Homework 3: Types

CSE 130: Programming Languages

Early deadline: Feb 7 23:59, Hard deadline: Feb 9 23:59

Names & IDs:

1 [42pts] Type embodiment and parametricity

Parametricity is a property of parametrically polymorphic functions which roughly says that "the more generic a function type is, the more restricted we are in implementing the function." This is because parametric polymorphism *removes information* about the concrete values, leaving only structural properties, later allowing the type to be *instantiated* with different concrete types of values. Intuitively, if we consider the set of inhabitants of some given type, then the more parametric the type, the smaller will be the set of values with such a type. In practice, parametricity allows us to reason about our functions based on the types[1], and is the theory behind some fancy extensions to the typeclass derivation mechanism of Haskell.

In this problem, we will explore some consequences of this property and learn to use types as a reasoning tool, through deriving programs *by calculation*.

1.1 [$2 \times 6 = 12$ pts] The Algebra of Datatypes

Definition: an *inhabitant* of a type τ is a value v such that $v :: \tau$. $|\tau|$ denotes the size of the set of inhabitants of τ.

To get started, we'll look at inhabiting ADTs, and then functions. For this question, do not consider `undefined` (sometimes written $\perp$) as a valid answer.

```
data Animal = Cat | Dog | Mouse
```

1. How many inhabitants does `Animal` have? Give an example.

Answer:

[1] P. Wadler, Theorems for Free! (1989)

```
data AnimalPair = AnimalPair Animal Animal
```

2. How many inhabitants does `AnimalPair` have? Give an example.

Answer:

```
data Maybe a = Just a | Nothing
```

3. How many inhabitants does `Maybe Animal` have? Give an example.

Answer:

4. How many inhabitants does `Maybe a` have? Give your answer in terms of `a`.

Answer:

```
data Pair a b = Pair a b
```

5. How many inhabitants does `Pair a b` have? Give your answer in terms of `a` and `b`.

Answer:

```
data Either a b = Left a | Right b
```

6. How many inhabitants does `Either (Maybe Animal) (Pair (Pair Animal Animal) Animal)` have?

Answer:

1.2 [$2 \times 6 = 12$ pts] Types and Lambda Calculus

Now we consider the effect polymorphism has on the inhabitation of function types. For each of the following types, (a) write down how many distinct functions there are for the given most general type and (b) give an example (if one exists). Write all your functions in λ form.

1. $a \to a$

Answer:

2. $a \to b$

Answer:

3. $a \to b \to a$

Answer:

4. $a \to b \to b$

Answer:

5. $(a \to b \to c) \to (a \to b) \to a \to c$

Answer:

6. $a \to \mathbb{Z}_6$. (Hint: $\mathbb{Z}_n$ is the set of integers modulo n.)

Answer:

1.3 [18pts] Type Tetris

Now we shall put parametricity to work as a reasoning tool to write Haskell programs completely mechanically — by simply plugging together functions of the correct types. For each part, give a definition in Haskell using only the following functions:

$$
\begin{aligned}
(\$) &:: (a \to b) \to a \to b \\
(.) &:: (b \to c) \to (a \to b) \to (a \to c) \\
flip &:: (a \to b \to c) \to (b \to a \to c) \\
map &:: (a \to b) \to [a] \to [b] \\
concat &:: [[a]] \to [a]
\end{aligned}
$$

You may, however, compose the given functions by creating λ abstractions. These are all standard Haskell functions, so you can verify your solutions out in GHCi. (Enter `:t <function>` to print the type of the function). For example, consider the function *hog* with the type:

$$hog :: (c \to d) \to (a \to b \to c) \to a \to b \to d$$

Given the above type, we can derive the following function definition:

$$hog = (.).(.)$$

To solve these problems, you may find it useful to first write them using λ abstractions.

1. [6pts] Give a definition for $bog :: (a \to [b]) \to [a] \to [b]$

Answer:

2. [6pts] Give a definition for $zog :: a \to [a \to b] \to [b]$.

Answer:

3. [6pts] Write *bog* using *hog*.

Answer:

2 [32pts] Type Inference

In this problem you will apply the Hindley-Milner type inference algorithm we discussed in class to figure out the type of two μHaskell declarations. For both, you must go through the five steps: creating the parse trees, assigning type variables, generating constraints, solving the constraints, and finally circling your final type answer (if no type error was encountered).

2.1 [14pts] Infer the type of `reverse`:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Answer:

2.2 [18pts] Infer the type of `foldl`:

```
foldl f acc []     = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
```

Answer:

Acknowledgements

Any acknowledgements, crediting external resources or people should be listed below.