A Taste of Programming with Generalised Algebraic Datatypes

Alexey Rodriguez Yakushev

Institute of Information and Computing Sciences Utrecht University

March 26, 2007



The enlightment path to GADTs

- Evaluator with dynamic type checking.
- Well-typed expressions with tag-less evaluator.
- Sized lists and utility functions (*head*, *tail*, *map*, *zipWith*, *replicate*, *transpose*).
- Simply typed lambda calculus

 If you are still alive, check out the work done by the dependent types community and look for more examples. The examples here are based on work by the Epigram and Omega people.



Write an interpreter for the following language:

data UExpr = LitInt Int | LitBool Bool | Inc UExpr | IsZ UExpr | If UExpr UExpr UExpr

Values in the language:

data Val = VBool Bool| VInt Int



Write an interpreter for the following language:

data UExpr = LitInt Int | LitBool Bool | Inc UExpr | IsZ UExpr | If UExpr UExpr UExpr

Values in the language:

data Val = VBool Bool| VInt Int



Interpreter:

▲ロト ▲御 ト ▲ 臣 ト ▲ 臣 ト ○ 臣 - の Q @

```
eval :: UExpr \rightarrow Val

eval (LitInt n) = VInt n

eval (Inc e)

= case eval e of

VInt n \rightarrow VInt (n + 1)

_ \rightarrow error "inc applied to non-int"

eval (IsZ e)

= case eval e of

VInt n \rightarrow VBool (n = 0)

_ \rightarrow error "isZ applied to non-int"
```



- Interpreter:
 - $eval (If c e_1 e_2)$ = case eval c of $VBool b \rightarrow if b$ then $eval e_1$ else $eval e_2$ _ $\rightarrow error$ "if condition is non-bool"
- Evaluation of **if** *isZ* 1 **then** 1 **else** *inc* 2:

eval (If (IsZ (LitInt 1)) (LitInt 1) (Inc (LitInt 2))) $\Rightarrow 3$

• Evaluation of *inc True*:

eval (Inc (LitBool True)) ⇒ error "inc applied to non-int"



Interpreter:

1

$$eval (If c e_1 e_2)$$

= case $eval c$ of
 $VBool b \rightarrow if b$ then $eval e_1$ else $eval e_2$
_ $\rightarrow error$ "if condition is non-bool"

• Evaluation of **if** *isZ* 1 **then** 1 **else** *inc* 2:

eval (If (IsZ (LitInt 1)) (LitInt 1) (Inc (LitInt 2))) $\Rightarrow 3$

• Evaluation of *inc True*:



Interpreter:

```
eval (If c e_1 e_2)
= case eval c of
VBool b \rightarrow if b then eval e_1 else eval e_2
_ \rightarrow error "if condition is non-bool"
```

• Evaluation of **if** *isZ* 1 **then** 1 **else** *inc* 2:

eval (If (IsZ (LitInt 1)) (LitInt 1) (Inc (LitInt 2))) $\Rightarrow 3$

• Evaluation of *inc True*:



Interpreter

▲ロト ▲御 ト ▲ 臣 ト ▲ 臣 ト ○ 臣 - の Q @

- eval is a tag-ful interpreter: it does type checking at runtime by checking the tags of values.
- If the expression is type-correct, the UExpr datatype cannot
- How can we encode well-typedness into the expression





Interpreter

▲ロト ▲御 ト ▲ 臣 ト ▲ 臣 ト ○ 臣 - の Q @

- eval is a tag-ful interpreter: it does type checking at runtime by checking the tags of values.
- If the expression is type-correct, the UExpr datatype cannot express it.
- How can we encode well-typedness into the expression datatype?



Interpreter

- eval is a tag-ful interpreter: it does type checking at runtime by checking the tags of values.
- If the expression is type-correct, the UExpr datatype cannot express it.
- How can we encode well-typedness into the expression datatype?



Expressions of different types can be encoded in different datatypes:

data IExpr = *ILit* Int | *IInc* IExpr | *IIf* (IfExpr IExpr) data BExpr = *BLit* Bool | *BIsZ* IExpr | *BIf* (IfExpr BExpr) data IfExpr a = IfExpr BExpr a a



But now eval becomes several functions:

evallExpr :: IExpr \rightarrow Int evallExpr(ILit n) = nevallExpr(IInc e) = evallExpr e + 1evallExpr(IIf e) = evallExpr(evallf e)*evalBExpr* :: BExpr \rightarrow Bool evalBExpr(BLit b) = bevalBExpr(BlsZ e) = evallExpr e = 0evalBExpr(Blf e) = evalBExpr(evallf e)*evallf* :: If Expr $a \rightarrow a$ evallf (IfExpr c $e_1 e_2$) = if evalBExpr c then e_1 else e_2



- Splitting datatypes splits the evaluator.
- Adding pairs or other datatypes becomes even more tricky (more splitting).
- There are good reasons to keep expressions as one datatype.
 - we would like to have one evaluate function.
 - it is difficult to write a function that parses expressions with split types!
- However we learned two valuable lessons from enforcing well-typedness in the datatype:
 - we can omit dynamic type checking from the evaluator: tag-less evaluation.
 - the evaluator is more likely to be correct due to the more precise types, i.e. the int expression evaluator cannot return a bool!
- Now we'll see how to encode well-typed expressions in GADTs.



- Splitting datatypes splits the evaluator.
- Adding pairs or other datatypes becomes even more tricky (more splitting).
- There are good reasons to keep expressions as one datatype.
 - we would like to have one evaluate function.
 - it is difficult to write a function that parses expressions with split types!
- However we learned two valuable lessons from enforcing well-typedness in the datatype:
 - we can omit dynamic type checking from the evaluator: tag-less evaluation.
 - the evaluator is more likely to be correct due to the more precise types, i.e. the int expression evaluator cannot return a bool!
- Now we'll see how to encode well-typed expressions in GADTs.



- Splitting datatypes splits the evaluator.
- Adding pairs or other datatypes becomes even more tricky (more splitting).
- There are good reasons to keep expressions as one datatype.
 - we would like to have one evaluate function.
 - it is difficult to write a function that parses expressions with split types!
- However we learned two valuable lessons from enforcing well-typedness in the datatype:
 - we can omit dynamic type checking from the evaluator: tag-less evaluation.
 - the evaluator is more likely to be correct due to the more precise types, i.e. the int expression evaluator cannot return a bool!
- Now we'll see how to encode well-typed expressions in GADTs.



- Splitting datatypes splits the evaluator.
- Adding pairs or other datatypes becomes even more tricky (more splitting).
- There are good reasons to keep expressions as one datatype.
 - we would like to have one evaluate function.
 - it is difficult to write a function that parses expressions with split types!
- However we learned two valuable lessons from enforcing well-typedness in the datatype:
 - we can omit dynamic type checking from the evaluator: tag-less evaluation.
 - the evaluator is more likely to be correct due to the more precise types, i.e. the int expression evaluator cannot return a bool!
- Now we'll see how to encode well-typed expressions in GADTs.



- Splitting datatypes splits the evaluator.
- Adding pairs or other datatypes becomes even more tricky (more splitting).
- There are good reasons to keep expressions as one datatype.
 - we would like to have one evaluate function.
 - it is difficult to write a function that parses expressions with split types!
- However we learned two valuable lessons from enforcing well-typedness in the datatype:
 - we can omit dynamic type checking from the evaluator: tag-less evaluation.
 - the evaluator is more likely to be correct due to the more precise types, i.e. the int expression evaluator cannot return a bool!
- Now we'll see how to encode well-typed expressions in GADTs.

Generalised algebraic datatypes

From Peyton-Jones, Vytiniotis and Weirich 2006:

data TExpr a **where** *TLitInt* :: Int \rightarrow TExpr Int *TIsZ* :: TExpr Int \rightarrow TExpr Bool *TIf* :: TExpr Bool \rightarrow TExpr a \rightarrow TExpr a

- On the declaration syntax:
- Constructors are given type signatures, like functions.
- In the signature, the arguments are constructor fields.
- Constructors may restrict the type arguments in the return type.



Generalised algebraic datatypes

From Peyton-Jones, Vytiniotis and Weirich 2006:

data TExpr a **where** *TLitInt* :: Int \rightarrow TExpr Int *TIsZ* :: TExpr Int \rightarrow TExpr Bool *TIf* :: TExpr Bool \rightarrow TExpr a \rightarrow TExpr a

On the declaration syntax:

▲ロト ▲御 ト ▲ 臣 ト ▲ 臣 ト ○ 臣 - の Q @

- Constructors are given type signatures, like functions.
- In the signature, the arguments are constructor fields.
- Constructors may restrict the type arguments in the return type.





Generalised algebraic datatypes

From Peyton-Jones, Vytiniotis and Weirich 2006:

data TExpr a **where** *TLitInt* :: Int \rightarrow TExpr Int *TIsZ* :: TExpr Int \rightarrow TExpr Bool *TIf* :: TExpr Bool \rightarrow TExpr a \rightarrow TExpr a

- On the declaration syntax:
- Constructors are given type signatures, like functions.
- In the signature, the arguments are constructor fields.
- Constructors may restrict the type arguments in the return type.



From Peyton-Jones, Vytiniotis and Weirich 2006:

data TExpr a **where** *TLitInt* :: Int \rightarrow TExpr Int *TIsZ* :: TExpr Int \rightarrow TExpr Bool *TIf* :: TExpr Bool \rightarrow TExpr a \rightarrow TExpr a

- ► Here, the argument to TExpr is called a type index.
- The type index tells something about the constructors that conform the value.
- At the same time the constructors use indices to restrict the values that can be plugged in as fields.
- In this case TExpr represents only well-typed expressions (modulo \product values).



From Peyton-Jones, Vytiniotis and Weirich 2006:

data TExpr a **where** *TLitInt* :: Int \rightarrow TExpr Int *TIsZ* :: TExpr Int \rightarrow TExpr Bool *TIf* :: TExpr Bool \rightarrow TExpr a \rightarrow TExpr a \rightarrow TExpr a

- ► Here, the argument to TExpr is called a type index.
- The type index tells something about the constructors that conform the value.
- At the same time the constructors use indices to restrict the values that can be plugged in as fields.
- In this case TExpr represents only well-typed expressions (modulo \(\triangle values).



From Peyton-Jones, Vytiniotis and Weirich 2006:

data TExpr a **where** *TLitInt* :: Int \rightarrow TExpr Int *TIsZ* :: TExpr Int \rightarrow TExpr Bool *TIf* :: TExpr Bool \rightarrow TExpr a \rightarrow TExpr a \rightarrow TExpr a

- Here, the argument to TExpr is called a type index.
- The type index tells something about the constructors that conform the value.
- At the same time the constructors use indices to restrict the values that can be plugged in as fields.
- In this case TExpr represents only well-typed expressions (modulo \product values).



From Peyton-Jones, Vytiniotis and Weirich 2006:

data TExpr a **where** *TLitInt* :: Int \rightarrow TExpr Int *TIsZ* :: TExpr Int \rightarrow TExpr Bool *TIf* :: TExpr Bool \rightarrow TExpr a \rightarrow TExpr a \rightarrow TExpr a

- Here, the argument to TExpr is called a type index.
- The type index tells something about the constructors that conform the value.
- At the same time the constructors use indices to restrict the values that can be plugged in as fields.
- In this case TExpr represents only well-typed expressions (modulo \product values).



Interpreter for well-typed expressions

 A tag-less interpreter from Peyton-Jones, Vytiniotis and Weirich 2006:

> $evalT :: TExpr a \rightarrow a$ evalT (TLitInt i) = i evalT (TIsZ e) = evalT e = 0 $evalT (TIf c e_1 e_2) = if evalT c then evalT e_1 else evalT e_2$

- Here's how typing works: the TExpr a in the signature is refined to the constructor return type in every arm.
- ► So, for *TIsZ* the right hand side must return TExpr Bool rather than TExpr a.
- The type checking algorithm requires a type signature for the function.



Interpreter for well-typed expressions

 A tag-less interpreter from Peyton-Jones, Vytiniotis and Weirich 2006:

> $evalT :: TExpr a \rightarrow a$ evalT (TLitInt i) = i evalT (TIsZ e) = evalT e = 0 $evalT (TIf c e_1 e_2) = if evalT c then evalT e_1 else evalT e_2$

- Here's how typing works: the TExpr a in the signature is refined to the constructor return type in every arm.
- ► So, for *TlsZ* the right hand side must return TExpr Bool rather than TExpr a.
- The type checking algorithm requires a type signature for the function.



Interpreter for well-typed expressions

 A tag-less interpreter from Peyton-Jones, Vytiniotis and Weirich 2006:

> $evalT :: TExpr a \rightarrow a$ evalT (TLitInt i) = i evalT (TIsZ e) = evalT e = 0 $evalT (TIf c e_1 e_2) = if evalT c then evalT e_1 else evalT e_2$

- Here's how typing works: the TExpr a in the signature is refined to the constructor return type in every arm.
- ► So, for *TlsZ* the right hand side must return TExpr Bool rather than TExpr a.
- The type checking algorithm requires a type signature for the function.



More GADT examples

- In this lecture we will see GADT examples that:
 - don't need function cases that are ruled out by datatype properties.
 - are more reliable in the sense that functions cannot break datatype properties.
- The examples that we will see are:





More GADT examples

- In this lecture we will see GADT examples that:
 - don't need function cases that are ruled out by datatype properties.
 - are more reliable in the sense that functions cannot break datatype properties.
- The examples that we will see are:
 - Sized lists.
 - Simply typed lambda calculus.





Sized lists

- Sized lists have a type index that gives the number of elements.
- We first define type-level naturals to count elements:

data Zero data Suc a

► Sized lists:

▲ロト ▲御 ト ▲ 臣 ト ▲ 臣 ト ○ 臣 - の Q @

data List a sz where Nil :: List a Zero Cons :: a → List a sz → List a (Suc sz)

Universiteit Utrecht



Sized lists

- Sized lists have a type index that gives the number of elements.
- We first define type-level naturals to count elements:

data Zero data Suc a

Sized lists:

▲ロト ▲御 ト ▲ 臣 ト ▲ 臣 ト ○ 臣 - の Q @

data List a sz where Nil :: List a Zero Cons :: a → List a sz → List a (Suc sz)

Universiteit Utrecht



Sized lists

- Sized lists have a type index that gives the number of elements.
- We first define type-level naturals to count elements:

data Zero data Suc a

Sized lists:

data List a sz where *Nil* :: List a Zero *Cons* :: $a \rightarrow List a sz \rightarrow List a (Suc sz)$



Sized lists examples

- Examples:
 - ex1 = Nil:: List a Zero $ex2 = Cons \ 1 \ Nil$:: List Int (Suc Zero) ex3 = Cons 1 (Cons 2 Nil) :: List Int (Suc (Suc Zero))





head and tail

・ロト・西・・田・・田・ 日・ シック

We can make *head* and *tail* total because empty lists are not part of their domain:

> head :: List a (Suc sz) \rightarrow a head (Cons x xs) = x tail :: List a (Suc sz) \rightarrow List a sz tail (Cons x xs) = xs

If you go ahead and give a case for Nil:

head Nil = error "impossible"

the compiler will complain:

Inaccessible case alternative: Can't match types 'Zero' and 'Suc n' In the pattern: Nil In the definition of 'head': head Nil - error "impossible"

Universiteit Utrecht



head and tail

We can make *head* and *tail* total because empty lists are not part of their domain:

> head :: List a (Suc sz) \rightarrow a head (Cons x xs) = x tail :: List a (Suc sz) \rightarrow List a sz tail (Cons x xs) = xs

If you go ahead and give a case for Nil:

head Nil = error "impossible"

the compiler will complain:

Inaccessible case alternative: Can't match types 'Zero' and 'Suc n' In the pattern: Nil In the definition of 'head': head Nil = error "impossible"


map and zipWith

The type guarantees that the result list has the same length as the initial one.

 $\begin{array}{l} \textit{map} :: (a \rightarrow b) \rightarrow \textit{List a sz} \rightarrow \textit{List b sz} \\ \textit{map f Nil} = \textit{Nil} \\ \textit{map f (Cons x xs)} = \textit{Cons (f x) (map f xs)} \end{array}$

 For *zipWith* the type makes the case for different lengths unnecessary.

 $\begin{aligned} zipWith :: (a \to b \to c) \to List \ a \ sz \to List \ b \ sz \to List \ c \ sz \\ zipWith \ f \ Nil \ Nil \\ = Nil \\ zipWith \ f \ (Cons \ x \ xs) \ (Cons \ y \ ys) \\ = Cons \ (f \ x \ y) \ (zipWith \ f \ xs \ ys) \end{aligned}$



map and zipWith

The type guarantees that the result list has the same length as the initial one.

$$map :: (a \rightarrow b) \rightarrow List \ a \ sz \rightarrow List \ b \ sz$$
$$map \ f \ Nil = Nil$$
$$map \ f \ (Cons \ x \ xs) = Cons \ (f \ x) \ (map \ f \ xs)$$

 For *zipWith* the type makes the case for different lengths unnecessary.

 $\begin{aligned} zipWith :: (a \to b \to c) \to List \ a \ sz \to List \ b \ sz \to List \ c \ sz \\ zipWith \ f \ Nil \ Nil \\ = Nil \\ zipWith \ f \ (Cons \ x \ xs) \ (Cons \ y \ ys) \\ = Cons \ (f \ x \ y) \ (zipWith \ f \ xs \ ys) \end{aligned}$



replicate

For replicate we will need value level naturals

replicate :: RNat $sz \rightarrow a \rightarrow List a sz$

so we define:

lata RNat *n* **where** *RZero* :: RNat Zero *RSuc* :: RNat sz → RNat (Suc sz)

And now we are able to write *replicate* down:

replicate RZero x = Nilreplicate (RSuc n) x = Cons x (replicate n x)



replicate

For replicate we will need value level naturals

replicate :: RNat $sz \rightarrow a \rightarrow List a sz$

so we define:

data RNat *n* where *RZero* :: RNat Zero *RSuc* :: RNat sz \rightarrow RNat (Suc sz)

And now we are able to write *replicate* down:

replicate RZero x = Nilreplicate (RSuc n) x = Cons x (replicate n x)



replicate

For replicate we will need value level naturals

replicate :: RNat $sz \rightarrow a \rightarrow List a sz$

so we define:

data RNat *n* where *RZero* :: RNat Zero *RSuc* :: RNat sz \rightarrow RNat (Suc sz)

And now we are able to write *replicate* down:

replicate RZero x = Nilreplicate (RSuc n) x = Cons x (replicate n x)



The type of transpose (drawing):

transpose :: List (List a cols) rows \rightarrow List (List a rows) cols

• The base case is tricky, transposing 0×3 gives 3×0 .

transpose Nil = replicate ____ Nil

So we need to pass the number of columns as an argument:

transpose :: RNat cols → List (List a cols) rows → List (List a rows) cols transpose ncols Nil = replicate ncols Nil transpose ncols (Cons xs xss) = zipWith Cons xs (transpose ncols xss)



The type of transpose (drawing):

transpose :: List (List a cols) rows \rightarrow List (List a rows) cols

• The base case is tricky, transposing 0×3 gives 3×0 .

transpose Nil = replicate ____ Nil

So we need to pass the number of columns as an argument:

transpose :: RNat cols → List (List a cols) rows → List (List a rows) cols transpose ncols Nil = replicate ncols Nil transpose ncols (Cons xs xss) = zipWith Cons xs (transpose ncols xss)



The type of transpose (drawing):

transpose :: List (List a cols) rows \rightarrow List (List a rows) cols

• The base case is tricky, transposing 0×3 gives 3×0 .

transpose Nil = replicate ____ Nil

So we need to pass the number of columns as an argument:

 $\begin{array}{l} transpose :: \ \text{RNat cols} \\ \rightarrow \ \text{List} \ (\text{List a cols}) \ \text{rows} \rightarrow \ \text{List} \ (\text{List a rows}) \ \text{cols} \\ transpose \ ncols \ \text{Nil} = \ replicate \ ncols \ \text{Nil} \\ transpose \ ncols \ (Cons \ xs \ xss) \\ = \ zipWith \ Cons \ xs \ (transpose \ ncols \ xss) \end{array}$



- We can avoid passing the number of columns explicitly in two ways:
 - write a transpose that requires non-zero dimensions.
 - use a type-class Nat that reifies type naturals to the value level, use a method *reifyNat* :: Nat $sz \Rightarrow RNat sz$.
- These are exercises for you. Since you have to code this,



- We can avoid passing the number of columns explicitly in two ways:
 - write a transpose that requires non-zero dimensions.
 - use a type-class Nat that reifies type naturals to the value level, use a method *reifyNat* :: Nat sz ⇒ RNat sz.
- These are exercises for you. Since you have to code this, you might as well continue with matrix multiplication.

Suppose we want to obtain sized lists from normal lists, as follows:

```
toSized :: [a] \rightarrow List a sz
```

What is sz supposed to be?!?!

A non-solution is to assume that sz is universally quantified. But this is wrong! Because

toSized [] :: List Int (Suc Zero)

is now well typed!

The solution is to use an existential quantified variable to say that we just don't know what sz is at compile time:

```
toSized :: [a] \rightarrow (\exists sz . List a sz)
```



Suppose we want to obtain sized lists from normal lists, as follows:

```
toSized :: [a] \rightarrow List a sz
```

What is sz supposed to be?!?!

A non-solution is to assume that sz is universally quantified. But this is wrong! Because

toSized [] :: List Int (Suc Zero)

is now well typed!

The solution is to use an existential quantified variable to say that we just don't know what sz is at compile time:

toSized :: [a] \rightarrow (\exists sz . List a sz)



▲□▶▲圖▶▲≣▶▲≣▶ / 重 / のへで

Suppose we want to obtain sized lists from normal lists, as follows:

```
toSized :: [a] \rightarrow List a sz
```

What is sz supposed to be?!?!

A non-solution is to assume that sz is universally quantified. But this is wrong! Because

toSized [] :: List Int (Suc Zero)

is now well typed!

The solution is to use an existential quantified variable to say that we just don't know what sz is at compile time:

toSized :: $[a] \rightarrow (\exists sz . List a sz)$



With GHC, we can only specify existential variables in data types:

> data ToSizedRes a where $ToSizedRes :: List a sz \rightarrow ToSizedRes a$

ToSizedRes wraps a sized list and "forgets" the size of it. Now we can define *toSized*.

 $toSized :: [a] \rightarrow ToSizedRes a$ toSized [] = ToSizedRes Nil toSized (x : xs) = case toSized xs of $ToSizedRes ls \rightarrow ToSizedRes (Cons x ls)$

But now, what is the point of having a sized list and not knowing the size?

 $\begin{array}{l} \textit{funnyHead} :: [a] \rightarrow a \\ \textit{funnyHead Is} \\ = \textbf{case} \ \textit{toSized Is of} \\ \textit{ToSizedRes sls} \rightarrow \textit{head sls} \ \ \text{-- type error!!!} \end{array}$

- This problem with existentials is solved in two ways:
 - wrapping something that knows what to do with the existential, or
 - wrapping something that tells us what the existential is, i.e. a witness of the existential.
- We follow the second solution.

Universiteit Utrecht 🛔



◆□ > ◆母 > ◆臣 > ◆臣 > ○臣 ● のへで

But now, what is the point of having a sized list and not knowing the size?

funnyHead :: $[a] \rightarrow a$ funnyHead Is = case toSized Is of ToSizedRes sIs \rightarrow head sIs -- type error!!!

- This problem with existentials is solved in two ways:
 - wrapping something that knows what to do with the existential, or
 - wrapping something that tells us what the existential is, i.e. a witness of the existential.
- We follow the second solution.

But now, what is the point of having a sized list and not knowing the size?

funnyHead :: $[a] \rightarrow a$ funnyHead Is = case toSized Is of ToSizedRes sIs \rightarrow head sIs -- type error!!!

- This problem with existentials is solved in two ways:
 - wrapping something that knows what to do with the existential, or
 - wrapping something that tells us what the existential is, i.e. a witness of the existential.
- We follow the second solution.

ToSizedRes packages a witness of the list size.

data ToSizedRes a where *ToSizedRes* :: RNat sz \rightarrow List a sz \rightarrow ToSizedRes a *toSized* :: $[a] \rightarrow$ ToSizedRes a toSized [] = ToSizedRes RZero Nil toSized (x : xs)= case to Sized xs of ToSizedRes wsz ls \rightarrow ToSizedRes (RSuc wsz) (Cons x ls)





Now we use the witness:

funnyHead :: [a] → a funnyHead Is = case toSized Is of ToSizedRes (RSuc _) sIs → head sIs _ → error "funnyHead []"

The witness seems redudant in this case, but if you have something more complex than a list, it starts to pay off.



Now we use the witness:

funnyHead :: [a] → a funnyHead Is = case toSized Is of ToSizedRes (RSuc _) sIs → head sIs _ → error "funnyHead []"

The witness seems redudant in this case, but if you have something more complex than a list, it starts to pay off.



- Challenge: Implement a UExpr to *TTerm* converter (untyped to well-typed expressions).
- Here a witness for the existential will be even more useful.





 append is a challenge because it would seem that its signature requires evaluation at the type level.

append :: List a $sz_1 \rightarrow List a sz_2 \rightarrow List a (plus sz_1 sz_2)$

But *plus* is not definable in Haskell (unless you use some serious type class hackery).

However, we can encode this using existential types and a witness that acts as evidence (proof) that sz₃ = sz₁ + sz₂:

> data Plus sz₁ sz₂ sz₃ where *PlusBase* :: Plus Zero sz₂ sz₂ *PlusStep* :: Plus sz₁ sz₂ sz₃ \rightarrow Plus (Suc sz₁) sz₂ (Suc sz₃)



 append is a challenge because it would seem that its signature requires evaluation at the type level.

```
append :: List a sz_1 \rightarrow List a sz_2 \rightarrow List a (plus sz_1 sz_2)
```

But *plus* is not definable in Haskell (unless you use some serious type class hackery).

However, we can encode this using existential types and a witness that acts as evidence (proof) that sz₃ = sz₁ + sz₂:

> data Plus $sz_1 sz_2 sz_3$ where *PlusBase* :: Plus Zero $sz_2 sz_2$ *PlusStep* :: Plus $sz_1 sz_2 sz_3$ \rightarrow Plus (Suc sz_1) sz_2 (Suc sz_3)



 append returns the concatenated list and the evidence that it has the size of the two input lists combined.

> **data** AppendRes a $sz_1 sz_2$ where *AppendRes* :: Plus $sz_1 sz_2 sz_3 \rightarrow List a sz_3$ \rightarrow AppendRes a $sz_1 sz_2$

The definition of append:

append :: List a $sz_1 \rightarrow List$ a $sz_2 \rightarrow AppendRes a <math>sz_1 sz_2$ append Nilys = AppendRes PlusBase ysappend (Cons x xs) ys= case append xs ys ofAppendRes proof res $\rightarrow AppendRes (PlusStep proof) (Cons x res)$



 append returns the concatenated list and the evidence that it has the size of the two input lists combined.

> **data** AppendRes a $sz_1 sz_2$ **where** *AppendRes* :: Plus $sz_1 sz_2 sz_3 \rightarrow List a sz_3$ \rightarrow AppendRes a $sz_1 sz_2$

The definition of append:

append :: List a $sz_1 \rightarrow List a sz_2 \rightarrow AppendRes a sz_1 sz_2$ append Nil ys = AppendRes PlusBase ysappend (Cons x xs) ys = case append xs ys of AppendRes proof res $\rightarrow AppendRes (PlusStep proof) (Cons x res)$

Write an interpreter for the following language:

data Expr = Var String | App Expr Expr | Abs String Expr

and many constants and functions.

- The same problem as before, the evaluator must distinguish between functions from non-functions and from constants of different types.
- As before, we eliminate runtime type checking encoding only well-typed terms.



Write an interpreter for the following language:

data Expr = Var String | App Expr Expr | Abs String Expr

and many constants and functions.

- The same problem as before, the evaluator must distinguish between functions from non-functions and from constants of different types.
- As before, we eliminate runtime type checking encoding only well-typed terms.



Write an interpreter for the following language:

data Expr = Var String | App Expr Expr | Abs String Expr

and many constants and functions.

- The same problem as before, the evaluator must distinguish between functions from non-functions and from constants of different types.
- As before, we eliminate runtime type checking encoding only well-typed terms.



▲ロト ▲御 ト ▲ 臣 ト ▲ 臣 ト ○ 臣 - の Q @

We say that the expression has type t under an environment env

data Expr env t where

The evaluation will require an environment of an appropriate type:

evaluate :: Env env \rightarrow Expr env t \rightarrow t



▲ロト ▲御 ト ▲ 臣 ト ▲ 臣 ト ○ 臣 - の Q @

Let us see pseudo-types for a few example expressions:

$$x$$
:: Expr ((x :: a) : env) a x y :: Expr ((x :: $a \rightarrow b$) : (y :: b) : env) b $(\lambda x . x)$:: Expr env $(a \rightarrow a)$

Replacing names by (de Bruijn) indices we get:

$$\begin{array}{ll} 0 & :: \operatorname{Expr}(a, \operatorname{env}) & a \\ 0 & 1 & :: \operatorname{Expr}(a \to b, (b, \operatorname{env})) & b \\ (\lambda \cdot 0) & :: \operatorname{Expr}\operatorname{env} & (a \to a) \end{array}$$



▲ロト ▲御 ト ▲ 臣 ト ▲ 臣 ト ▲ 臣 - の Q ()

Let us see pseudo-types for a few example expressions:

$$x$$
:: Expr ((x :: a) : env) a x y :: Expr ((x :: $a \rightarrow b$) : (y :: b) : env) b $(\lambda x . x)$:: Expr env $(a \rightarrow a)$

Replacing names by (de Bruijn) indices we get:

$$\begin{array}{ll} 0 & :: \operatorname{Expr}\left(a, \operatorname{env}\right) & a \\ 0 & 1 & :: \operatorname{Expr}\left(a \to b, (b, \operatorname{env})\right) b \\ (\lambda \cdot 0) & :: \operatorname{Expr}\operatorname{env} & (a \to a) \end{array}$$



Variables with type t under an environment env

```
data Var env t where

VFirst :: Var (a, env) a

VNext :: Var env a \rightarrow Var (b, env) a
```

Environments (heterogeneous list):

lata Env env **where** *EEmpty* :: Env () *EExt* :: $a \rightarrow Env env \rightarrow Env (a, env)$

And variable lookup:



Variables with type t under an environment env

```
data Var env t where

VFirst :: Var (a, env) a

VNext :: Var env a \rightarrow Var (b, env) a
```

Environments (heterogeneous list):

```
data Env env where

EEmpty :: Env ()

EExt :: a \rightarrow Env env \rightarrow Env (a, env)
```

And variable lookup:



Variables with type t under an environment env

```
data Var env t where

VFirst :: Var (a, env) a

VNext :: Var env a \rightarrow Var (b, env) a
```

Environments (heterogeneous list):

data Env env where EEmpty :: Env () $EExt :: a \rightarrow Env env \rightarrow Env (a, env)$

And variable lookup:

 $\begin{array}{ll} \textit{lookup} :: Var env a \rightarrow Env env \rightarrow a \\ \textit{lookup VFirst} & (EExt x es) = x \\ \textit{lookup (VNext v) (EExt x es)} = \textit{lookup v es} \\ & \text{Universiteit Utrecht} \end{array}$



Well-typed expressions

data Expr env t where

 $\begin{array}{l} \textit{EVar} :: \text{Var env } t \rightarrow \text{Expr env } t \\ \textit{EApp} :: \text{Expr env } (t_1 \rightarrow t_2) \rightarrow \text{Expr env } t_1 \rightarrow \text{Expr env } t_2 \\ \textit{EAbs} :: \text{Expr } (a, \text{env}) \ b \rightarrow \text{Expr env } (a \rightarrow b) \\ \textit{ECon} :: a \rightarrow \text{Expr env } a \end{array}$

Evaluator:

 $evaluate :: Env env \rightarrow Expr env t \rightarrow t$ evaluate env (EVar v) = lookup v env $evaluate env (EApp e_1 e_2) = (evaluate env e_1)$ $(evaluate env e_2)$ $evaluate env (EAbs e) = \lambda x \rightarrow evaluate (EExt x env) e$ evaluate env (ECon c) = cUniversiteit Utrecht

Well-typed expressions

data Expr env t **where** $EVar :: Var env t \rightarrow Expr env t$ $EApp :: Expr env (t_1 \rightarrow t_2) \rightarrow Expr env t_1 \rightarrow Expr env t_2$ $EAbs :: Expr (a, env) b \rightarrow Expr env (a \rightarrow b)$ $ECon :: a \rightarrow Expr env a$

Evaluator:

 $\begin{array}{ll} evaluate :: Env env \rightarrow Expr env t \rightarrow t \\ evaluate env (EVar v) &= lookup v env \\ evaluate env (EApp e_1 e_2) &= (evaluate env e_1) \\ & (evaluate env e_2) \\ evaluate env (EAbs e) &= \lambda x \rightarrow evaluate (EExt x env) e \\ evaluate env (ECon c) &= c \\ & &$
Conclusions

- Increased reliability: functions cannot break GADT-encoded properties.
- Properties not totally guaranteed: we can fake properties with ⊥, and patterns are not properly checked for exhaustiveness.
- Efficiency problems: Sometimes proof objects have to be inspected at runtime.
- Existential syntax: GADT programs need a lot of them, it might be time to revisit the "boxed existentials" design choice.
- When is it useful for real programs? When you want to be (almost) sure that a core part of your system is correct. I have used them in such a way.

Universiteit Utrecht

