INFOAFP – Exam

Andres Löh

Wednesday, 16 April 2008, 09:00–12:00

Solutions

- Not all possible solutions are given.
- In many places, much less detail than I have provided in the example solution was actually required.
- Solutions may contain typos.

Evaluation strategies (19 points total)

1 (5 points)**.** Give an example of a Haskell expression of type *Bool* that evaluates to *True* and that would not terminate (i.e., loop forever) in a language with strict evaluation.

Using equational reasoning, give the reduction sequence of your expression to *True* and indicate clearly where a strict reduction strategy would select another redex. •

Solution 1. Here is one option:

$$example_1 = head\ (repeat\ True)$$

And the reduction:

$$\begin{aligned}
& example_1 \\
\equiv\ & \quad \{ \text{ definition of } example_1 \ \} \\
& head\ (repeat\ True) \\
\equiv\ & \quad \{ \text{ definition of } repeat \ \} \\
& head\ (True : repeat\ True) \\
\equiv\ & \quad \{ \text{ definition of } head \text{ (strict evaluation would reduce } repeat\ True \text{ again) } \} \\
& True
\end{aligned}$$

Another option:

$$\begin{aligned}
& loop = loop \\
& example_2 = const\ True\ loop
\end{aligned}$$

And the reduction:

$$\begin{aligned}
& example_2 \\
\equiv\ & \quad \{ \text{ definition of } example_2 \ \} \\
& const\ True\ loop \\
\equiv\ & \quad \{ \text{ definition of } const \text{ (strict evaluation would reduce } loop\text{) } \} \\
& True
\end{aligned}$$

○

2 (4 points)**.** Haskell has non-strict semantics (i.e., the implementations use lazy evaluation) and is called a pure language. (S)ML on the other hand has strict semantics and is an impure language.

What does purity mean in this context? Give an (small!) example of why Haskell is considered pure and (S)ML is not. [Syntactic correctness, particularly of (S)ML code, is not important.] •

Solution 2. Purity refers to the fact that functions are pure. A pure function depends only on its input, i.e., will always have the same result when applied to a specific input. In SML, one can write

val x = **ref** 0

and then

$$\textbf{fun}\, f\ () = (x := !x + 1; !x)$$

The function f has type $unit \to int$, i.e., there is only one value $()$ we can apply f to. Consequently, if pure, f would have to be a constant function – but f will produce different results because it depends on the current value of the reference x as a side effect.

In Haskell, all functions (except for corner cases such as *unsafePerformIO*) are pure. Side effects are only possible by producing values of certain built-in types such as *IO* that are subsequently interpreted by the run-time system. ○

3 (4 points)**.** Would a lazy impure language or a strict pure language be possible? Would such programming languages be useful? Discuss briefly. ●

Solution 3. Both variants are possible. While some impure languages are known to allow laziness in selected areas, a generally lazy impure language would probably not be very useful, because effects and laziness do not mix well. It becomes extremely hard to track when side effects will occur.

A strict pure language can be useful. Strict languages do not have $\bot$ as a value of every datatype, and that makes reasoning about programs simpler. Most strict languages are impure just because strict evaluation is easier to track for the programmer, and impure handling of side effects may seem more convenient. There is, however, no good reason not to be more disciplined in a strict setting. Ωmega is an example of a strict pure language. ○

4 (6 points)**.** Consider the following Haskell functions. Divide the functions into equivalence classes, i.e., group the functions that are semantically equivalent (efficiency is irrelevant). Give as many examples as needed to demonstrate that each of the classes has indeed different behaviour.

$s1 :: [a] \to b \to b$
$s1\ xs\ y = \textbf{case}\ xs\ \textbf{of}\ \{[\,] \to y; \_ \to y\}$

$s2 :: [a] \to b \to b$
$s2\ xs\ y = seq\ xs\ y$

$s3 :: [a] \to b \to b$
$s3\ xs\ y = y$

$s4 :: [a] \to b \to b$
$s4\ xs\ y = \textbf{if}\ null\ xs\ \textbf{then}\ y\ \textbf{else}\ y$

$s5 :: [a] \to b \to b$
$s5\ xs\ y = \textbf{if}\ map\ (const\ 0)\ xs == [\,]\ \textbf{then}\ y\ \textbf{else}\ y$

$s6 :: [a] \to b \to b$

$s6\ xs\ y = \textbf{case}\ xs\ \textbf{of}\ \{[\,] \to y; [x] \to y; \_ \to y\}$

$s7 :: [a] \to b \to b$
$s7\ xs\ y = seq\ [xs]\ y$

•

Solution 4. There are three equivalence classes that can be distinguished by looking at the cases $xs = \bot$ and $xs = 0 : \bot$.

Functions *s3* and *s7* return *y* in all cases.

Functions *s1*, *s2*, *s4* and *s5* evaluate the first argument to weak head normal form, and therefore return $\bot$ if $xs = \bot$, but return y for $xs = 0 : \bot$ because that is already in weak head normal form.

Function *s6* has a pattern $[x]$ and therefore has to evaluate the tail of *xs* to weak head normal form as well. It will therefore return $\bot$ for both $xs = \bot$ and $xs = 0 : \bot$. ○

Interactive programs (12 points total)

Consider the following datatype:

$$\begin{aligned}\textbf{data}\ GP\ a &= End\ a\\ &\mid Get\ (Int \to GP\ a)\\ &\mid Put\ Int\ (GP\ a)\end{aligned}$$

A value of type *GP* can be used to describe programs that read and write integer values and return a final result of type *a*. Such a program can end immediately (*End*). If it reads an integer, the rest of the program is described as a function depending on this integer (*Get*). If the program writes an integer (*Put*), the value of that integer and the rest of the program are recorded.

The following expression describes a program that continuously reads integers and prints them:

$echo = Get\ (\lambda n \to Put\ n\ echo)$

5 (1 point)**.** What is the (inferred) type of *echo*? •

Solution 5.

$echo :: GP\ a$

○

6 (4 points)**.** Write a function

$run :: GP\ a \to IO\ a$

that can run a *GP*-program in the *IO* monad. A *Get* should read an integer from the console, and *Put* should write an integer to the console.

Here is an example run from GHCi:

```
Main⟩ run echo
? 42
42
? 28
28
? 1
1
? −5
−5
? Interrupted.
Main⟩
```

[To better distinguish inputs from outputs, this version of *run* prints a question mark when expecting an input. It is not required that your version does the same.] •

Solution 6. Here is the variant which prints the question mark. For this to run properly in a compiled program, we would also have to flush *stdout* or modify the buffering behaviour of *stdout*.

```
run (End x)   = return x
run (Get g)   = putStr "? " ≫ getLine ≫= run ∘ g ∘ read
run (Put n k) = putStrLn (show n) ≫ run k
```

Simple version without prompt:

```
run′ (End x)   = return x
run′ (Get g)   = getLine ≫= run′ ∘ g ∘ read
run′ (Put n k) = putStrLn (show n) ≫ run′ k
```

It would of course be nice to also handle incorrect inputs more gracefully than *read* does. ○

7 (3 points)**.** Write a *GP*-program *add* that reads two integers, writes the sum of the two integers, and ultimately returns (). •

Solution 7.

```
add :: GP ()
add = Get (λx → Get (λy → Put (x + y) (End ())))
```

○

8 (4 points)**.** Write a *GP*-program *accum* that reads an integer. If the integer is 0, it returns the current total. If the integer is not 0, it adds the integer to the current total, prints the current total, and starts from the beginning. •

Solution 8.

```
accum :: GP Int
accum = accum′ 0

accum′ :: Int → GP Int
accum′ t = Get (λn → if n == 0 then End t
                                else Put (t + n) (accum′ (t + n)))
```

○

Simulation (21 points total)

9 (4 points)**.** Instead of running a *GP*-program in the *IO* monad, we can also simulate the behaviour of such a program by providing a (possibly infinite) list of input values. Write a function

```
simulate :: GP a → [Int] → (a, [Int])
```

that takes such a list of input values and returns the final result plus the (possibly infinite) list of all the output values generated. ●

Solution 9. This is the straight-forward solution:

```
simulate (End x)   inp     = (x, [])
simulate (Get g)   (i : inp) = simulate (g i) inp
simulate (Put n k) inp     = let (r, out) = simulate k inp
                             in  (r, n : out)
```

Using an accumulating argument to hold the list of outputs is not helpful, because then it is not possible to analyze partial results – in particular, *echo* can never be simulated with an accumulating version because it never ends. ○

10 (3 points)**.** What is the result of evaluating the following two expressions?

```
simulate accum [5, 4 . . 0]

simulate accum [5, 4 . . 1]
```

●

Solution 10. The first expression results in

$$(15, [5, 9, 12, 14, 15])$$

whereas the second expression results in a pattern match failure. Note that the exact nature of the result of the second expression depends on how *simulate* deals with an inputs list that is not sufficiently long. ○

11 (4 points)**.** Define a QuickCheck property that states the following property using *simulate*:

"If *echo* is given n numbers as input, then the first n numbers of its output will be identical to the input." •

Solution 11.

$$\begin{aligned}&\mathit{echoP} :: [\mathit{Int}] \rightarrow \mathit{Bool}\\ &\mathit{echoP}\ \mathit{xs} = \mathit{take}\ (\mathit{length}\ \mathit{xs})\ (\mathit{snd}\ (\mathit{simulate}\ \mathit{echo}\ \mathit{xs})) == \mathit{xs}\end{aligned}$$

Or even simpler:

$$\begin{aligned}&\mathit{echoP}' :: [\mathit{Int}] \rightarrow \mathit{Bool}\\ &\mathit{echoP}'\ \mathit{xs} = \mathit{xs}\ \text{`}\mathit{isPrefixOf}\text{`}\ \mathit{snd}\ (\mathit{simulate}\ \mathit{echo}\ \mathit{xs})\end{aligned}$$

using *isPrefixOf* from *Data.List*. Indeed, these properties are fulfilled – evaluating

$$\mathit{tests} = \mathit{quickCheck}\ \mathit{echoP} \gg \mathit{quickCheck}\ \mathit{echoP}'$$

results in

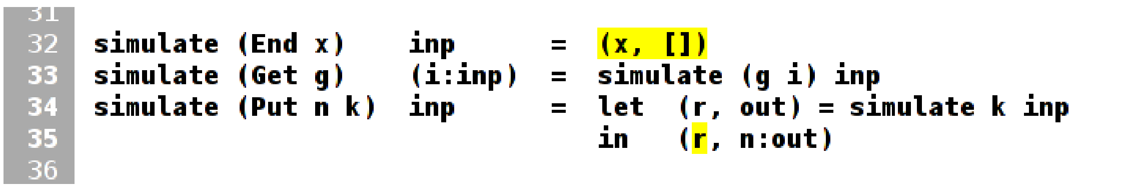
```
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Using an accumulating version of *simulate*, the properties will fail immediately. ○

12 (4 points)**.** Which parts of the definition of *simulate* are covered by your property, and which are not? (I.e., which parts of the definition of *simulate* would be highlighted by HPC after running QuickCheck on your property – assuming that QuickCheck generates suitably random lists.) •

Solution 12.

Here is the actual output of HPC:



```
simulate (End x)    inp      =  (x, [])
simulate (Get g)    (i:inp)  =  simulate (g i) inp
simulate (Put n k)  inp      =  let  (r, out) = simulate k inp
                                in   (r, n:out)
```

The rhs of the case for *End* is not covered, because *echo* does not ever produce *End*. Also, the first component of the resulting pair is not covered, because the property only checks the list, i.e., the second component.

Using an accumulating version of *simulate*, the *End* case will also not be covered, plus the accumulator itself will not be covered in all the cases – since the property always fails with an exception, there will never be a chance to actually perform the comparison on the resulting list. ○

13 (6 points)**.** This is an attempt to define a QuickCheck property for *accum*:

```
accumP :: [Int] → Property
accumP xs = all (λx → x > 0) xs ⟹
                simulate accum (xs ++ [0]) == (last sl, sl)
  where sl = scanl1 (+) xs
```

Here, *scanl1* is defined as follows

```
scanl1 :: (a → a → a) → [a] → [a]
scanl1 f [ ]      = [ ]
scanl1 f (x : xs) = scanl f x xs

scanl :: (a → b → a) → a → [b] → [a]
scanl f x xs = x : case xs of
                 [ ]    → [ ]
                 y : ys → scanl f (f x y) ys
```

There are at least two problems with this property. Describe how they can be fixed [a description is sufficient]. •

Solution 13. The definition of *accumP* fails with an exception on an empty input list due to the call to *last*. There are several ways to fix this. For instance, one can use a variant of *last* that returns 0 on the empty list.

Also, the condition that all values of the list must be positive will quickly exhaust the test values. One can use a custom generator to fix this problem.

Here is a new version of the property that works:

```
accumP′ :: Property
accumP′ = forAll pos $ λxs →
            let sl = scanl1 (+) xs
            in  simulate accum (xs ++ [0]) == (last′ sl, sl)

last′ xs = if null xs then 0 else last xs

pos :: Gen [Int]
pos = sized $ λn → do
                 k ← choose (0, n)
                 replicateM k (choose (1, n + 1))
```

Calling *quickCheck accumP′* yields:

```
+++ OK, passed 100 tests.
```

○

Functors and monads (24 points total)

A map function for *GP* can be defined as follows:

```
instance Functor GP where
  fmap f (End x)   = End (f x)
  fmap f (Get g)   = Get (fmap f ∘ g)
  fmap f (Put n x) = Put n (fmap f x)
```

14 (2 points)**.** Describe the difference between the behaviour of *run accum* and the behaviour of *run* (*fmap* (∗2) *accum*). ●

Solution 14. Only the final result will be doubled. The values printed during the execution of *accum* are unchanged. ○

Instances of class *Functor* should generally fulfill the following two laws:

$$\begin{aligned} &\forall x. && \mathit{fmap}\ \mathit{id}\ x \equiv x \\ &\forall f\ g\ x. && \mathit{fmap}\ (f \circ g)\ x \equiv \mathit{fmap}\ f\ (\mathit{fmap}\ g\ x) \end{aligned}$$

15 (8 points)**.** Prove the **first** of the two laws using equational reasoning (and ignoring that values can be $\bot$).

Note that if you want to prove a property $P\ p$ for any $p :: GP\ a$ via structural induction, you have to prove the following three cases:

$$\begin{aligned} &\forall x. && P\ (End\ x) \\ &\forall g. && (\forall x. P\ (g\ x)) \Rightarrow P\ (Get\ g) \\ &\forall n\ p. && P\ p \Rightarrow P\ (Put\ n\ p) \end{aligned}$$

(Here, $\Rightarrow$ denotes logical implication.) Note that the second case is slightly unusual due to the function argument of *Get*: you may assume that $P\ (g\ x)$ holds for any value of x! ●

Solution 15. We prove $\forall y. \mathit{fmap}\ \mathit{id}\ y \equiv y$ by structural induction on $y :: GP\ a$.

Case $y = End\ x$:

$$\begin{aligned} & \mathit{fmap}\ \mathit{id}\ (End\ x) \\ \equiv\ & \{\ \text{definition of } \mathit{fmap}\ \} \\ & End\ (\mathit{id}\ x) \\ \equiv\ & \{\ \text{definition of } \mathit{id}\ \} \\ & End\ x \end{aligned}$$

Case $y = Get\ g$:

$$\begin{aligned} & \mathit{fmap}\ \mathit{id}\ (Get\ g) \\ \equiv\ & \{\ \text{definition of } \mathit{fmap}\ \} \\ & Get\ (\mathit{fmap}\ \mathit{id} \circ g) \end{aligned}$$

```
  ≡   { definition of (∘) }
    Get (λx → fmap id (g x))
  ≡   { induction hypothesis }
    Get (λx → g x)
  ≡   { eta-reduction (i.e., rewriting to point-free form) }
    Get g
```

Case *y* = *Put n x*:

```
    fmap id (Put n x)
  ≡   { definition of fmap }
    Put n (fmap id x)
  ≡   { induction hypothesis }
    Put n x
```

○

16 (5 points)**.** Define a sensible monad instance for *GP*. ●

Solution 16.

```
instance Monad GP where
  return x          = End x
  (Get g)   >>= f = Get (λn → g n >>= f)
  (Put n x) >>= f = Put n (x >>= f)
  (End x)   >>= f = f x
```

○

17 (5 points)**.** Define a sensible *MonadState* instance for *GP*. Recall the *MonadState* class:

```
class (Monad m) ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
```

●

Solution 17.

```
instance MonadState Int GP where
  get   = Get (λn → End n)
  put n = Put n (End ())
```

○

18 (4 points)**.** What is the difference between the normal state monad as defined in module *Control.Monad.State* and *GP*? Discuss whether you think it is a good idea to make *GP* an instance of *MonadState*. ●

Solution 18. While *State* maintains a single piece of state that can be affected using *get* and *put*, *GP* makes use of two independent streams for input and output. As a consequence, two subsequent invocations of *get* can produce different results, or a *get* immediately after *put* will normally produce a different value than the one written using *put*. In other words, properties one might expect for instances of *MonadState* such as

$$\begin{array}{lll}\forall f\ x. & \textbf{do}\ \{get; x \leftarrow get; f\ x\} & \equiv \textbf{do}\ \{x \leftarrow get; f\ x\} \\ \forall f\ x\ y. & \textbf{do}\ \{put\ x; y \leftarrow get; f\ y\} & \equiv \textbf{do}\ \{put\ x; f\ x\}\end{array}$$

do not hold for *GP*. On the positive side, it is not given that all instances of *MonadState* should have these properties, and *GP* does implement the *MonadState* interface – so one can argue that it should be reused. ○

Type classes (10 points total, 5 bonus points)

19 (2 points)**.** Consider this program:

```
equal :: (Eq s, MonadState s m) ⇒ m Bool
equal = do
          x ← get
          y ← get
          return (x == y)
```

Is the given type signature the most general type signature for *equal*? What would happen if the type signature would be omitted? •

Solution 19. Yes, the type signature gives the most general type. Without the type signature, one would get a type error due to the monomorphism restriction. If the monomorphism restriction is explicitly disabled in GHC, then GHC is able to infer the type. ○

20 (8 points)**.** Translate type classes into explicit evidence in the above function *equal*. Desugar the **do**-notation in the process [use the "simple" desugaring, without the possibility to pattern match on the left hand side of an arrow]. Define the dictionary types that are required – you may omit class methods that are not relevant to this example. You may also declare local abbreviations using **let**. •

Solution 20.

```
data EqD a           = EqD           {eq :: a → a → Bool}

data MonadD m        = MonadD        {ret :: ∀a.a → m a,
                                      bind :: ∀a b.m a → (a → m b) → m b}

data MonadStateD s m = MonadStateD{monad :: MonadD m,
                                      gt :: m s}
```

```
equalD :: EqD s → MonadStateD s m → m Bool
equalD eqD msD =
  let (≫=) = bind (monad msD)
      return = ret (monad msD)
  in  gt msD ≫= λx →
      gt msD ≫= λy →
      return (eq eqD x y)
```

○

21 (5 *bonus points*)**.** Haskell does not offer a scoping mechanism for instances. Instances are always exported from modules, even if nothing else is. Also, instances cannot be local. For example,

```
let instance Eq Char where
      x == y = ord (toUpper x) == ord (toUpper y)
in "hello" == "HeLlo"
```

(using *ord* and *toUpper* from *Data.Char*) is not legal Haskell.

Why do you think this decision has been made? Are there any problems you can think of? ●

Solution 21. The main problem is that it becomes tricky to track when the local instances apply. The reason is that context reduction does not always take place immediately.

In the example above, one would probably expect that the type of the expression is *Bool* and the local instance is used. However, what if we replace the expression after **in** by

```
λxs ys → xs ++ ys == ys ++ xs
```

The type of this function is $Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool$. At this point, it is not yet clear if the function will be applied to strings or to some other lists.

One can of course specify the behaviour of this and comparable cases, but it is not immediately clear which behaviour would be most intuitive, and all such changes would at least make the language more complicated.

Exactly the same problems apply to instances that are local to a module. ○

GADTs and kinds (14 points total)

Here is a variation of *GP*:

```
data GP′ :: ∗ → ∗ where
  Return :: a → GP′ a
  Bind   :: GP′ a → (a → GP′ b) → GP′ b
  Get′   :: GP′ Int
  Put′   :: Int → GP′ ()
```

This is a GADT. The type *GP′* can trivially be made an instance of the classes *Monad* and *MonadState*:

```
instance Monad GP′ where
  return = Return
  (≫=) = Bind

instance MonadState Int GP′ where
  get = Get′
  put = Put′
```

22 (6 points)**.** A value of type *GP* can easily be transformed into a value of type *GP′* as follows:

```
gp2gp′ :: GP a → GP′ a
gp2gp′ (End x)   = Return x
gp2gp′ (Get f)   = Get′ ≫= λx → gp2gp′ (f x)
gp2gp′ (Put n k) = Put′ n ≫ gp2gp′ k
```

Define a transformation in the other direction, i.e., a function

```
gp′2gp :: GP′ a → GP a
```

such that $gp'2gp \circ gp2gp' \equiv id$ for all values that do not contain $\bot$. Does $gp2gp' \circ gp'2gp$ also yield the identity? [No formal proof is required.] •

Solution 22. In this version, the already defined monadic structure on *GP* is used. It is, of course, possible to expand the definitions of *return*, (≫=), *get* and *put*.

```
gp′2gp (Return x) = return x
gp′2gp (Bind m f) = gp′2gp m ≫= λx → gp′2gp (f x)
gp′2gp Get′       = get
gp′2gp (Put′ x)   = put x
```

The composition $gp2gp' \circ gp'2gp$ is not the identity. As an example, consider the expression *Bind* (*Return* 0) *Put′* :: *GP′* (). A formal proof was not required, but here it is anyway:

```
   (gp2gp′ ∘ gp′2gp) (Bind (Return 0) Put′)
≡    { definition of (∘) }
   gp2gp′ (gp′2gp (Bind (Return 0) Put′))
≡    { definition of gp′2gp }
   gp2gp′ (gp′2gp (Return 0) ≫= λx → gp′2gp (Put′ x))
≡    { definition of gp′2gp, twice }
   gp2gp′ (return 0 ≫= λx → put x)
≡    { definition of return for GP }
```

$$
\begin{array}{ll}
 & \mathit{gp2gp}'\ (\mathit{End}\ 0 \gg\!= \lambda x \to \mathit{put}\ x) \\
\equiv & \{\ \text{definition of } (\gg\!=) \text{ for } \mathit{GP}\ \} \\
 & \mathit{gp2gp}'\ ((\lambda x \to \mathit{put}\ x)\ 0) \\
\equiv & \{\ \text{beta-reduction}\ \} \\
 & \mathit{gp2gp}'\ (\mathit{put}\ 0) \\
\equiv & \{\ \text{definition of } \mathit{put} \text{ for } \mathit{GP}\ \} \\
 & \mathit{gp2gp}'\ (\mathit{Put}\ 0\ (\mathit{End}\ ())) \\
\equiv & \{\ \text{definition of } \mathit{gp2gp}'\ \} \\
 & \mathit{Put}'\ 0 \gg \mathit{gp2gp}'\ (\mathit{End}\ ()) \\
\equiv & \{\ \text{definition of } (\gg)\ \} \\
 & \mathit{Put}'\ 0 \gg\!= \lambda\_ \to \mathit{gp2gp}'\ (\mathit{End}\ ()) \\
\equiv & \{\ \text{definition of } (\gg\!=) \text{ for } \mathit{GP}'\ \} \\
 & \mathit{Bind}\ (\mathit{Put}'\ 0)\ (\lambda\_ \to \mathit{gp2gp}'\ (\mathit{End}\ ())) \\
\equiv & \{\ \text{definition of } \mathit{gp2gp}'\ \} \\
 & \mathit{Bind}\ (\mathit{Put}'\ 0)\ (\lambda\_ \to \mathit{Return}\ ())
\end{array}
$$

The result $\mathit{Bind}\ (\mathit{Put}'\ 0)\ (\lambda\_ \to \mathit{Return}\ ())$ is not equal to $\mathit{Bind}\ (\mathit{Return}\ 0)\ \mathit{Put}' :: \mathit{GP}'\ ()$.
○

23 (4 points)**.** Do the monad laws hold for *GP* and GP'? [Give a counterexample if not, argue briefly if yes – no formal proof is required.]

Describe advantages and disadvantages of the two variants. ●

Solution 23. It is easy to see that GP' does not fulfill the monad laws. The monad laws require that

$$\mathit{return}\ 0 \gg\!= \mathit{put} \equiv \mathit{put}\ 0$$

However, $\mathit{Bind}\ (\mathit{Return}\ 0)\ \mathit{Put}'$ is not equal to $\mathit{Put}'\ 0$.

On the other hand, *GP* fulfills the monad laws. Here is a proof for completeness (even though it was not required):

We first show that

$$\forall x\, f.\quad \mathit{return}\ x \gg\!= f \equiv f\ x$$

We can prove this directly, without induction:

$$
\begin{array}{ll}
 & \mathit{return}\ x \gg\!= f \\
\equiv & \{\ \text{definition of } \mathit{return} \text{ for } \mathit{GP}\ \} \\
 & \mathit{End}\ x \gg\!= f \\
\equiv & \{\ \text{definition of } (\gg\!=) \text{ for } \mathit{GP}\ \} \\
 & f\ x
\end{array}
$$

The next law requires that

$$\forall m.\quad m \gg\!= \mathit{return} \equiv m$$

We prove this by structural induction on m:
Case $m = End\ x$:

$$\begin{array}{ll}
 & End\ x \gg\!= return \\
\equiv & \{ \text{ definition of } (\gg\!=) \text{ for } GP \ \} \\
 & return\ x \\
\equiv & \{ \text{ definition of } return \text{ for } GP \ \} \\
 & End\ x
\end{array}$$

Case $m = Get\ g$:

$$\begin{array}{ll}
 & Get\ g \gg\!= return \\
\equiv & \{ \text{ definition of } (\gg\!=) \text{ for } GP \ \} \\
 & Get\ (\lambda n \to g\ n \gg\!= return) \\
\equiv & \{ \text{ induction hypothesis } \} \\
 & Get\ (\lambda n \to g\ n) \\
\equiv & \{ \text{ eta-reduction, i.e., rewrite to point-free form } \} \\
 & Get\ g
\end{array}$$

Case $m == Put\ n\ x$:

$$\begin{array}{ll}
 & Put\ n\ x \gg\!= return \\
\equiv & \{ \text{ definition of } (\gg\!=) \text{ for } GP \ \} \\
 & Put\ n\ (x \gg\!= return) \\
\equiv & \{ \text{ induction hypothesis } \} \\
 & Put\ n\ x
\end{array}$$

As final step, we have to show the associativity of $(\gg\!=)$, i.e., that

$$\forall m\ f\ h. \quad (m \gg\!= f) \gg\!= h \equiv m \gg\!= (\lambda x \to f\ x \gg\!= h)$$

We show this by induction on m.
Case $m = End\ x$:

$$\begin{array}{ll}
 & (End\ x \gg\!= f) \gg\!= h \\
\equiv & \{ \text{ definition of } (\gg\!=) \text{ for } GP \ \} \\
 & f\ x \gg\!= h \\
\equiv & \{ \text{ abstracting from } x \ \} \\
 & (\lambda x \to f\ x \gg\!= h)\ x \\
\equiv & \{ \text{ definition of } (\gg\!=) \text{ for } GP \ \} \\
 & End\ x \gg\!= (\lambda x \to f\ x \gg\!= h)
\end{array}$$

Case $m = Get\ g$:

$$\begin{array}{ll}
 & (Get\ g \gg\!= f) \gg\!= h \\
\equiv & \{ \text{ definition of } (\gg\!=) \text{ for } GP \ \}
\end{array}$$

$$
\begin{aligned}
& \mathit{Get}\ (\lambda n \to g\ n \gg\!= f) \gg\!= h \\
\equiv\ & \quad \{\ \text{definition of } (\gg\!=) \text{ for } \mathit{GP}\ \} \\
& \mathit{Get}\ (\lambda m \to (\lambda n \to g\ n \gg\!= f)\ m \gg\!= h) \\
\equiv\ & \quad \{\ \text{beta-reduction}\ \} \\
& \mathit{Get}\ (\lambda m \to (g\ m \gg\!= f) \gg\!= h) \\
\equiv\ & \quad \{\ \text{induction hypothesis}\ \} \\
& \mathit{Get}\ (\lambda m \to g\ m \gg\!= (\lambda x \to f\ x \gg\!= h)) \\
\equiv\ & \quad \{\ \text{definition of } (\gg\!=) \text{ for } \mathit{GP}\ \} \\
& \mathit{Get}\ g \gg\!= (\lambda x \to f\ x \gg\!= h)
\end{aligned}
$$

Case $m = \mathit{Put}\ n\ k$:

$$
\begin{aligned}
& (\mathit{Put}\ n\ k \gg\!= f) \gg\!= h \\
\equiv\ & \quad \{\ \text{definition of } (\gg\!=) \text{ for } \mathit{GP}\ \} \\
& \mathit{Put}\ n\ (k \gg\!= f) \gg\!= h \\
\equiv\ & \quad \{\ \text{definition of } (\gg\!=) \text{ for } \mathit{GP}\ \} \\
& \mathit{Put}\ n\ ((k \gg\!= f) \gg\!= h) \\
\equiv\ & \quad \{\ \text{induction hypothesis}\ \} \\
& \mathit{Put}\ n\ (k \gg\!= (\lambda x \to f\ x \gg\!= h)) \\
\equiv\ & \quad \{\ \text{definition of } (\gg\!=) \text{ for } \mathit{GP}\ \} \\
& \mathit{Put}\ n\ k \gg\!= (\lambda x \to f\ x \gg\!= h)
\end{aligned}
$$

This completes the proof of the monad laws for *GP*. ○

24 (4 points)**.** Define type synonyms of kind

$$((* \to *) \to *) \to *$$

and

$$(* \to *) \to (* \to *) \to (* \to *)$$

without using any user-defined **data**types. ●

Solution 24.

type $F\ f = f\ [\,]$
type $G\ f\ g\ x = (f\ x, g\ x)$

$\mathit{Main}\rangle\ :\!\mathit{kind}\ F$
$F :: ((* \to *) \to *) \to *$
$\mathit{Main}\rangle\ :\!\mathit{kind}\ G$
$G :: (* \to *) \to (* \to *) \to * \to *$

○