COMP3141

Pre-exam Consultation

James Davidson

University of New South Wales

Schedule

I do have a rather hard cutoff time of 13:00 today, so I'll try my best to go through the following:

1. **Demo:** Custom instances (`Functor`, `Applicative`, `Monad`)
2. **Demo:** Checking lawfulness via QuickCheck
3. **Demo:** Structural induction
4. More on GADTs
5. Counting members of types
6. More on effects
7. "Implementation(s) which satisfy .."

More on GADTs

The prototypical example of GADTs in Haskell is a representation for sized vectors, wherein we encode their size at the type level:

```
{-# LANGUAGE KindSignatures, DataKinds, GADTs #-}

data Nat = Zero | Succ Nat
data Vec :: * -> Nat -> * where
    Nil  :: Vec a Zero
    Cons :: a -> Vec a n -> Vec a (Succ n)
```

The `DataKinds` pragma lets us write our type `Nat` in place of a `*`.

Why is this more useful than `Vec :: * -> * -> *` here?

More on GADTs

The prototypical example of GADTs in Haskell is a representation for sized vectors, wherein we encode their size at the type level:

```
{-# LANGUAGE KindSignatures, DataKinds, GADTs #-}

data Nat = Zero | Succ Nat
data Vec :: * -> Nat -> * where
    Nil  :: Vec a Zero
    Cons :: a -> Vec a n -> Vec a (Succ n)
```

The `DataKinds` pragma lets us write our type `Nat` in place of a `*`.
Why is this more useful than `Vec :: * -> * -> *` here?

Quick demo: Inspecting `Nat`'s kinds in a REPL

Counting members of types

```
{-# LANGUAGE KindSignatures, DataKinds, GADTs #-}

data Size = Zero | One
data MyType = Foo | Bar | Baz
data P :: * -> Size -> * where
  PF :: P a Zero
  PT :: P a Zero
  PP :: a -> P a Zero -> P a One
```

Sample exam question

How many elements does the type `P MyType One` have?

- Perhaps a good question to ask here is what does a value of this type even look like?

- Perhaps a good question to ask here is what does a value of this type even look like? One such value is `PP Foo PF`.
- So the only way to construct a value of this type is through the `PP` constructor. It takes two arguments: a value of type `MyType`, and a value of type `P MyType Zero`.

- Perhaps a good question to ask here is what does a value of this type even look like? One such value is `PP Foo PF`.
- So the only way to construct a value of this type is through the `PP` constructor. It takes two arguments: a value of type `MyType`, and a value of type `P MyType Zero`. We can reduce this problem to counting `(MyType, P MyType Zero)` pairs, since each naturally corresponds to a `P MyType One` value.

- Perhaps a good question to ask here is what does a value of this type even look like? One such value is `PP Foo PF`.
- So the only way to construct a value of this type is through the `PP` constructor. It takes two arguments: a value of type `MyType`, and a value of type `P MyType Zero`. We can reduce this problem to counting `(MyType, P MyType Zero)` pairs, since each naturally corresponds to a `P MyType One` value.
- There are three values of type `MyType`, and two ways of constructing a value of type `P MyType Zero`. Why?

- Perhaps a good question to ask here is what does a value of this type even look like? One such value is `PP Foo PF`.
- So the only way to construct a value of this type is through the `PP` constructor. It takes two arguments: a value of type `MyType`, and a value of type `P MyType Zero`. We can reduce this problem to counting `(MyType, P MyType Zero)` pairs, since each naturally corresponds to a `P MyType One` value.
- There are three values of type `MyType`, and two ways of constructing a value of type `P MyType Zero`. Why?
- So there are **six** `(MyType, P MyType Zero)` pairs, and hence **six** `P MyType One` values.

(Yes, the answer to the sample exam question is 10 and not 6. This question is very slightly different!)

Here are all 6 values:

```
vals :: [P MyType One]
vals = [ PP Foo PF
       , PP Foo PT
       , PP Bar PF
       , PP Bar PT
       , PP Baz PF
       , PP Baz PT
       ]
```

Effects and monads

Side effects $\cong$ external effects: phenomena you can observe from outside the scope of the function. Letting these run wild in your program makes it really hard to do reason about them (goodbye, equational reasoning!), but they are necessary to be useful!

A quote from Simon Peyton Jones

"In the end, any program must manipulate state. A program that has no side effects whatsoever is a kind of black box. All you can tell is that the box gets hotter."

Effects and monads

Monads are just a control mechanism for dealing with effects (e.g. the `IO` monad for dealing with i/o effects, the most common type of effect). This does not eliminate the effects, it only subjugates them.

Identifying effects

Identifying effectful functions in Haskell is easy: everything is immutable, so chances are unless you are dealing with i/o operations explicitly, your code *probably* has no side effects.

Identifying effects

Identifying effectful functions in Haskell is easy: everything is immutable, so chances are unless you are dealing with i/o operations explicitly, your code *probably* has no side effects.

Some other basic checks to see if a function is effectful or not in other languages (e.g. C):

- Does it do i/o?

Identifying effects

Identifying effectful functions in Haskell is easy: everything is immutable, so chances are unless you are dealing with i/o operations explicitly, your code *probably* has no side effects.

Some other basic checks to see if a function is effectful or not in other languages (e.g. C):

- Does it do i/o?
- Can it throw exceptions/error messages on some inputs?

Identifying effects

Identifying effectful functions in Haskell is easy: everything is immutable, so chances are unless you are dealing with i/o operations explicitly, your code *probably* has no side effects.

Some other basic checks to see if a function is effectful or not in other languages (e.g. C):

- Does it do i/o?
- Can it throw exceptions/error messages on some inputs?
- Does it involve direct control over non-local memory? (e.g. pointers, memory allocation, global variables, ...)

Identifying effects

Identifying effectful functions in Haskell is easy: everything is immutable, so chances are unless you are dealing with i/o operations explicitly, your code *probably* has no side effects.

Some other basic checks to see if a function is effectful or not in other languages (e.g. C):

- Does it do i/o?
- Can it throw exceptions/error messages on some inputs?
- Does it involve direct control over non-local memory? (e.g. pointers, memory allocation, global variables, ...)
- Is it *referentially transparent?* In other words, can we substitute the appropriate return value of a function for every call to that function and have everything behave identically?

"Implementation(s) which satisfy .."

Sorry to disappoint, but there really isn't a formulaic way to do these kinds of questions! A lot of it comes down to looking at the properties in question and seeing what they tell you about the function you're working with, but that's the only things you can really say about them in general.

It's worth keeping in mind that these questions are also (intentionally!) aimed at distinguishing those who understand the themes of this course at a surface level from those who understand them more deeply.

Quiz 2 has some good insights, and I really can't do much better than those here today. My advice is to study the solutions we provided for those questions and see if you can figure out how the question was approached in each case.

FIN

Good luck, and thanks for sticking with us this term!