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.

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

sum acc x = sum (

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:

$$
\begin{aligned}
&f :: (Num\ a) \Rightarrow [a] \rightarrow a \rightarrow a\\
&f\ []\qquad y = y \qquad \text{-- (1)}\\
&f\ (x : xs)\quad y = f\ xs\ (x * y) \qquad \text{-- (2)}
\end{aligned}
$$

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

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

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 ff does.

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

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

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


We shall prove by inducti and values yy,

$f\ xs\ y = pro$

f xs y = prod

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 kk and list ksks.

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

   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*.

```
data Sound = Caw | Cheep
```


As part of their studies, they h s of Magpie song:


```
[Caw, Caw, Cheep, C
```

[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 Cheep Cheep 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:

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

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

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

```
Caws 2 (Cheeps 3 Nil)
```

Caws 2 (Cheeps 3 Nil)

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

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

```
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 *Sounds*Sounds values which also expand into the sequence [Caw, Caw, Cheep, Cheep, Cheep][Caw, Caw, Cheep, Cheep, Cheep].

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

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

2. (9 Marks) The Haskell function *take* :: *Int* → [*a*] → [*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 *Sounds*Sounds type:

```
rTake :: Int → Sounds → Sounds
rTake   0   r              =   Nil
rTake   m   Nil            =   Nil
rTake   m   (Cheeps n r)
        |   m > n          =   Cheeps n (rTake (m − n) r)
        |   m < n          =   Cheeps m Nil
rTake   m   (Caws n r)
        |   m > n          =   Caws n (rTake (m − n) r)
        |   m < n          =   Caws m Nil
```

```
rTake :: Int → Sounds → Sounds
rTake   0   r              =   Nil
rTake   m   Nil            =   Nil
rTake   m   (Cheeps n r)
        |   m > n          =   Cheeps n (rTake (m − n) r)
        |   m < n          =   Cheeps m Nil
rTake   m   (Caws n r)
        |   m > n          =   Caws n (rTake (m − n) r)
        |   m < n          =   Caws m Nil
```


i. (6 Marks) Write a set *al correctness* of this function. properties *Hint:* Already men*fu*mentioned in this part. It should mainta nts 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.

```
newtype Msg = MkMsg String

encrypt :: Msg → Msg
decrypt :: Msg → Msg

size :: Msg → Int

send :: Msg → IO ()
```

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.

```
data Expr = Plus Expr Expr
          | Equal Expr Expr
          | Or Expr Expr
          | Not Expr
          | C Int
```

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:

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

Or (Equal (C 2) (C 2)) (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 expressions to a result:

$$eval :: Expr\ a \to a$$

eval :: Expr $a \to a$

Part E (25 Marks)

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

$$a \gg\!= fmap\ f \circ pure \equiv fmap\ f\ a$$

$a \gg\!=$ fmap $f \circ$ pure $\equiv$ fmap $f\ a$

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

Nothing Nothing), 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)

ii. (2 Marks)

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

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

iii. (3 Marks)

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

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

iv. (3 Marks)

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

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

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

```
data X = Just
       | Have ()
       | Fun (Maybe (Maybe Void))
```

```
data X = Just
       | Have ()
       | Fun (Maybe (Maybe Void))
```


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!)