Type families and data kinds

AFP Summer School

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Today

- How do GADTs work?
- Kinds beyond *
- Programming with types

Calling functions on vectors

Given two vectors `xs : Vec a n` and `ys : Vec a m`.

Suppose I want to zip these vectors together using:

```
zipVec :: Vec a n -> Vec b n -> Vec (a,b) n
```

Question

What happens when I call `zipVec xs ys`?

Calling functions on vectors

Given two vectors `xs : Vec a n` and `ys : Vec a m`.

Suppose I want to zip these vectors together using:

```
zipVec :: Vec a n -> Vec b n -> Vec (a,b) n
```

Question

What happens when I call `zipVec xs ys`?
I get a type error: `n` and `m` are not necessarily equal!

Comparing the length of vectors

We can define a boolean function that checks when two vectors have the same length

```
equalLength :: Vec a m -> Vec b n -> Bool
equalLength Nil Nil = True
equalLength (Cons _ xs) (Cons _ ys) =
  equalLength xs ys
equalLength _ _ = False
```

Comparing the length of vectors

Such a function is not very useful…

Suppose I want to use this to check the lengths of my vectors:

```
if equalLength xs ys
  then zipVec xs ys
  else error "Wrong lengths"
```

Question

Will this type check?

Comparing the length of vectors

Such a function is not very useful…

Suppose I want to use this to check the lengths of my vectors:

```
if equalLength xs ys
  then zipVec xs ys
  else error "Wrong lengths"
```

Question

Will this type check?
No! Just because `equalLength xs ys` returns True, does not guarantee that `m` and `n` are equal…
How can we enforce that two types are indeed equal?

Equality type

Just as we saw for the `Sum` type, we can introduce a GADT that represents a 'proof' that two types are equal:

```
data Equal :: * -> * -> * where
  Refl  ::   Equal a a
```

Equality type

We can ‘prove’ properties of our equality relation:

```
refl :: Equal a a
sym :: Equal a b -> Equal b a
trans :: Equal a b -> Equal b c -> Equal a c
```

Equality type

We can ‘prove’ properties of our equality relation:

```
refl :: Equal a a
sym :: Equal a b -> Equal b a
trans :: Equal a b -> Equal b c -> Equal a c
```

Question

How are these functions defined?

Equality type

We can ‘prove’ properties of our equality relation:

```
refl :: Equal a a
sym :: Equal a b -> Equal b a
trans :: Equal a b -> Equal b c -> Equal a c
```

Question

How are these functions defined?
What happens if you don’t pattern match on the `Refl` constructor?

Equality type

Instead of returning a boolean, we can now provide evidence that the length of two vectors is equal:

```
eqLength :: Vec a m -> Vec b n -> Maybe (Equal m n)
eqLength Nil          Nil          = Just Refl
eqLength (Cons x xs)  (Cons y ys)  =
  case eqLength xs ys of
    Just Refl -> Just Refl
    Nothing   -> Nothing
eqLength _            _            = Nothing
```

Using equality

```
test :: Vec a m -> Vec b (Succ n) -> Maybe (a,b)
test xs ys =
  case eqLength xs ys
    Just Refl -> head (zipVec xs ys)
    _         -> Nothing
```

Question

Why does this type check?

Expressive power of equality

The equality type can be used to encode other GADTs.

Recall our expression example using phantom types:

```
data Expr a =
    LitI    Int
  | LitB    Bool
  | IsZero  (Expr Int)
  | Plus    (Expr Int) (Expr Int)
  | If      (Expr Bool) (Expr a) (Expr a)
```

Expressive power of equality

We can use equality proofs and phantom types to ‘implement’ GADTs:

```
data Expr a =
    LitI  (Equal a Int) Int
  | LitB  (Equal a Bool) Bool
  | IsZero (Equal a Bool) (Equal b Int)
  | Plus (Equal a Int) (Expr Int) (Expr Int)
  | If (Expr Bool) (Expr a) (Expr a)
  ...
```

Safe vs unsafe coercions

Using our equality function we can safely coerce between types:

```
coerce :: Equal a b -> a -> b
coerce Refl x = x
```

Question

Why does this type check?

Safe vs unsafe coercions

Using our equality function we can safely coerce between types:

```
coerce :: Equal a b -> a -> b
coerce Refl x = x
```

Question

Why does this type check?

Question

What about this definition:

```
coerce :: Equal a b -> a -> b
coerce p x = x
```

Aside: irrefutable patterns

Haskell also allows irrefutable patterns:

```
lazyHead ~(x:xs) = x
```

This does not force the list to weak head normal form.

Aside: irrefutable patterns

In tandem with GADTs this is particularly dangerous:

```
coerceL :: Equal a b -> a -> b
coerceL ~Refl x = x
```

Question

How could this cause well-typed program to crash with a type error?

Aside: irrefutable patterns

In tandem with GADTs this is particularly dangerous:

```
coerceL :: Equal a b -> a -> b
coerceL ~Refl x = x
```

Question

How could this cause well-typed program to crash with a type error?

```
foo :: Bool -> Int
foo b = coerceL undefined b
```

Apparently unrelated language features may interact in unexpected ways!

System FC

We saw that Haskell's core language, System FC, is a typed lambda calculus, extended with data types and pattern matching.

One of its more distinct features is *coercions* and *casts*.

- Coercions play the same role as our `Equal` data type;
- If two types are coercible, one can be cast to the other:

```
isZero :: (a ~ Int) => a -> Bool
```

There is quite a lot of work necessary to guarantee that this does not accidentally make the type system unsound!

Pattern matching on GADTs introduces such coercions in the individual branches.

Problems with GADTs

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

To define this function, we needed to construct an explicit relation describing how to add two types, `n` and `m`.

Problems with GADTs

```
toVec :: [a] -> Vec a ???
```

To define this function, we needed to reify natural numbers on the type level – defining a *singleton type* `SNat`.

Passing explicit Sums

In Alejandro's lecture, we saw how to pass an explicit argument, explaining how to add two 'type-level' natural numbers:

```
data Sum :: * -> * -> * -> * where
  SumZero :: Sum Zero n n
  SumSucc :: Sum n m s -> Sum (Succ n) m (Succ s)
```

But constructing this evidence by hand is tedious…

Multi-parameter type classes

One way to automate this, is through a *multi-parameter type class*

```
class Summable a b c | a b -> c where
  makeSum :: Sum a b c

instance Summable Zero n n where
  makeSum = SumZero

instance Summable n m s =>
           Summable (Succ n) m (Succ s) where
  makeSum = SumSucc makeSum

append :: Sum n m s =>
            Vec a n -> Vec a m -> Vec a s
append = vappend makeSum
```

Multi-parameter type classes

Type classes define *relations* between types:

- `Eq` defines a subset of all types that support an equality function;
- `MonadState` defines a subset of pairs of types `s` and `m`, where `m` supports read/write operations on a state of type `s`.

The `Summable` type class is special case of such relations – it is really defining a *function* between types.

Multi-parameter type classes

For some time, multi-parameter type classes with functional dependencies were the *only* way in Haskell to define such type-level computations.

But there has been a flurry of research in the last decade exploring alternative language extensions.

```
... the interaction of functional dependencies
with other type-level features such as
existentials and GADTs is not well understood
and possibly problematic.
```

Kiselyov, Peyton Jones, Shan in *Fun with type families*

Associated types and type families

Type classes let you capture an *interface* – such as monads (supporting return and bind), or monoids (supporting zero and addition).

These interaces can describe *functions*.

But what if we would like them to describe *types*.

Associated types

Associated types let you declare a type in a class declaration:

```
class Collects c where
  type Elem c -- Associated type synonym
  empty :: c
  insert :: Elem c -> c -> c
  toList :: c -> [Elem c]
```

Any instance of the `Collects` class must choose a type of elements, together with definitions for the functions.

Associated types – examples

```
instance Eq e => Collects [e] where
  type Elem [e] = e
  empty = []
  ...
instance Collects BitSet where
  type Elem BitSet = Char
  ...
```

Addition through association

We can use such associated types to replace the functional dependencies we saw previously:

```
class Summable n m where
  type TheSum n m
  makeSum :: Sum n m (TheSum n m)

instance Summable Zero n where
  type TheSum Zero m = m
  ...

instance Summable n m =>
  Summable (Succ n) m where
    type TheSum (Succ n) m = Succ (TheSum n m)
    ...
```

Associated types or multiparameter type?

Both approaches are similar in expressive power.

Multiparameter type classes are no longer fashionable – mainly because they can make type class resolution unpredictable.

Associated types have gained traction in other languages – such as Apple's Swift.

Type families

Associated types always require a class definition – even if we're only interested in the types.

Type families build upon the technology that associated types provide, enabling you to write:

```
type family Sum n m
type instance Sum Zero n = n
type instance Sum (Succ n) m = Succ (Sum n m)
```

This looks much more like regular programming…

Type families

If we piggyback on the associated type machinery, however, all our type families are *open* – we can add bogus definitions:

```
type instance Sum n Zero = Zero
```

Furthermore, all our ‘type level’ code is essentially untyped.

```
type instance Sum Bool Int = Char
```

Closed type functions

The more modern *closed type families* allow you to define a function between types using pattern matching:

```
type family Count (f :: *) :: Nat where
  Count (a -> b) = 1 + (Count b)
  Count x = 1
```

GHC will try to match a given type against the patterns one by one, taking the first branch that matches successfully.

This almost lets us program with types almost as if they were regular values.

Kinds

So far we have seen that two different forms of kinds:

- all types have kind *
- given two kinds k1 and k2, we can form the kind k1 -> k2 – corresponding to the type constructor taking something of kind k1 to produce a type of kind k2.

Kinds

So far we have seen that two different forms of kinds:

- all types have kind *
- given two kinds k1 and k2, we can form the kind k1 -> k2 – corresponding to the type constructor taking something of kind k1 to produce a type of kind k2.

This is essentially the simply typed lambda calculus with one base type.

As soon as we do richer programming with types, however, we would like stronger guarantees about the safety of the *type level* computations that we write.

Example

```
data Apply f a = MkApply (f a)
```

Question

What is the kind of `Apply`?

Example

```
data Apply f a = MkApply (f a)
```

Question

What is the kind of `Apply`?
Many different answers exist: `(* -> *) -> * -> *` being the most obvious. But there's no reason that `a` must have kind `*`.
You can make the case for *kind polymorphism* – just as we have polymorphism in types (GHC supports this).

Promotion

```
data Zero
data Succ n

data Nat = Zero | Succ Nat
```

How can we ensure all numbers in our types to be built from `Zero` and `Succ`?

Promotion

Using the `DataKinds` language extension we can introduce new kinds and automatically *promote* data constructors into their type-level variants:

```
{-# LANGUAGE DataKinds #-}

data Nat = Zero | Succ Nat
```

This declaration introduces:

- a new kind `Nat`
- a type `'Zero :: Nat`
- a type `'Succ :: Nat -> Nat`

(This only works for algebraic data types, not for GADTs)

Example: booleans

```
-- the usual definition of booleans
data Bool = True | False

-- Not function on values
not :: Bool -> Bool
not True = False
not False = True

-- Not function on types
type family Not (a :: Bool) :: Bool
type instance Not True = False
type instance Not False = True
```

Type-level literals

GHC takes the idea of programming with types quite far.

It has added support and syntax for:

- type-level strings;
- type-level lists;
- type-level integers;

…

Outlook generic programming: Reflecting types

We can even use GADTs to *reflect* types themselves as data:

```
data Type :: * -> * where
  INT  :: Type Int
  BOOL :: Type Bool
  LIST :: Type a -> Type [a]
  PAIR :: Type a -> Type b -> Type (a,b)
```

Safe dynamically typed values

We can define dynamically typed values by packing up a type representation with a value:

```
data Dynamic :: * where
  Dyn  ::  Type a -> a -> Dynamic
```

Safe dynamically typed values

We can define dynamically typed values by packing up a type representation with a value:

```
data Dynamic :: * where
  Dyn  ::  Type a -> a -> Dynamic
```

To unwrap these values safely, we check whether the types line up as expected:

```
coerce :: Type a -> Dynamic -> Maybe a
coerce t (Dyn t' x) =
  case eqType t t'
    Just Refl -> Just x
    _         -> Nothnig
```

Generic programming

We can also define new functions *by induction on the type structure*:

```
f :: Type a -> ... a ...
```

In this way, we can define our own versions of functions such as show, read, equality, etc.

Outlook: writing webserver with Servant

Servant is a library for describing web APIs.

From such a description, it will generate documentation, a simple webserver, etc.

Instead of describing the APIs using Haskell values – you describe the API as a (complex) Haskell type.

```
type HackageAPI =
       "users" :> Get '[JSON] [UserSummary]
  :<|> "user" :> Capture "username" Username
              :> Get '[JSON] UserDetailed
  :<|> "packages" :> Get '[JSON] [Package]
```

And then *generate* any desired functionality from this description.

Recap: GADTs

GADTs give you more power to define interesting types in Haskell.

We can decorate our types with *more specific information*.

But we still cannot do any interesting *computation* using types.

Recap: GADTs

- GADTs can be used to encode advanced properties of types in the type language.
- We end up mirroring expression-level concepts on the type level (e.g. natural numbers).
- GADTs can also represent data that is computationally irrelevant and just guides the type checker (equality proofs, evidence for addition).
  Such information could ideally be erased, but in Haskell, we can always cheat via `undefined :: Equal Int Bool`…

Recap: type families

- Haskell has various different ways to program with types;
- We'll see numerous applications of these ideas next week, such as data type generic programming.
- But the 'value language' and 'type language' live in very different worlds…