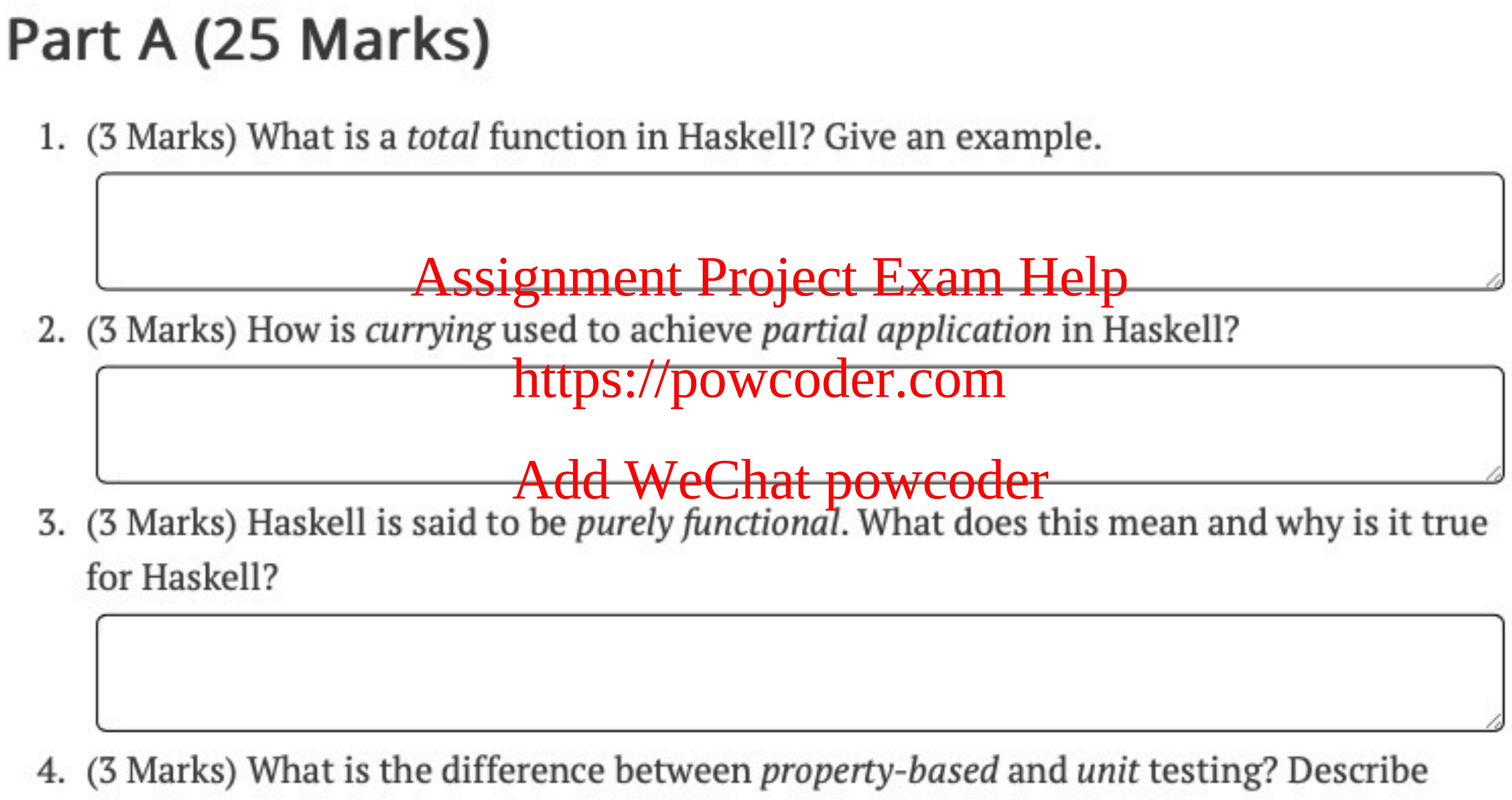

Part A (25 Marks)

1. (3 Marks) What is a *total* function in Haskell? Give an example.

2. (3 Marks) How is *currying* used to achieve *partial application* in Haskell?

3. (3 Marks) Haskell is said to be *purely functional*. What does this mean and why is it true for Haskell?

4. (3 Marks) What is the difference between *property-based* and *unit* testing? Describe

scenarios where each is appropriate.

5. (3 Marks) What is an example of a *static* means of software assurance? Describe how it works.

6. (3 Marks) What does the type `IO`IO accomplish in Haskell?

7. (3 Marks) Write down a simpler type that is isomorphic to `Maybe ()`Maybe ().

8. (4 Marks) Consider the following code.

$$\begin{aligned} sum &:: Int \to Int \to Int \\ sum\ acc\ 0 &= acc \\ sum\ acc\ x &= sum\ (acc + x)\ (x - 1) \end{aligned}$$

$$\begin{aligned} \mathrm{sum} &:: \mathrm{Int} \longrightarrow \mathrm{Int} \longrightarrow \mathrm{Int} \\ \mathrm{sum\ acc}\ 0 &= \mathrm{acc} \\ \mathrm{sum\ acc}\ x &= \mathrm{sum}\ (\mathrm{acc} + x)\ (x - 1) \end{aligned}$$

This function crashes when given a large number for xx, e.g. $sum\ 0\ 1000000$ sum 0 1000000. Why?

Part B (25 Marks)

The following questions pertain to the given Haskell code:

$$\begin{array}{lllll} f :: (Num\ a) \Rightarrow [a] \to a \to a & & & & \\ f\ [\,] & y & = & y & \text{--}\quad (1) \\ f\ (x : xs) & y & = & f\ xs\ (x * y) & \text{--}\quad (2) \end{array}$$

$$\begin{array}{lllll} f :: (\mathrm{Num}\ a) \Rightarrow [a] \longrightarrow a \longrightarrow a & & & & \\ f\,[\,] & \mathrm{y} & = & \mathrm{y} & \text{--}\quad (1) \\ f\,(x : \mathrm{xs}) & \mathrm{y} & = & f\,\mathrm{xs}\ (x * \mathrm{y}) & \text{--}\quad (2) \end{array}$$

1. (3 Marks) State the type, if one exists, of the expression $f\ [0 :: Integer, 4]$ $f\,[0 :: \mathrm{Integer}, 4]$.

2. (4 Marks) Show via step-by-step equational reasoning the evaluation of $f\ [1, 5, 7]\ 1$ $f[1, 5, 7]\ 1$.

3. (2 Marks) In your own words, describe what the function f f does.

4. (12 Marks) Consider the $product$ product function, written in Haskell as follows.

$$\begin{array}{llcll} product & [\,] & = & 1 & \text{-- (A)} \\ product & (x : xs) & = & product\ xs * x & \text{-- (B)} \end{array}$$

```
product   []        =   1                  --   (A)
product   (x : xs)  =   product xs * x     --   (B)
```

We shall prove by induction on lists that, for all lists xs xs and values y y,

$$f\ xs\ y = product\ xs * y$$

f xs y = product xs ∗ y

i. (3 Marks) First show this for the base case where $xs = [\,]$ xs = [] using equational reasoning.

ii. (9 Marks) Next, we have the case where $xs = (k : ks)$ xs = (k: ks) for some item k k and list ks ks.

   a. (3 Marks) What is the *inductive hypothesis* about ks ks?

   b. (6 Marks) Using this inductive hypothesis, prove the above theorem for the inductive case using equational reasoning. You may assume without proof the usual multiplication properties of $(*)$ (∗) (e.g. associativity).

5. (4 Marks) As a consequence of the theorem proven in the previous section, we can now define an alternative form of $product$product:

$$product' \; xs = f \; xs \; 1$$

product ′ xs = f xs 1

If given a very large input list, both $product$product and $product'$product ′ crash, but for different reasons. Why do they crash?

Part C (25 Marks)

The Australian Institute of Ornithology has been studying the singing of the Australian Bush Magpie. They have identified two main sounds, the *caw* and the *cheep*.

$$\textbf{data } Sound = \mathsf{Caw} \mid \mathsf{Cheep}$$

data Sound = Caw | Cheep

As part of their studies, they have recorded extended sequences of Magpie song:

$$[\mathsf{Caw}, \mathsf{Caw}, \mathsf{Cheep}, \mathsf{Cheep}, \mathsf{Cheep}]$$

[Caw, Caw, Cheep, Cheep, Cheep]

These recordings proved to grow to very large sizes, so they decided to employ a simple type of compression. Instead of recording three seperate CheepCheep sounds, they just record the number of times a sound was heard along with the sound. We use the following Haskell data type to represent a compressed sequence of Magpie sounds:

$$\begin{array}{lcl} \textbf{data } Sounds & = & \mathsf{Nil} \\ & \mid & \mathsf{Caws}\ Int\ Sounds \\ & \mid & \mathsf{Cheeps}\ Int\ Sounds \end{array}$$

data Sounds = Nil
| Caws Int Sounds
| Cheeps Int Sounds

For example, the compressed version of the example above would be:

$$\mathsf{Caws}\ 2\ (\mathsf{Cheeps}\ 3\ \mathsf{Nil})$$

Caws 2 (Cheeps 3 Nil)

The following function expands this encoding back to the lists of $Sound$Sound seen above:

$$\begin{array}{llcl}
\mathit{expand} :: \mathit{Sounds} \rightarrow [\mathit{Sound}] \\
\mathit{expand} & \mathsf{Empty} & = & [] \\
\mathit{expand} & (\mathsf{Caws}\ n\ r) & = & \mathit{replicate}\ n\ \mathsf{Caw} \mathbin{+\!\!+} \mathit{expand}\ r \\
\mathit{expand} & (\mathsf{Cheeps}\ n\ r) & = & \mathit{replicate}\ n\ \mathsf{Cheep} \mathbin{+\!\!+} \mathit{expand}\ r
\end{array}$$

expand :: Sounds → [Sound]
expand Empty = []
expand (Caws *n* *r*) = replicate *n* Caw ++ expand *r*
expand (Cheeps *n* *r*) = replicate *n* Cheep ++ expand *r*

1. (16 Marks) We would like the encoding of a given sequence to be unique.
   i. (4 Marks) Give two more examples of SoundsSounds values which also expand into the sequence $[\mathsf{Caw}, \mathsf{Caw}, \mathsf{Cheep}, \mathsf{Cheep}, \mathsf{Cheep}]$[Caw, Caw, Cheep, Cheep, Cheep].

   ii. (6 Marks) Which *data invariants* must be maintained to ensure that there is only one SoundsSounds value for a given list of SoundSound? Describe the invariants in informal English.

   iii. (6 Marks) Define a Haskell function $\mathit{wellformed} :: \mathit{Sounds} \rightarrow \mathit{Bool}$ wellformed :: Sounds → Bool which returns `True`True iff the data invariants hold for the input SoundsSounds value. The Haskell code doesn't have to be syntactically perfect, so long as the intention is clear.

2. (9 Marks) The Haskell function $\mathit{take} :: \mathit{Int} \rightarrow [a] \rightarrow [a]$take :: Int → [a] → [a] takes the given number of elements from the beginning of a list. Below is our attempt to define the same function for our SoundsSounds type:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathit{rTake} :: \mathit{Int} \rightarrow \mathit{Sounds} \rightarrow \mathit{Sounds}} \\
\mathit{rTake} & 0 & r & = \ \mathsf{Nil} \\
\mathit{rTake} & m & \mathsf{Nil} & = \ \mathsf{Nil} \\
\mathit{rTake} & m & (\mathsf{Cheeps}\ n\ r) & \\
 & | & m > n & = \ \mathsf{Cheeps}\ n\ (\mathit{rTake}\ (m - n)\ r) \\
 & | & m < n & = \ \mathsf{Cheeps}\ m\ \mathsf{Nil} \\
\mathit{rTake} & m & (\mathsf{Caws}\ n\ r) & \\
 & | & m > n & = \ \mathsf{Caws}\ n\ (\mathit{rTake}\ (m - n)\ r) \\
 & | & m < n & = \ \mathsf{Caws}\ m\ \mathsf{Nil}
\end{array}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\text{rTake} :: \text{Int} \longrightarrow \text{Sounds} \longrightarrow \text{Sounds}} \\
\text{rTake} & 0 & r & = \ \text{Nil} \\
\text{rTake} & m & \text{Nil} & = \ \text{Nil} \\
\text{rTake} & m & (\text{Cheeps}\ n\ r) & \\
 & | & m > n & = \ \text{Cheeps}\ n\ (\text{rTake}\ (m - n)\ r) \\
 & | & m < n & = \ \text{Cheeps}\ m\ \text{Nil} \\
\text{rTake} & m & (\text{Caws}\ n\ r) & \\
 & | & m > n & = \ \text{Caws}\ n\ (\text{rTake}\ (m - n)\ r) \\
 & | & m < n & = \ \text{Caws}\ m\ \text{Nil}
\end{array}
$$

i. (6 Marks) Write a set of properties to specify *functional correctness* of this function. *Hint:* All the other functions you need to define the properties have already been mentioned in this part. It should maintain data invariants as well as refinement from the abstract model.

ii. (3 Marks) Is this function correct? Why or why not?

Part D (25 Marks)

1. (10 Marks) Consider the following type signatures.

$$\begin{array}{l}
\textbf{newtype}\ \mathit{Msg} = \mathsf{MkMsg}\ \mathit{String} \\[1ex]
\mathit{encrypt} :: \mathit{Msg} \rightarrow \mathit{Msg} \\
\mathit{decrypt} :: \mathit{Msg} \rightarrow \mathit{Msg} \\[1ex]
\mathit{size} :: \mathit{Msg} \rightarrow \mathit{Int} \\[1ex]
\mathit{send} :: \mathit{Msg} \rightarrow \mathit{IO}\ ()
\end{array}$$

$$\begin{array}{l}
\textbf{newtype}\ \mathrm{Msg} = \mathrm{MkMsg}\ \mathrm{String} \\[1ex]
\mathrm{encrypt} :: \mathrm{Msg} \rightarrow \mathrm{Msg} \\
\mathrm{decrypt} :: \mathrm{Msg} \rightarrow \mathrm{Msg} \\[1ex]
\mathrm{size} :: \mathrm{Msg} \rightarrow \mathrm{Int} \\[1ex]
\mathrm{send} :: \mathrm{Msg} \rightarrow \mathrm{IO}\ ()
\end{array}$$

We wish to enforce that no message is sent using the *send*send function without being encrypted with the *encrypt*encrypt function first.
Rewrite the above type signatures to enforce this using Haskell's type system. The function *size*size must work with both encrypted and unencrypted messages.
Hint: Attach *phantom* type parameters to the *Msg*Msg type.

2. (15 Marks) We are making a language used to express logical expressions about *Int*Int values.

$$\begin{array}{rcl}
\textbf{data}\ \mathit{Expr} & = & \mathsf{Plus}\ \mathit{Expr}\ \mathit{Expr} \\
& | & \mathsf{Equal}\ \mathit{Expr}\ \mathit{Expr} \\
& | & \mathsf{Or}\ \mathit{Expr}\ \mathit{Expr} \\
& | & \mathsf{Not}\ \mathit{Expr} \\
& | & \mathsf{C}\ \mathit{Int}
\end{array}$$

$$\begin{array}{rcl}
\textbf{data}\ \mathrm{Expr} & = & \mathrm{Plus}\ \mathrm{Expr}\ \mathrm{Expr} \\
& | & \mathrm{Equal}\ \mathrm{Expr}\ \mathrm{Expr} \\
& | & \mathrm{Or}\ \mathrm{Expr}\ \mathrm{Expr} \\
& | & \mathrm{Not}\ \mathrm{Expr} \\
& | & \mathrm{C}\ \mathrm{Int}
\end{array}$$

We wish to enforce that all expressions in this language are *well-typed* by construction. We consider booleans and integers to be two distinct types of expression. For example, the following expression is well-typed:

$$\mathsf{Or}\ (\mathsf{Equal}\ (\mathsf{Plus}\ (\mathsf{C}\ 2)\ (\mathsf{C}\ 2))\ (\mathsf{C}\ 4))\ (\mathsf{Equal}\ (\mathsf{C}\ 1)\ (\mathsf{C}\ 2))$$

Or (Equal (Plus (C 2) (C 2)) (C 4)) (Equal (C 1) (C 2))

But this next expression is not well-typed, because an integer expression (C 4)(C 4) is provided as an argument for OrOr:

$$\mathsf{Or}\ (\mathsf{Equal}\ (\mathsf{C}\ 2)\ (\mathsf{C}\ 2))\ (\mathsf{C}\ 4)$$

Or (Equal (C 2) (C 2)) (C 4)

The constructor EqualEqual works on operands of either type (however they must be the same).

i. (10 Marks) Define a new version of *Expr*Expr which enforces in Haskell's type system that all expressions are well-typed. Use a *generalised algebraic data type (GADT)* with a type parameter, *Expr a*Expr a.

ii. (5 Marks) Define a function to evaluate well-typed expressions to a result:

$$\mathit{eval} :: \mathit{Expr}\ a \rightarrow a$$

eval :: Expr $a \longrightarrow a$

Part E (25 Marks)

1. (10 Marks) Consider the following property about Monads:

$$a \gg\!= \mathit{fmap}\ f \circ \mathit{pure} \equiv \mathit{fmap}\ f\ a$$

$a \gg\!= \text{fmap}\ f \circ \text{pure} \equiv \text{fmap}\ f\ a$

i. (4 Marks) By reasoning equationally in two separate cases for *aa* (JustJust and

NothingNothing), show that this law holds for all *Maybe*Maybe values.

ii. (6 Marks) Using all the laws we have seen for Monads, Applicatives and Functors, show that this property holds for all (law abiding) Monads.
Hint: Use the extra laws that relate Monads to Applicatives and Applicatives to Functors.

2. (10 Marks) Each of the following Haskell programs are intended to serve as logical proofs of propositions. For each program, give the *logical proposition* they prove, or explain why they do not correspond to a logical proof.

i. (2 Marks)

$$f\ (x, (y, z)) = ((x, y), z)$$

$$f\,(x, (y, z)) = ((x, y), z)$$

ii. (2 Marks)

$$f\ g\ (x, y) = g\ x\ y$$

$$f\,g\,(x, y) = g\,x\,y$$

iii. (3 Marks)

$$\begin{aligned} f\ (\mathsf{Left}\ x)\ m &= m\ x \\ f\ (\mathsf{Right}\ x)\ m &= m\ x \end{aligned}$$

$$\begin{aligned} f\,(\text{Left}\ x)\ m &= m\ x \\ f\,(\text{Right}\ x)\ m &= m\ x \end{aligned}$$

iv. (3 Marks)

$$f\ (\mathsf{Left}\ x) = f\ (\mathsf{Right}\ x)$$
$$f\ (\mathsf{Right}\ x) = f\ (\mathsf{Left}\ ())$$

$$f\,(\mathrm{Left}\ x) = f\,(\mathrm{Right}\ x)$$
$$f\,(\mathrm{Right}\ x) = f\,(\mathrm{Left}\ ())$$

3. (5 Marks) Here is a Haskell data type:

$$\begin{array}{lcl} \textbf{data}\ X & = & \mathsf{Just} \\ & | & \mathsf{Have}\ () \\ & | & \mathsf{Fun}\ (\mathsf{Maybe}\ (\mathsf{Maybe}\ \mathsf{Void})) \end{array}$$

$$\begin{array}{lcl} \textbf{data}\ X & = & \mathrm{Just} \\ & | & \mathrm{Have}\ () \\ & | & \mathrm{Fun}\ (\mathrm{Maybe}\ (\mathrm{Maybe}\ \mathrm{Void})) \end{array}$$

Using known type isomorphisms, simplify this type as much as possible.

Part F (1 Mark)

Write a Haiku about functional programming. 0.5 marks if it's funny, 0.5 marks if it is actually a Haiku.

END OF EXAM

(don't forget to save!)

Time Remaining

2h 9m 44s

● ● ● ● ● ● ● ●   ● ● ● ● ● ● ●   ● ● ● ● ●   ● ● ●   ● ● ● ● ● ● ●   ●

Save