Session 2: Types and classes

COMP2221: Functional programming

Lawrence Mitchell[*]

[*] lawrence.mitchell@durham.ac.uk

Recap

- What are some differences between functional and imperative programming?
- Which programming model more closely mirrors the way computers execute?
- What are interpreters and compilers? (in *very* broad terms)
- What are some advantages of an interpreter?
- Side effects (definition)
- Why can side effects easily introduce bugs?

Types

On a computer: everything is bits. → types tell us how to interpret the bits.

- Mathematics and programming rely on the notion of *types*
- Tell us *how* to interpret a variable
- Provide restrictions on valid *operations*

Example: Java/C

```
int a = 4;
int b = 3;

double c = a/b;
```

```
double a = 4;
double b = 3;

double c = a/b;
```

Integer division and floating point division use different instructions. → types tell us which instruction to use.

Types

- Mathematics and programming rely on the notion of *types*
- Tell us *how* to interpret a variable
- Provide restrictions on valid *operations*

Example: Java/C

```
int a = 4;
int b = 3;

double c = a/b;
```

```
double a = 4;
double b = 3;

double c = a/b;
```

Result depends on input types.

Since computers represent *everything* as sequences of bits, types are also required to define what these bit streams mean.

Types are ...

...required to know what a bit sequence means.

Implementation

Find the correct implementation of, for example, '+'

Correctness

check whether an operation on some data is valid and/or well-defined.

check whether a code fragment is correct (type safety)

Documentation

document the code's semantics (for the reader)

Types in Haskell

Haskell is

Strongly, statically typed.

$\Rightarrow$ every well-formed expression has exactly one type, these types are known at *compile time*

Definition (Type)

A type identifies a *collection* of values

Example

- `Bool` the two logical values `True` and `False`.
- `Bool -> Bool` the set of all functions that take a `Bool` as input and produce a `Bool` as output.
- We will see more standard types soon

Notation and inspection

Attaching types

Haskell's notation for "e is of type T" is spelt

```haskell
e :: T
-- False is of type Bool
False :: Bool
-- not is of type Bool -> Bool
not :: Bool -> Bool
```

What type does X have?

Every valid expression in Haskell must have a valid type.

You can ask GHCi what the type of an expression is with the command `:type expr`

```
Prelude> :type sum
sum :: Num a => [a] -> a
```

Type checking I

- Translators must check for *type correctness*

Definition (Statically typed language)

We check correctness at translation time. (C/Java/Haskell/...)

$\Rightarrow$ invalid types mean "translation error"

```haskell
-- Invalid
foo :: a -> Int
foo f = 1 + f
```

Incorrect programs are spotted "early"

Definition (Dynamically typed language)

We check correctness at run time. (Python "duck typing")

$\Rightarrow$ invalid types only detected if we "use them"

```python
# Fine as long as f supports addition with a number
def foo(f):
    return 1 + f
```

Have non-unique return types.

Type checking II

- How does the translator determine the type of an expression?

Explicit annotation
Programmer annotates all variables with type information (e.g. C/Java)

C++ you can use auto
But only local. →

Type inference
Translator *infers* the types of variables based on the operations used (e.g. Haskell/ML)

Hindley-Milner-style type inference.

Duck typing
Translator/runtime just tries the operation, if it succeeds, that was a valid type! (Python)

→ link to some commentary on this on the website.

Demo time

Let's look at some types

Building block summary

- Prerequisites: none
- Content
  - Different concepts of typing (dynamic/static)
  - Looked at some builtin Haskell types
  - Looked at list and tuple types
- Expected learning outcomes
  - student *knows* names of basic Haskell types compiling a programming language
  - student can *explain* difference between lists and tuples in Haskell.
  - student can *use* the Haskell interpreter to determine the type of an expression.
- Self-study
  - None

Functions have types

Programming with functions

- Functions have types in all programming languages, Haskell makes this particularly explicit

Functions of one argument “unary”

Map from one type to another

```haskell
not :: Bool -> Bool
and :: [Bool] -> Bool
```

Functions of two arguments “binary”

Map from two types to another

```haskell
add :: (Int, Int) -> Int
```

“add eats two Ints and returns an Int”

An alternative view

- Since functions are *first class objects*, functions of *more than one* argument are typically written in Haskell as *functionals*
- Naturally extends from binary to n-ary functions

“Curried” view of binary functions

```haskell
add :: Int -> (Int -> Int)
```

“add eats an Int and returns a function which eats an Int and returns an Int”

- This idea comes from the formalism of Lambda calculus

Currying

Definition (Currying (informal))

Turn a function of n arguments into a function of $n - 1$ arguments.

History

- Idea first introduced by Gottlob Frege
- Developed by Moses Schönfinkel in the context of combinatory logic
- Further extended by Haskell Brooks Curry working in logic and category theory
- Name "currying" coined by Christopher Strachey (1967)

Why currying?

- *easier* to reason about and prove things with functions of only 1 variable!
- Flexibility in programming: makes composing functions simpler
- Related to *partial evaluation* where we bind some variables in an n-ary function to a value

Demo time

Let’s look at some functions

Building block summary

- Prerequisites: none
- Content
  - Specifying input and output types of functions
  - Functions have types, and so returning functions is natural
  - Functions of multiple variables can be defined using tuples, or else returning functions on a reduced parameter list
  - Introduction to currying
- Expected learning outcomes
  - student *knows* how to specify the type of a function
  - student *knows* two ways of writing functions of multiple arguments.
  - student can *explain* the difference between these paradigms (currying)
  - student can *illustrate* where currying or not makes a difference in semantics of function application
- Self-study
  - Lecture code.