Advanced Functional Programming

Type Families and Data Kinds

Trevor L. McDonell

Utrecht University

Yesterday...

In the previous lecture, we saw some examples of programming with GADTs

- This allowed us to define rich types, enforcing all kinds of properties
- But we also ran into some limitations

Today

- A bit more on the `Equal` data type
- Kinds beyond *
- Programming with types

Equality type

We introduce `Equal` as a proof that two types are equal:

```
data Equal a b where
  Refl :: Equal a a
```

We could even ‘prove’ some properties of the relation:

```
refl  :: Equal a a
sym   :: Equal a b → Equal b a
trans :: Equal a b → Equal b c → Equal a c
```

Expressive power of equality

Equality proofs and phantom types are enough to ‘implement’ GADTs:

```
data Expr a
  = LitI   (Equal a Int)  Int
  | LitB   (Equal a Bool) Bool
  | IsZero (Equal a Bool) (Expr 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!

Towards generic programming

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

```
data Type a 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:

```
cast :: Type a → Dynamic → Maybe a
cast t (Dyn t' x) =
  case eqType t t' of
    Just Refl → Just x
    _         → Nothing
```

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.

System FC

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

```
toVec :: [a] → Vec a ???
```

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

```
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`.

Passing explicit Sums

In the last lecture, we saw how to pass explicit evidence explaining how to add two ‘type-level’ natural numbers:

```
data Sum m n s where
  SumZero ::                Sum Zero     n n
  SumSucc :: Sum m n s → Sum (Succ m) n (Succ s)
```

But constructing this evidence by hand is tedious...

Multi-parameter type classes

We can automate this through a *multi-parameter type class*

```
class Summable m n s | m n → s where
  makeSum :: Sum m n s

instance Summable Zero n n where
  makeSum = SumZero

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

append :: Summable m n s ⇒ Vec a m → Vec a n → 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 equality;
- `MonadState` (from `mtl`) 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. *Fun with type families*.

Associated types and type families

Type classes let you capture an *interface*, such as monads (supporting bind and return), or monoids (supporting an associative binary operator and identity element).

These interfaces 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
  ...

instance Collects IntSet where
  type Elem IntSet = Int
  ...
```

Addition through association

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

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

instance Summable Zero n where
  type TheSum Zero n = n

instance Summable m n ⇒ Summable (Succ m) n where
  type TheSum (Succ m) n = Succ (TheSum m n)

append :: Summable m n
       ⇒ Vec a m → Vec a n → Vec a (TheSum m n)
```

Associated types or multiparameter type?

Both approaches are similar in expressive power.

Multiparameter type classes with functional dependencies are no longer fashionable

- They can make type class resolution unpredictable

Associated types have gained traction in other languages

- Example: Rust, 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 m n
type instance Sum Zero     n = n
type instance Sum (Succ m) n = Succ (Sum m n)
```

This looks much more like regular programming…

Closed 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
```

Closed 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
```

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

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

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

Apartness for closed type families

GHC only moves to the next branch if it know that the previous one may never match.

- The types are *apart* from the pattern

```
> :kind! Sum n (Succ m)
-- does not reduce further!
```

The need for more kinds

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

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

We want “type-level types”, as we have in the term-level.

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` $\rightarrow$ `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.

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` $\rightarrow$ `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.

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`

Example: booleans

```haskell
-- 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 where
  Not 'True  = 'False
  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

List membership

`Member x xs` should be true if `x` is a member of the type-level list `xs`.

```
class Member x xs
instance Member x (x ': xs)
instance Member x xs ⇒ Member x (y ': xs)
instance TypeError ('Text "Not a member: " ':<>: 'ShowType x)
      ⇒ Member x '[]
```

Question

Is this correct?

List membership

`Member x xs` should be true if `x` is a member of the type-level list `xs`.

```
class Member x xs
instance Member x (x ': xs)
instance Member x xs ⇒ Member x (y ': xs)
instance TypeError ('Text "Not a member: " ':<>: 'ShowType x)
      ⇒ Member x '[]
```

Question

Is this correct?

```
instance {-# OVERLAPS #-}
         Member x (x ': xs)
instance {-# OVERLAPPABLE #-}
         Member x xs ⇒ Member x (y ': xs)
```

The case against overlapping instances

1. Overlapping instances make type resolution brittle
    - Which instance is selected depends on how much we know about a type
2. Overlapping instances are not modular
    - Adding a new instance which overlaps may render previous resolution incorrect

List membership with a closed type family

```
type family Member' x xs where
  Member' x '[]       = 'False
  Member' x (x ': xs) = 'True
  Member' x (y ': xs) = Member' x xs

type Member x xs = Member' x xs ~ 'True
```

- The second branch is *non-linear*, `x` is repeated
  - Not allowed in term-level pattern matching
  - Fine with closed type families
- `Member` defines a synonym for a constraint
  - In GHC, constraints are just types of the special `Constraint` kind

Polymorphic kinds

Which is the kind of `Member'`?

```
Member' :: *    → [*]   → Bool
Member' :: Nat → [Nat] → Bool
```

Polymorphic kinds

Which is the kind of `Member'`?

```
Member' :: *    → [*]   → Bool
Member' :: Nat → [Nat] → Bool
```

Kinds may be *polymorphic*, as types are!

```
Member' :: k → [k] → Bool
```

Type-level application

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

Question

What is the kind of `Apply`?

```
> :set -XPolyKinds
> :info Apply
type role Apply representational nominal
data Apply (f :: k → *) (a :: k) = MkApply (f a)
```

Proxy data types

Consider the `read` function:

```
read :: Read a ⇒ String → a
```

How do we fix the type we want to get?

Proxy data types

Consider the `read` function:

```
read :: Read a ⇒ String → a
```

How do we fix the type we want to get?

First solution: annotate it

```
read "123" :: Int
```

In many cases, the type can be inferred instead.

Proxy data types

Second solution: fix it via another parameter

```
-- Wrap it with an additional argument
read' :: Read a ⇒ a → String → a
read' _ = read
-- And use it like this
read' (undefined :: Int) "123"
```

- The first argument is never touched
- Gives valuable information to the compiler

These are called *proxy* arguments.

Proxy data types

```haskell
-- Polymorphic kind!
data Proxy (a :: k) = Proxy
-- The proxy is also ignored
read' :: Read a ⇒ Proxy a → String → a
read' _ = read
-- To use it you create a Proxy value
read (Proxy :: Proxy Int) "123"
```

Question

What is the benefit of using `Proxy`?

(No) proxy data types

Third solution: explicit type application

```
read @Int "123"
```

The `TypeApplications` extension has been available since GHC-8.

Outlook: writing a webserver with Servant

Servant is a library for describing web APIs. From such a description, it will generate documentation, a simple web server, a JavaScript client, etc.

- Instead of describing the APIs using Haskell values—you describe the API as a (complex) Haskell type
- And then *generate* any desired functionality from it

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

Summary

GADTs give you more power to define interesting types

- We can decorate our types with *more specific information*
- We can represent data that is computationally irrelevant, but guides the type checker

But we still cannot do any interesting *computation*

- We need to use type classes or type families
- We end up mirroring expression-level concepts on the type level (e.g. natural numbers)

The ‘value language’ and ‘type language’ live in very different worlds…