can, and in fact should be, bi-directional. Inheritance in the forward direction is useful, because as usual, you might want to define extended functionality in terms of existing functionality. Notice it's present in the declaration of class Eval. **POINT TO IT.**

CLICK

Well it turns out this is not complete fantasy. John Hughes came up with a method that encodes what I've just shown you. I call it retrospective super-classing, because you decide what is a super-class after the fact.

Time to fantasise

NICTA

What if we could do this?

```
class cxt b => Alpha cxt b where
   alpha :: b -> (String, String) -> Exp cxt

data Exp cxt = forall b. Alpha cxt b => MkExp b

class Alpha cxt b => Eval b where
   eval :: b -> Env Eval -> Exp Eval
```

Then this would type check!

```
eval (MkExp (Var "x") :: Exp Eval) []
```

Inheritance is bi-directional

John Hughes came up with something that encodes this.

I call it *retrospective super-classing*

[12:00](1:30)

It's time to fantasise. What if it were possible to declare a class like so. **POINT TO IT.** These declarations look pretty similar to what we had before but they (**CLICK)** have a new "cxt" (short for ***context***) parameter that appears in a number of places. But look closer, this isn't valid Haskell. **CLICK.** This is not a type parameter, it's a ***class parameter****,* a way to abstract over classes, something that Haskell does not have. But let's pretend that it does.

CLICK

Look. This expression type checks. **[POINT TO IT]** The cxt parameter has been filled with Eval. This expression type checks because the Eval class is a super-class of Alpha now and hence the eval method is available.

CLICK

The inheritance can, and in fact should be, bi-directional. Inheritance in the forward direction is useful, because as usual, you might want to define extended functionality in terms of existing functionality. Notice it's present in the declaration of class Eval. **POINT TO IT.**

CLICK

Well it turns out this is not complete fantasy. John Hughes came up with a method that encodes what I've just shown you. I call it retrospective super-classing, because you decide what is a super-class after the fact.

Retrospective super-classing

NICTA

Implicit dictionaries become explicit.

The *Sat* class

```
class Sat a where dict :: a
```

Explicit dictionary declaration

```
data EvalD b =
  EvalD { eval' :: b -> Env EvalD -> Exp EvalD }
```

Functionality classes

```
class Sat (cxt b) => Alpha cxt b where
  alpha :: b -> (String, String) -> Exp cxt

class (Sat (EvalD b), Alpha EvalD b) => Eval b where
  eval :: b -> Env EvalD -> Exp EvalD
```


[13:00](1:00)

The basic trick to retrospective super-classing is to make implicit dictionaries explicit. We introduce a new class, Sat (short for ***satisfies***) which has one method called "dict". An instance of this class serves one purpose, to match up implicit dictionaries with explicit dictionaries, which are just data types we define ourselves.

EvalD is an example of an explicit dictionary. It contains exactly the same methods as class Eval but we name the method with a prime character.

We must now annotate the two functionality classes Alpha and Eval with Sat constraints. This Sat constraint **POINT TO FIRST** should be read as saying, type B satisfies context C-X-T.

We'll now have a look at the instance heads of the functionality instances for data type Exp_0.

Retrospective super-classing

NICTA

Instances for Exp_0

```
instance ( Sat (cxt (Exp   cxt))
         , Sat (cxt (Exp_0 cxt))
         ) => Alpha cxt (Exp_0 cxt) where
  alpha (Var v) = ...

instance ( Sat (EvalD (Exp   EvalD))
         , Sat (EvalD (Exp_0 EvalD))
         ) => Eval (Exp_0 EvalD) where
  eval (Var v) = ...
```

Knot-tying instance. (Creating the explicit dictionary)

```
instance Eval b => Sat (EvalD b) where
  dict = EvalD { eval' = eval }
```


[14:30](1:30)

The instance declaration for Alpha says that any type that unifies with "Exp cxt" or "Exp_0 cxt" satisfies the context C-X-T, or if you want, that class Alpha has super-class C-X-T. You might be wondering why there are two Sat constraints one for the wrapper type, Exp, and one for Exp_0. You might not always need both of these constraints, but in my translation I like to play it safe. The constraint on the Exp_0 type would be required if you wanted to create new values of the data type inside the method. The one on Exp is required because we may be calling alpha recursively and it's in the ***class*** constraints.

Now let's look at the instance head for Eval. Notice that we're saying that type Exp EvalD ***and*** Exp_0 EvalD satisfy the EvalD context.

Finally, we'll have a look how we create the explicit dicitionary. We assign the methods encapsulated in the explicit dictionary to their implicit cousins. I like to think of this instance as "tying the knot" of Sat constraints. I'll have more about to say this on the next slide.

Does this really work?

NICTA

Let's look at it in use

```
let exp = lam "v" (var "v")
          :: Sat (cxt (Exp cxt)) => Exp cxt
in eval' dict exp [] :: Exp EvalD
```

Constraint resolution

Sat (EvalD (Exp EvalD)) from use of method "dict"

⇛ Eval (Exp EvalD) from instance head of knot-tying instance

⇛ Sat (EvalD (Exp EvalD)) from instance head of Eval (Exp_0 EvalD)

This is only possible with *recursive dictionaries*

[16:00](1:30)

Now let's have a look at a sample use. Notice we no longer make a call to "eval" we make a call to eval-prime dict instead. This is necessary so that constraint resolution works properly. I'll now talk about that a little bit. **THIS** constraint is introduced from the use of method dict. If we look at the instance head of the Sat instance **GO BACK ONE SLIDE** we can see that it introduces this constraint **POINT TO SECOND.** This actually introduces a few constraints but one of them is this one here. **POINT TO THIRD.** We are back where we started!

But the fantastic thing is -- this works as long as you are allowed to build recursive dictionaries. Recursive dictionaries are those that are built using recursive functions once dictionary translation has been done. Recursive dictionaries, along with the extension to the type checker necessary to allow them, have been possible since GHC 6.4. They have other useful applications. If you're interested in recursive dictionaries I can explain them in full during question time.

Adding new variant

```
data Exp_1 cxt = Let String (Exp cxt)(Exp cxt)

instance ( Sat (EvalD (Exp   EvalD))
         , Sat (EvalD (Exp_0 EvalD))
         , Sat (EvalD (Exp_1 EvalD))
         ) => Alpha EvalD (Exp_1 EvalD) where
  alpha (Let name body exp) s =
      letE (swap s name) (alpha body s) (alpha exp s)

instance ( Sat (EvalD (Exp   EvalD))
         , Sat (EvalD (Exp_0 EvalD))
         , Sat (EvalD (Exp_1 EvalD))
         ) => Eval (Exp_1 EvalD) where
   eval (Let name body exp) env =
      eval' dict (app (lam name exp) body) env
```


[17:30](1:30)

What I've left out

- Can't extend beyond Eval in this example
- Every new class should have a new cxt parameter for extensions
  - `class (Sat (EvalD cxt b), Alpha (EvalD cxt) b) => Eval cxt b`
- Require a notion of "capping classes"
  - empty extensions that "tie the knot"
- I've got a formal translation
  - from the syntactic sugar to what we've seen.

20 minutes wasn't really enough time to go into much detail about my solution. I've also simplified my solution somewhat.

Related work

- Scala
  - Zenger and Odersky.
    - Makes sense - Scala fuses functional and OO programming.
    - Uses notion of mix-ins. But they're well typed!
- Ocaml
  - polymorphic variants. Jacques Garrigue
- Haskell
  - Löh and Hinze.
    - Not implemented. (Requires "best fit" pattern matching)
    - Wouldn't really work in plug-in setting

NICTA

THE END

Demo available!