Topic 9: Type Checking

Recommended Exercises and Readings

- From Haskell: The craft of functional programming (3rd Ed.)
  - Exercises:
    - 13.17, 13.18, 13.19, 13.20, 13.21, 13.22
  - Readings:
    - Chapter 13.5, 13.6 and 13.7

Type systems

- There are (at least) three different ways that we can compare type systems
  - Static vs. dynamic typing
  - Strong vs. weak typing
  - Explicit vs. implicit typing

Static vs. Dynamic Typing

- Static typing
  - The type of a variable is known at compile time
    - It is explicitly stated by the programmer…
    - Or it can be inferred from context
  - Type checking can be performed without running the program
  - Used by Haskell
  - Also used by C, C++, C#, Java, Fortran, Pascal, …

Static vs. Dynamic Typing

- Dynamic typing
  - The type of a variable is not known until the program is running
    - Typically determined from the value that is actually stored in it
  - Program can crash at runtime due to incorrect types
  - Permits downcasting, dynamic dispatch, reflection, …
  - Used by Python, Javascript, PHP, Lisp, Ruby, …

Static vs. Dynamic Typing

- Static vs. Dynamic typing is a question of timing…
  - When is type checking performed?
    - At compile time (before the program runs)? Static
    - As the program is running? Dynamic

Static vs. Dynamic Typing

- Some languages do a mixture of both of static and dynamic typing
  - Most type checking in Java is performed at compile time…
  - … but downcasting is permitted, and can't be verified at compile time
    - Generated code includes a runtime type check
    - A ClassCastException is thrown if the downcast wasn't valid
  - Most type checking in C++ is performed at compile time..
  - … but downcasting is permitted, and can't be verified at compile time
    - A C-style cast on incompatible pointer types results in a C++ reinterpret_cast and often causes unpredictable behaviour for incompatible types
    - Using dynamic_cast in C++ returns NULL for an invalid cast

Static vs. Dynamic Typing

- Advantages of static typing

- Advantages of dynamic typing

Strong vs. Weak Typing

- In a strongly typed language…
  - One must provide a value that matches the expected type
    - Exact match
    - Subtype match
    - Type class match
    - "Duck" match
- In a weakly typed language…
  - One has much more freedom to mix values of different types
- Type strength is a continuum
  - Independent of when the type checking is performed

Example: Adding a Character to an Integer

- Haskell:
  - Prohibited
- Python:
  - Prohibited
    - But Python doesn't really have a character type – it's just a string that has length 1
- Java:
  - Permitted if the result is stored in an int
  - Prohibited if the result is stored in a char
    - Unless it is explicitly cast to a char
- C++:
  - Permitted
    - No warning, even with -Wall
- C:
  - Permitted
    - No warning, even with -Wall
- Perl:
  - Permitted
    - And Perl doesn't really have a character type – it's just a string that has length 1
    - Generates a warning with -w

Example: Adding a String to a String

- Haskell
  - Prohibited with +
  - Permitted with ++
- Python:
  - Permitted
- Java:
  - Permitted
- C++:
  - Permitted
- C:
  - Prohibited with +
    - Two pointers can't be added together
  - Permitted with strcat or strncat
    - Programmer must ensure that there is sufficient memory available
- Perl:
  - Permitted with +
    - But it results in 0
    - Generates a warning with -w
  - Permitted with .
    - Which concatenates the strings

Example: Adding an Integer and a Float

- Haskell:
  - Prohibited
- Python
  - Permitted
    - Result is a float
- Java
  - Permitted
    - If the result is stored in a float
  - Prohibited
    - If the result is stored in an int
      - Unless the result is explicitly cast to an int
- C++:
  - Permitted
    - Result is an int or float, depending on the type of variable it is stored in
- C:
  - Permitted
    - Result is an int or float, depending on the type of variable it is stored in
- Perl:
  - Permitted
    - Result is a float

Explicit vs. Implicit Typing

- Explicit typing
  - Programmer is responsible for specifying types while writing their code
  - Examples: C, Java
- Implicit typing
  - Types are inferred by the type system, or simply aren't needed
  - Examples: Python, Perl

Explicit vs. Implicit Typing

- Haskell allows the programmer to explicitly specify types
  - But explicit types are rarely required
  - Haskell will infer the types (in most situations) when they are not provided
- C++ 11 added the auto keyword
  - Variables still need to be declared…
  - … but the type is automatically inferred by the compiler
    - This is **really** convenient for complicated template types

Type Systems

- Static vs. Dynamic
  - When is type checking done?
- Strong vs. Weak
  - How much type checking is done?
- Explicit vs. Implicit
  - Does the programmer specify the types directly, or does the compiler / interpreter infer them?

Type Checking in Haskell

- An expression can include
  - Literals
  - Operators
  - Function calls
  - Parentheses
  - …
- In Haskell, the type of every expression can be determined at compile time, and checked for type correctness

Type Checking in Haskell

- Consider the expression:

```
chr (ord ch – ord 'a' + ord 'A')
```

- Is this expression type correct?
  - Known from previous type signatures
    - chr :: Int -> Char
    - ord :: Char -> Int

Type Checking in Haskell

- Consider the expression

```
succ (4 :: Int) + False
```

- Is this expression type correct?
  - Known from previous type signatures:
    - succ :: Enum a => a -> a

Type Checking in Haskell

- Type checking guards:

```
f :: t1 -> t2 -> … -> tk -> t
f p1 p2 … pk
  | g1    = e1
  …
  | gj    = ej
```


Polymorphic Type Checking

- Some expressions can't be reduced to a single type
  - The expression's result could be one of several (or many) types
- Monomorphic type checking
  - Each expression can be reduced to a single type, or declared type incorrect
- Polymorphic type checking
  - Each expression is reduced to a set of possible types, or declared type incorrect

Type Checking in Haskell

- Consider the following function, f:

```
f (x,y) = (x, ['a' .. y])
```

- What can be inferred about its type?

Type Checking in Haskell

- Consider the following function, g:

```
g m zs = m + length zs
```

- What can be inferred about its type?

Unification

- Unification identifies the set of types that satisfy two (or more) types
  - What types can (a, [Char]) represent?
  - What types can (Int, [b]) represent?
  - What's the unification of (a, [Char]) and (Int, [b])

Unification

- Unify the types (a, [a]) and ([b], c)
- Unify the types [Int] -> [Int] and a -> [a]

Polymorphic Literals and Functions

- The same literal can appear multiple times in an expression
  - It doesn't necessarily have the same type each time it occurs

```
zip ([] ++ [1]) (['a','b'] ++ [])
```

- Similarly, polymorphic functions can have a different type every time that they occur

Type Classes

- Type classes must also be considered when performing type checking
  - Type classes are attached to type variables as necessary
  - Unification ensures that the types identified are instances of the required classes
  - Type class requirements are simplified where possible
    - A type variable that includes requirements Eq a and Ord a is reduced to Ord a because Ord extends Eq

Example

- Type check the following expression where **`f :: Eq a => a -> b -> Int`**:

```
f (succ . succ) 'c'
```


Example

- Type check the following expression where **`f :: Eq a => a -> b -> Int`**:

```
f 'c' (succ . succ)
```

Some Final Thoughts on Type Checking

- Correct code is great!
- Incorrect code should either
  - Not compile
  - Crash when it is executed
- Incorrect code that runs and doesn't crash is a nightmare!
  - Especially as the project gets larger…