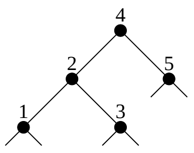
Trees

1. Consider the following data type *Tree* and consider an example inhabitant *tree*:

   data *Tree a* = *Bin* (*Tree a*) *a* (*Tree a*) | *Leaf* **deriving** *Show*

   tree :: *Tree Int*
   tree = *Bin* (*Bin* (*Bin Leaf* 1 *Leaf*) 2 (*Bin Leaf* 3 *Leaf*)) 4 (*Bin Leaf* 5 *Leaf*)



(a) (3 points) Define a higher-order function *foldTree* that corresponds to the fold or catamorphism over our trees using recursion directly. Write down the type of *foldTree*.

Solution:

```
foldTree :: (result → a → result → result, result) → Tree a → result
foldTree alg@(bin, leaf) tree =
  case tree of
    Bin l a r → bin (foldTree alg l) a (foldTree alg r)
    Leaf      → leaf
```

1 point for the type signature; 1 point per case branch.

(b) (4 points) Use *foldTree* to define the following two functions:

```
height  :: Tree a → Int
mapTree :: (a → b) → Tree a → Tree b
```

The function *height* should produce the height of a tree, whereas *mapTree* applies a given function to all the elements of a tree. For example, *height tree* gives 3, and *mapTree even tree* returns *Bin* (*Bin* (*Bin Leaf False Leaf*) *True* (*Bin Leaf False Leaf*)) *True* (*Bin Leaf False Leaf*).

Solution:

```
height    = foldTree (λl _ r → 1 + max l r, 0)
mapTree f = foldTree (λl a r → Bin l (f a) r, Leaf)
```

2 points per function; 1 point per tuple pair.

(c) (3 points) Define a type *TreeF* that corresponds to the *pattern functor* representing the *Tree* data type. You may do so by defining a data type directly, or using the pattern functor combinators we have seen in class. Show that this type is indeed a functor by defining its *Functor* instance.

Solution:

```
type TreeF a t = BinF t a t | LeafF
```

1 point for the right kind; 1 point per constructor.

(d) (4 points) Use *foldTree* to write a function for collecting the elements of a tree in a depth-first order.

```
depthfirst :: Tree a → [a]
```

For instance, *depthfirst tree* gives $[1, 3, 2, 5, 4]$. Take into account that concatenation of two lists (++) requires a traversal over the left operand – for full points you should prevent quadratic behavior for *depthfirst*.

Solution:

```
depthfirst tree =
  foldTree (λf a g → f ∘ g ∘ (a:), id) tree []
```

2 points for naive solution. 1 point for using difference lists; 1 point for base case; 2 points for each component of the algebra.

(e) (2 points) A colleague defines the following *Arbitrary* instance for trees:

```
instance Arbitrary a ⇒ Arbitrary (Tree a) where
  arbitrary = oneof [arbitraryBin, return Leaf]
    where
      arbitraryBin :: Arbitrary a ⇒ Gen (Tree a)
      arbitraryBin = do
        t1 ← arbitrary
        x ← arbitrary
        t2 ← arbitrary
        return (Bin t1 x t2)
```

What problems might you encounter when running tests on random trees generated in this fashion? How can these be fixed?

Solution: It may generate diverge when generating test data. These can be fixed by using combinators such as *sized*.

(f) (6 points) Give the definition and type signature of a function *unfoldTree* corresponding to the unfold or anamorphism for our tree data type. Use *unfoldTree* to write a function *buildTree* (partial) inverse of *depthfirst* using it. Formulate the QuickCheck property relating *depthfirst* and *buildTree* that you expect to hold.

Solution:

```
unfoldTree :: (s → Maybe (s, a, s)) → s → Tree a
unfoldTree coalg seed = case coalg seed of
  Nothing → Leaf
  Just (l, x, r) → Bin (unfoldTree coalg l) x (unfoldTree coalg r)
buildTree :: [a] → Tree a
buildTree = unfoldTree step
  where
  step [] = Nothing
  step (x : xs) = Just ([], x, xs)
buildTreeProp :: [Int] → Bool
buildTreeProp xs = depthfirst (buildTree xs) ≡ xs
```

1 point for type signature; 1 point per branch; 2 point for buildTree; 1 point for property.

(g) (2 points) Using *unfoldTree* it becomes possible to computations returning *infinite* data structures. Use *unfoldTree* to define an infinitely deep tree *natTree*, where every node is labeled with its depth: the root is labelled 0; its children are labelled 1; the root's grandchildren are labelled 2; etc.

Solution:

```
natTree :: Tree Int
natTree = unfoldTree coalg 0
  where
  coalg n = Just (n + 1, n, n + 1)
```

(h) (4 points) A colleague claims that the definition of *natTree* relies on Haskell's lazy evaluation. Explain how – inspired by work on data fusion – infinite data structures can even be represented in *strict* programming languages.

> **Solution:** We can represent an infinite data type by the coalgebra generating it – this is what data fusion techniques do. The resulting definitions are no longer recursive and therefore no longer rely on lazy evaluation.

Success

2. With the data type *Step* we can encode certain search problems. At every point in the search space, we can return success, failure, or choice of steps that may themselves fail or succeed:

   data *Step a* = *Success a* | *Fail* | *Steps* [*Step a*]

(a) (4 points) Make *Step* an instance of the *Functor* and *Foldable* type classes. These are declared as follows:

```
class Functor f where
  fmap :: (a → b) → f a → f b
class Foldable f where
  foldMap :: Monoid m ⇒ (a → m) → f a → m
```

> **Solution:** instance Functor Step where fmap f Fail = Fail fmap f (Success a) = Success (f a) fmap f (Steps steps) = Steps (map (fmap f) steps)
>
> instance Foldable Step where foldMap f Fail = mempty foldMap f (Success x) = f x foldMap f (Steps steps) = foldr mempty mappend (map (foldMap f) steps)

(b) (3 points) Show how to use the *foldMap* function to define a function *collect*, that computes a list with all the *Success* values in a *Step* data structure.

> **Solution:**
>
> ```
> collect :: Step a → [a]
> collect = foldMap (λx → [x])
> ```

(c) (4 points) Write the monadic *join* function for the *Step* data type:

```
join :: Step (Step a) → Step a
```

> **Solution:** join :: Step (Step a) -¿ Step a join Fail = Fail join (Success a) = a join (Steps steps) = Steps (map join steps)

(d) (4 points) Give an instance definition fo the *Step* data type for the *Monad* type class. Your instance declaration should respect the three monad laws (but you don't have to prove this).

> **Solution:**
>
> ```
> instance Monad Steps where
>   return x = Success x
>   Fail ≫= f = Fail
>   Success x ≫= f = f x
>   (Steps steps) ≫= f = Steps (map (λstep → step ≫= f) steps)
> ```

(e) (4 points) With *Step* being a *Monad*, we can now use Haskell's **do** notation:

$$\begin{array}{l} m :: \mathit{Step}\ (\mathit{Int}, \mathit{Int}) \\ m = \mathbf{do}\ a \leftarrow \mathit{Success}\ 1 \\ \qquad\quad b \leftarrow \mathit{Steps}\ [\mathit{Fail}, \mathit{Success}\ 2] \\ \qquad\quad \mathit{return}\ (a, b) \end{array}$$

Give the value of m in terms of the three constructors of *Step*.

Solution: Steps [Fail,Success (1,2)]

(f) (6 points) The following law should hold for the *Step* monad:

$$\mathit{fmap}\ f \circ \mathit{fmap}\ g \equiv \mathit{fmap}\ (f \circ g)$$

Prove that the instance declaration you provided for **(a)** respects this law. (If the law does not hold, your instance definition is probably wrong). If you rely on any auxiliary lemmas, these should be stated explicitly.

Solution: Do induction over *Steps*:

- The case for *Fail* we have:

$$\begin{array}{l} \mathit{fmap}\ f\ (\mathit{fmap}\ g\ \mathit{Fail}) \\ \equiv \\ \mathit{fmap}\ f\ \mathit{Fail} \\ \equiv \\ \mathit{Fail} \\ \equiv \\ \mathit{fmap}\ (f \circ g)\ \mathit{Fail} \end{array}$$

- The case for *Success* we have:

$$\begin{array}{l} \mathit{fmap}\ f\ (\mathit{fmap}\ g\ (\mathit{Success}\ x)) \\ \equiv \\ \mathit{fmap}\ f\ (\mathit{Succes}\ (g\ x)) \\ \equiv \\ \mathit{Succes}\ (f\ (g\ x)) \\ \equiv \\ \mathit{fmap}\ (f \circ g)\ (\mathit{Success}\ x) \end{array}$$

- The case for *Steps*:

$$\begin{array}{l} \mathit{fmap}\ f\ (\mathit{fmap}\ g\ (\mathit{Steps}\ \mathit{steps})) \\ \equiv \\ \mathit{fmap}\ f\ (\mathit{Steps}\ (\mathit{map}\ (\mathit{fmap}\ g)\ \mathit{steps}) \\ \equiv \\ \mathit{Steps}\ (\mathit{map}\ (\mathit{fmap}\ f)\ (\mathit{map}\ (\mathit{fmap}\ g)\ \mathit{steps})) \\ \equiv \\ \mathit{Steps}\ (\mathit{map}\ (\mathit{fmap}\ (f \circ g))\ \mathit{steps}) \\ \equiv \\ \mathit{fmap}\ (f \circ g)\ (\mathit{Steps}\ \mathit{steps}) \end{array}$$

Printf

3. In this question, you will show how to use GADTs to define a well-typed *printf* function. The *printf* function is typically used for debugging. The first argument is a string. This string is printed to `stdout`. If the string contains special *formatting directives*, the call to *printf* requires additional arguments. For

the purpose of this exercise, we will assume two formatting directives `%d` and `%s`. Here are some example invocations of *printf*:

```
printf("Hello world\n");
\\ Prints: Hello world\n
printf("Hello there %s", "Wouter");
\\ Prints: Hello there Wouter
printf("%s = %d", "x", 5);
\\ Prints: x = 5
```

Instead of taking a string as its first argument, however, we will use (a variation of) the following data type to represent our formatting directives:

```
data Format where
  Lit :: String → Format → Format   -- Print a literal string fragment and continue
  D :: Format → Format              -- Require an integer input
  S :: Format → Format              -- Require a string input
  Done :: Format                    -- No further directives
```

(a) (3 points) Show how the formatting strings from the three examples above can all be represented as values of type *Format*.

Solution:

- *Lit* `"Hello World"` *Done*
- *Lit* `"Hello there "` (*S Done*)
- *S* (*Lit* `" = "` (*D Done*))

(b) (4 points) The *Format* data type is a simple algebraic data type. Define a GADT *Format′ a* that takes a single type argument. The three examples above should have the following types:

- *Format′ String*
- *Format′* (*String* → *String*)
- *Format′* (*String* → *Int* → *String*)

A formatting directive of type *Format′ a* requires additional arguments specified by the type *a*. Ensure that your answers from part **(a)** give rise to a *Format′* data type of the correct type.

Solution:

```
data Format a where
  Done :: Format String
  Lit :: String → Format a → Format a
  D :: Format a → Format (Int → a)
  S :: Format a → Format (String → a)
```

(c) (6 points) Define a function *format* :: *Format′ a* → *a* that behaves like the *printf* function described above. You may find it useful to define this function using an auxiliary function:

format′ :: *Format′ a* → *String* → *a*

which is defined using an accumulating parameter.

Solution:

$$
\begin{array}{l}
\mathit{format} :: \mathit{Format}\ a \to a \\
\mathit{format}\ f = \mathit{format}'\ f\ \texttt{""} \\
\quad \textbf{where} \\
\qquad \mathit{format}' :: \mathit{Format}\ a \to \mathit{String} \to a \\
\qquad \mathit{format}'\ (\mathit{Done})\ s = s \\
\qquad \mathit{format}'\ (\mathit{Lit}\ \mathit{str}\ f)\ s = \mathit{format}'\ f\ (s +\!\!+\ \mathit{str}) \\
\qquad \mathit{format}'\ (D\ f)\ s = \lambda d \to \mathit{format}'\ f\ (s +\!\!+\ \mathit{show}\ d) \\
\qquad \mathit{format}'\ (S\ f)\ s = \lambda s' \to \mathit{format}'\ f\ (s +\!\!+\ s')
\end{array}
$$

(d) (4 points) To provide exactly the same interface as *printf*, a fellow student proposes to define an auxiliary function:

$$\mathit{toFormat}' :: \mathit{String} \to \mathit{Format}'\ a$$

that parses the *String* argument and recognizes any formatting directives. What will go wrong when trying to define the *toFormat* function? How might you be able to fix this?

Solution: The above type signature is too general. It guarantees to return a *Format′ a for all a*. It needs to be hidden or existentially quantified rather than universally quantified.

SKI – Take home

4. **Please submit your solution no later than midnight on Friday April 14th via submit.**

   Ulf Norell's Agda tutorial defines a data type for the well-typed, well-scoped lambda terms:

   $$\begin{array}{l}\textbf{data}\ \mathit{Term}\ (\Gamma : \mathit{Ctx}) : \mathit{Type} \to \mathit{Set}\ \textbf{where}\\ \quad \mathit{var} : \tau \in \Gamma \to \mathit{Term}\ \Gamma\ \tau\\ \quad \mathit{app} : \mathit{Term}\ \Gamma\ (\sigma \Rightarrow \tau) \to \mathit{Term}\ \Gamma\ \sigma \to \mathit{Term}\ \Gamma\ \tau\\ \quad \mathit{lam} : \mathit{Term}\ (\sigma :: \Gamma)\ \tau \to \mathit{Term}\ \Gamma\ \tau\end{array}$$

   In the lectures, we saw how to define a simple evaluator for such lambda terms.

   $$\begin{array}{l}\mathit{Val} : \mathit{Type} \to \mathit{Set}\\ \mathit{Val}\ i \qquad\quad = \mathit{Unit}\\ \mathit{Val}\ (\sigma \Rightarrow \tau) = \mathit{Val}\ \sigma \to \mathit{Val}\ \tau\\ \mathit{eval} : \mathit{Term}\ \Gamma\ \sigma \to \mathit{Env}\ \Gamma \to \mathit{Val}\ \sigma\end{array}$$

   In this exercise you will define a translation from these lambda terms to SKI combinators and try to prove it correct. You may find it useful to consult background material on combinatory logic to help understand the translation from lambda terms to SKI combinators.

   To define this translation, you will need to define several intermediate steps:

   (a) (4 points) Define a data type for well-typed combinator terms. This data type should be able to represent:

   - the atomic combinators S, K, and I;
   - the application of one combinator term to another;
   - variables drawn from some context.

   Crucially, this term language should *not* include a constructor for lambdas. Be careful to choose your datatype so that it can *only* represent *well-typed* SKI terms. You may want to use `ghci` to check the types of the three atomic combinators for you.

   (b) (4 points) Define an interpretation function, *evalSKI*, that given an *SKI* term of type σ and a suitable environment, produces a value of type *Val* σ.

   (c) (6 points) Define a translation from lambda terms, *Term* Γ σ, to your *SKI* data type. *Hint:* define an auxiliary function *lambda* to handle the case for abstractions *lam*. What is the type of *lambda*?

   (d) (6 points) Formulate the property that your translation is correct and prove that this is the case for applications and variables. What goes wrong when you try to prove the branch for lambda abstractions? What property can you formulate and prove that relates *evalSKI* and the auxiliary *lambda* that you defined above?