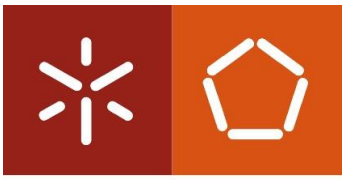

Universidade Minho

Departamento de Informática

Arquitetura e Cálculo
Modelação e análise de sistemas de tempo real utilizando monads e efeitos algébricos

Eduardo Jorge Barbosa A83344
Márcio Sousa A82400

15 de Junho de 2020

Conteúdo

1 Introdução

Este trabalho tem como objectivo modelar o problema da travessia dos aventureiros, previamente resolvido na fase 1. Nesta parte do projecto utilizam-se a *monad* de listas e de durações, em *Haskell*, para implementar efeitos algébricos, tendo como objectivo a resolução do problema segundo este prisma. Às *monads* previamente mencionadas acrescentamos também uma *monad* de *logs* para guardar cada acção. As *queries* implementadas na fase 1 são traduzidas para *Haskell*, com certas limitações, sendo estas limitações discutidas mais à frente na secção 3. De forma a facilitar a análise do problema criou-se 3 interfaces de texto para o utilizador. Na secção 6 é feita uma comparação entre as duas soluções.

2 Modelação em Haskell

Tal como na fase 1, é necessário modelar o problema dos 4 aventureiros. Recordando: '4 aventureiros, com graus de destreza diferentes, têm que atravessar uma ponte frágil, sendo que apenas duas pessoas podem atravessar ao mesmo tempo. O tempo da travessia corresponde ao tempo da pessoa mais lenta. Como restrição adicional, pelo menos um aventureiro terá que levar a lanterna.' A modelação em Haskell, tal como a própria linguagem, tem como por base *monads*. Para este problema são utilizadas duas *monads*:

- **Lista:** modela não-determinismo;
- **Duração:** modela atrasos temporais.

Estas duas *monads* podem ser combinadas numa só *monad* que expressa computações não-deterministicas com uma duração associada. Denominada de **ListDur**, esta *monad* é o busílis deste trabalho.

2.1 Monads

A *monad* de listas e a *monad* de durações podem ser combinadas para gerar uma nova *monad*, a *ListDur* que modela computações não determinísticas. Precisamos destes efeitos algébricos para expressar as varias acções possíveis em cada estado, não-determinismo, e quanto demora essa ação, duração.

2.1.1 Monad de listas

As listas em haskell admitem várias *lawful monads*. A *monad* utilizada é a presente em *Haskell*:

```
return x = [x]
xs >>= f = concat (fmap f xs)
```

O **return** cria uma lista de um só elemento com o valor passado. O **bind** recebe uma lista e uma função que recebe um valor e por sua vez produz uma lista.

```
[a] -> (a -> [b]) -> [b]
```

Aplicando esta função a cada elemento da lista recebida como argumento, são criadas N listas novas, onde N é o comprimento da lista original. Finalmente os dois efeitos monádicos, listas de listas, são colapsados num só.

2.1.2 Monad de durações

A *monad* de durações tem como objectivo associar um custo temporal a um valor, sendo por isso intuitivo representar esta *monad* como o produto $\mathbb{N} * \mathbb{A}$.

O **return** cria uma duração sem nenhum *delay*. O **bind** recebe um valor com uma duração associada e uma função que aplicada a um $\mathbb{A}$, gera outro valor com uma duração associada. A implementação consiste em aplicar a função ao valor original, somar as duas durações e manter o novo valor.

```
data Duration a = Duration (Int, a)

getDuration :: Duration a -> Int
getDuration (Duration (d,_)) = d

getValue :: Duration a -> a
getValue (Duration (_,x)) = x

instance Monad Duration where
    (Duration (i,x)) >>= k = Duration (i + (getDuration (k x)), getValue (k x))
    return x = (Duration (0,x))
```

É possível perceber que esta instância monádica é *lawful* visto que qualquer $\mathbb{B} * \mathbb{A}$ é uma *monad* desde que $\mathbb{B}$ seja um *monoid*. No caso da *monad* das durações, o $\mathbb{B}$ é $\mathbb{N}$ sendo que forma um *monoid* sobre a adição e 0, que é a propriedade utilizada. Esta generalização é provada em anexo nas seccções 8.1 e 8.2.

2.1.3 Monad ListDur

A *ListDur monad* permite ter N valores com uma duração associada.

O **return** cria uma lista com um só valor, com duração 0. O **bind** recebe uma lista de durações e uma função que gera uma lista de durações a partir de um valor. Aplica-se a função recebida a cada valor da lista original e somam-se as durações par a par, mantendo o novo valor. Mantendo o valor mais recente e somando as durações, o **bind** representa a execução sequencial de acções, guardando o estado mais actualizado.

```
data ListDur a = LD [Duration a]

remLD :: ListDur a -> [Duration a]
remLD (LD x) = x

instance Monad ListDur where
   return x = LD [Duration (0, x)]
   l >>= k = LD $ remLD l >>= (\(Duration (s, x)) ->
        fmap (\(Duration (s', z)) -> Duration (s + s', z)) . remLD . k $ x)
```

2.2 Entidades do problema

A linguagem de progamação *Haskell* possui tipos de dados algébricos. Tipos de dados algébricos são, concisamente, construídos a partir de outros tipos de dados, podendo ser *sum types* ou *product types*. Intuitivamente *sum types* representam alternativas, possuindo vários construtores diferentes. *Sum types* agregam vários tipos num só construtor, combinando-os.

Os aventureiros são representados a partir de um *sum type*, sendo que cada construtor tem aridade nula.

```
data Adventurer = P1 | P2 | P5 | P10
```

Por sua vez, a lanterna é representada através do tipo de dados unitário.

É do interesse do problema agregar os aventureiros e a lanterna. Esta agregação é feita utilizando o *Either*, um co-produto geralmente utilizado para representar escolhas.

```
type Objects = Either Adventurer ()
```

Com as entidades do problema codificadas em *Haskell* resta então representar o estado do problema. A ideia é mapear cada objecto para uma margem, uma função do tipo $Objects -> 2$ é a escolha mais intuitiva. O tipo de dados de retorno é uma abstração das margens, sendo que qualquer tipo com 2 habitantes serve. Escolheu-se utilizar os booleanos, sendo que **False** representa a margem inicial (esquerda) e **True** a margem segura (direita).

```
type State = Objects -> Bool
```

O estado inicial é a função $constFalse$.

2.3 Jogadas válidas

Dado um estado é necessário calcular a lista de durações de todas as jogadas possíveis. Isto é, $State -> ListDurState$. A lógica desta função é a seguinte:

1. Calcular em que margem se encontra a lanterna;
2. Enumerar os aventureiros que se encontram na mesma margem que a lanterna;
3. Calcular o produto cartesiano desta lista, sem pares repetidos;
4. Para cada par, alterar o estado dos aventureiros e da lanterna.

Desta forma é possivel assegurar que apenas são calculadas jogadas válidas, isto é:

- Apenas os aventureiros que se encontram na mesma margem que a lanterna podem ter o seu estado alterado;
- No máximo apenas dois aventureiros mudam de estado.

2.4 Execução

Tendo um estado inicial e uma função que dado um estado calcula a lista de jogadas possíveis, é preciso criar uma função que dado um número natural N e um estado, calcula as jogadas possíveis após N jogadas. Utilizando a instância monádica da *ListDur* o código fica bastante conciso.

```
exec :: Int -> State -> ListDur State
exec 0 _ = LD []
exec 1 s = allValidPlays s
exec n s = allValidPlays s >>= exec (n-1)
```

3 Queries

Um dos objectivos deste trabalho é a tradução das *queries* da fase 1 para *Haskell*. Visto que em *Haskell* a estratégia utilizada para o cálculo do estados é *bruteforce* não foi possível traduzir todas as *queries* sem fazer algumas alterações. Os casos mais notórios são a verificação de *deadlocks* e a verificação que quando um aventureiro muda de margem está sempre acompanhado pela lanterna. Para codificar estas verificações foi necessário limitar a um número finito de jogadas possíveis, sendo que em cada jogada a condição é verificada. Todas as restantes *queries* foram codificadas sem qualquer limitação.

3.1 Deadlock

Apesar da limitação da verificação de *deadlocks*, esta *querie* é bastante elegante. A ideia da verificação de *deadlocks* ocorreu a partir da verificação da existência de pontos fixos.

$$fx = x$$

Seja f a função que dado um estado calcula a lisa dos próximos estados possíveis. Se um destes novos estados for igual ao recebido, estamos perante um *deadlock* possível. No entanto dado estarmos a lidar com listas potencialmente infinitas e poder não existir um *deadlock* esta computação pode não terminar. É então necessário limitar a um número finito de jogadas.

3.2 Comparação: Só passam dois aventureiros

Esta *query* exemplifica a diferença das duas plataformas. Em *Haskell* a *query* é bastante concisa mas poderá ser difícil de ler. Já no *UPPAAL* a *query* é longa, chegando a ser dividida em duas, mas é bastante mais acessível.

3.2.1 Haskell

```
{- At most 2 adventurers pass -}
atMost2 :: Int -> Bool
atMost2 n = all max2FromHere allStates where
    allStates = fmap getValue . remLD $ exec n gInit
    max2FromHere :: State -> Bool
    max2FromHere s = all (<= 2) . fmap diff . nextStates $ s where
        diff st = length . filter (False ==) . (zipWith (==) current) $ fmap st
            advNoFlashLight
        current = fmap s advNoFlashLight
        nextStates = fmap getValue . remLD . allValidPlays
```

3.2.2 UPPAAL

```
A[] not ((TheDoctor.ReadyToReturn and RiverSong.ReadyToReturn and
    DonnaNoble.ReadyToReturn and RoseTaylor.ReadyToReturn) or
        (TheDoctor.ReadyToReturn and RiverSong.ReadyToReturn and
            DonnaNoble.ReadyToReturn) or
        (TheDoctor.ReadyToReturn and RiverSong.ReadyToReturn and
            RoseTaylor.ReadyToReturn) or
        (RiverSong.ReadyToReturn and DonnaNoble.ReadyToReturn and
            RoseTaylor.ReadyToReturn))

A[] not ((TheDoctor.ReadytoCross and RiverSong.ReadytoCross and DonnaNoble.ReadytoCross
    and RoseTaylor.ReadytoCross) or
        (TheDoctor.ReadytoCross and RiverSong.ReadytoCross and DonnaNoble.ReadytoCross)
            or
        (TheDoctor.ReadytoCross and RiverSong.ReadytoCross and RoseTaylor.ReadytoCross)
            or
        (RiverSong.ReadytoCross and DonnaNoble.ReadytoCross and RoseTaylor.ReadytoCross))
```

4 Logs

Tendo o problema codificado em *Haskell*, nesta fase é possível fazer asserções sobre o sistema e calcular a enésima jogada mas não é possível ver os passos intermédios. Conseguimos comprovar

que em 17 minutos e 5 jogados é possível que todos os aventureiros cheguem à margem direita mas não conseguimos ver as jogadas que levaram a esse estado.

Guardar os estados intermédios, ou *logs*, é um problema resolvido por outra *monad*, a *writer monad*. O registo de *logs* nada mais é do que acumular valores, tendo um valor inicial "neutro". Estamos perante um *monoid*. Esta *monad* pode ser representada em *Haskell* da seguinte forma:

```
newtype Writer w a =
    Writer { runWriter :: (w, a) }

instance Functor (Writer w) where
    fmap f (Writer (w, a)) = Writer (w, f a)

instance Monoid w => Applicative (Writer w) where
    pure a = Writer (mempty, a)
    (Writer (w, f)) <*> (Writer (w', a)) = Writer (w <> w', f a)

instance Monoid w => Monad (Writer w) where
    return = pure
    (Writer (w, a)) >>= amb = let (Writer (w', a')) = amb a
                              in Writer (w <> w', a')
```

A *writer monad* pode ser combinada com a nossa *ListDur monad* da seguinte forma:

```
newtype ListDurLog w a = LDL [(w, Duration a)] deriving Show

remLDL :: ListDurLog w a -> [(w, Duration a)]
remLDL (LDL x) = x

instance Functor (ListDurLog w) where
   fmap f = LDL . fmap (fmap (fmap f)) . remLDL

instance Monoid w => Applicative (ListDurLog w) where
   pure x = LDL . pure $ (mempty, pure x)
   l1 <*> l2 = LDL $ do
        (w, df) <- remLDL l1
        (w', f) <- remLDL l2
        pure (w <> w', df <*> f)

instance Monoid w => Monad (ListDurLog w) where
   return = pure
   l >>= k = LDL $ remLDL l >>= (\(w, (Duration (s, x))) ->
        fmap (\(w', (Duration (s', z))) -> (w <> w', Duration (s + s', z))) . remLDL . k
            $ x)
```

A prova que se trata de uma *lawful monad* é apresentada em anexo na secção 8.3.

Depois da alteração da nossa *monad* é preciso alterar as funções *exec* e *allValidPlays* para trabalharem com este tipo de dados. Tudo o resto mantém-se igual.

5 Interfaces

Para um melhor entendimento e interpretação dos resultados e do estado actual em que se encontra o modelo, é necessária uma boa interface, que proporcione ao utilizador toda a informação necessária para que este consiga fazer o uso mais eficiente possível do mesmo. Com este objectivo em mente, o programa recorre a 3 funções de IO para apresentar informações diferentes, tirando partido do ambiente interactivo do GHCI.

5.1 allQueries

Esta função é responsável por correr todas as *queries*. Recebe um inteiro como argumento que representa o número de jogadas para limitar algumas das *queries*.

```
*Adventurers> allQueries 5
Is it possible for all adventurers to be on the other side in <=17 minutes and not exceeding 5 moves? True
Is it possible for all adventurers to be on the other side in < 17 minutes and not exceeding 5 moves? False
Is it possible for any adventurer to be on the other side in < their own time? False
Any adventurer must always pass with the flashlight (5 move(s)): True
At most only two adventurers pass (5 move(s)): True
No deadlock possible in 5 moves: True
*Adventurers>
```

Figura 1: allQueries 5

5.2 run

Esta função recebe duas listas de inteiros. A primeira lista representa tempos em minutos e a segunda o número de jogadas. Se for possível que todos os aventureiros passem para o lado direito da ponte no tempo e número de acções dadas, a interface apresenta todas as acções executadas para que se atingisse o objectivo. Em caso de não ser possível, é mostrado "*No Path Available*".

```
*Adventurers> run [17, 18] [5, 5]
P1 and P2 both crossed the bridge
P1 crossed back the bridge
P5 and P10 both crossed the bridge
P2 crossed back the bridge
P1 and P2 both crossed the bridge
 (All in 17 minutes and 5 step(s))

No path available
 (For 18 minutes and 5 step(s))
*Adventurers>
```

Figura 2: run

5.3 interface

Durante a execução de acções sobre o modelo (transições), a interface começa por exibir o tempo passado e número de acções até ao momento, bem como uma breve descrição de todas as acções tomadas. Para facilitar a leitura do estado actual ao utilizador, a interface apresenta também uma pequena representação "gráfica"do problema, onde se vê a ponte, a lanterna e todos os aventureiros no lado da ponte em que se encontram. Por fim apresenta as acções que podem ser tomadas a partir do estado em que se encontra, e o número a ser passado como *input* para a escolha de cada acção possível.

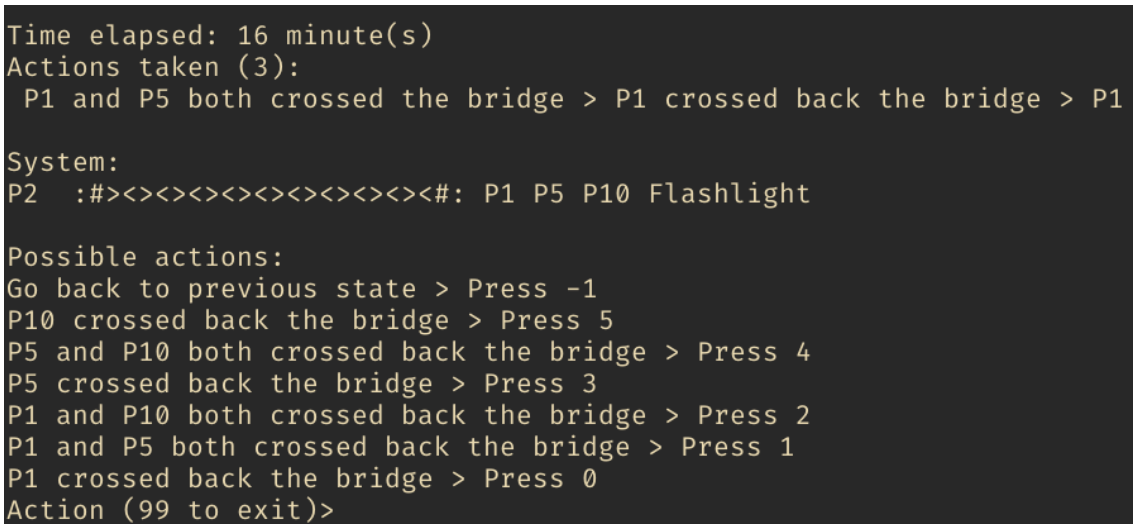

Figura 3: interface

6 Comparação das soluções

A modelação com recurso a Haskell é intuitiva. É uma perspectiva baseada em programação em vez de modelação sendo mais natural para nós. Além disso, a própria linguagem é baseada em *monads*, oferecendo notação muito *user friendly*. Finalmente a mesma linguagem é utilizada para modelar o problema e escrever as *queries*.

Como aspectos negativos, temos que salientar a performance do código e legibilidade do mesmo. Apesar de não ter sido posto um grande esforço para optimizar o código, este exercício é um problema conhecido em Haskell. A comunidade reconhece a dificuldade de raciocinar e optimizar código haskell, principalmente devido à sua natureza *lazy*. Além do mais, devido à abordagem *bruteforce* estamos limitados em termos de performance logo à partida, bem como limitados em algumas *queries* (existência de *deadlocks*).

No que toca a injuntividade e facilidade de leitura de resultados, o *UPPAAL* sai-se melhor, uma vez que proporciona um interface que nos permite modelar o problema seleccionando os componentes (nodos, arestas) e colocando-os da forma mais correcta, tornando-se assim mais intuitivo de utilizar. Em termos de apresentação de resultados, o *Haskell* não permite visualizar tão bem o que se está a passar em cada momento (sem trabalho extra), sendo o *UPPAAL* melhor também neste aspecto.

Relativamente a *queries*, sabemos que o *UPPAAL* tem uma linguagem própria para as mesmas (expressões *CTL*). Estas expressões são bastante optimizadas, aumentando em muito a performance.

Assim sendo, conclui-se que para efeitos de modelação, embora o trabalho possa ser conseguido em *Haskell*,o *UPPAAL* é uma melhor opção valendo o esforço de adaptação a este novo paradigma.

7 Conclusões

O problema apresentado foi implementado em *Haskell* com sucesso. Além disso o grupo conseguiu implementar funcionalidades extras como: registo de *logs*, 3 *interfaces* textuais e *queries* adicionais. Em acréscimo, foi feita ainda uma tentativa de prova que as *monads* utilizadas são *lawful*. O grupo reconhece que esta prova não é formal existindo espaço para erros. No entanto, dado o espaço temporal em que este trabalho se encontra, e a disciplina, consideramos que é um esforço louvável. Para finalizar, deste trabalho conseguiu-se tirar uma generalização: qualquer par cujo primeiro elemento é um *monoid* origina uma *monad*.

Entre ambas as estratégias tomadas (autómatos temporais e *monads*) é agora notório que, embora Haskell/*monads* permita atingir os objetivos, não está ao nível do *UPPAAL* no que diz respeito a este tipo de modelação. Constatou-se que Haskell leva a melhor sobre o *UPPAAL* apenas no sentido de ser mais natural para alguém que esteja familiarizado com programação neste paradigma. No entanto, após alguma prática com o *UPPAAL* este é claramente mais simples e intuitivo de utilizar e, assim sendo, conclui-se que a aproximação por *UPPAAL*, na modelação de problemas, é melhor do que a aproximação utilizando *Haskell*.

8 Anexos

8.1 Prova que um produto de um *monoid* com outro tipo é um *monad*

Assumindo que $\mathbb{B}$ é um *monoid*.

8.1.1 Functor

```
fmap id == id
   { def }
 (i, id x) == id ( (i, x))
   { nat-id }
 (i, x) == (i, x)
   { reflexivity }
True
```

```
fmap (f . g) == fmap f . fmap g
   { def x2 }
 (i, f (g x)) == fmap f ( (i, g x))
   { def }
 (i, f (g x)) == (i, f (g x))
   { reflexivity }
True
```

8.1.2 Applicative

```
Identity
pure id <*> v == v
   { pure def }
 (mempty, id) <*> v == v
   { v = (j, x) }
 (mempty, id) <*> (j, x) == (j, x)
   { <*> def }
 (mempty <> j, id x) == (j, x)
   { nat-id, mempty <> x = x }
 (j, x) ==  (j, x)
```

```
    { reflexivity }
True
```

```
Homomorphism
pure f <*> pure x == pure (f x)
    { pure def x3 }
 (mempty, f) <*> (mempty, x) == (mempty, f x)
    { <*> def, mempty <> mempty = mempty }
 (mempty, f x) == (mempty, f x)
    { reflexivity }
True
```

```
Interchange
u <*> pure y == pure ($ y) <*> u
    { pure def x2, ($ y) = \f -> f y }
u <*>  (mempty, y) == (mempty, (\f -> f y)) <*> u
    { u = (j, u) }
 (j, u) <*> (mempty, y) == (mempty, (\f -> f y)) <*> (j, u)
    { <*> def x2, mempty <> x = x, x <> mempty = x, function application}
 (j, u y) == (j, u y)
    { reflexivity }
True
```

```
Composition
pure (.) <*> u <*> v <*> w == u <*> (v <*> w)
    {pure def}
 (mempty, (.)) <*> u <*> v <*> w == u <*> (v <*> w)
    { <*> def x2, mempty <> w = w, u = (d0, u), v = (d1, v), w = (d2, w) }
 (d0, (u .)) <*> (d1, v) <*> (d2, w) == (d0, u) <*> (d1 <> d2, v w)
    { <*> def x3}
 (d0 <> d1 <> d2, (u . v) w) == (d0 <> (d1 <> d2), u (v w))
    { a <> (b <> c) = a <> b <> c, (f . g) v = f (g v) }
 (d0 <> d1 <> d2, (u . v) w) == (d0 <> d1 <> d2, (u . v) w)
    { reflexivity }
True
```

8.1.3 Monad

```
Left Identity
return a >>= f == f a
    {return def}
 (mempty, a) >>= f a = f a
    { >>= def }
 (mempty <> (\(d,_) -> d) (f a), (\(_,v) -> v) (f a)) = f a
    { function application, f a = (x, v), mempty <> b = b }
 (x, v) == (x, v)
    { reflexivity }
True
```

```
Right Identity
m >>= return == m
    { m = (d, m) }
 (d, m) >>= return == (d, m)
```

```
    { >>= def, return def }
 (d, m) == (d, m)
    { reflexivity }
True
```

```
Associativity
(m >>= f) >>= g == m >>= (\x -> f x >>= g)
    { m = (d, m), f x = (d0, b), >>= def }
(d <> d0, b) >>= g == (d, m) >>= (\x -> f x >>= g)
    { g x = (d1, v), >>= def x3}
(d <> d0 <> d1 , v) == (d <> d0 <> d1 , v)
    { reflexivity }
True
```

8.2 Duration

O tipo de dados *Duration* é isomórfico ao par previamente provado.

```
from :: Duration x > (Int, x)
from (Duration x) = x

to :: (Int, x) -> Duration x
to = Duration
```

O *mempty* corresponde ao 0 e o <> corresponde à adição. Além desta especialização são utilizadas *data unwrappers* para lidar com o *type constructor*.

8.3 ListDur

8.3.1 Functor

```
fmap id == id
    { def }
LDL . fmap (fmap (fmap id)) . remLDL == id
    { valids fmap }
LDL . id . remLDL == id
    { LDL . remLDL = id, nat-id }
id == id
    { reflexivity}
True
```

```
fmap (f . g) == fmap f . fmap g
    { fmap def }
fmap (f . g) == LDL . fmap (fmap (fmap f)) . remLDL . LDL . fmap (fmap (fmap g)) . remLDL
    { LDL . remLDL = id, nat-id }
fmap (f . g) == LDL . fmap (fmap (fmap f)) . fmap (fmap (fmap g)) . remLD
    { fmap (f . g) = fmap f . fmap g x3 }
fmap (f . g) == LDL . fmap (fmap (fmap f .g)) . remLDL
    { fmap def }
fmap (f . g) == fmap (f . g)
    { reflexivity }
True
```

8.3.2 Applicative

```
Identity
pure id <*> v == v
    { pure def x2 }
LDL [(mempty, (0, id))] <*> v == v
    { <*> def x2 }
forall (w, (d, v')) in v => LDL [(mempty <> w, (0 + d, id v'))] == v
    { mempty <> z = z, nat id, 0 + d = d }
forall (w, (d, v')) in v => LDL [(w, (d, v')] == v
    { LDL [(w, (d, v'))] = v }
v == v
    { reflexivity }
True
```

```
Homomorphism
pure f <*> pure x == pure (f x)
    { pure def x2 }
LDL [(mempty, (0, f))] <*> LDL [(mempty, (0, x))] == pure (f x)
    { <*> def, pure def }
LDL [(mempty, (0, f x))] == LDL [(mempty, (0, f x))]
    { reflexivity }
True
```

```
Interchange
u <*> pure y == pure ($ y) <*> u
    { pure def x2, ($ y) = \f -> f y }
u <*> LDL [(mempty, (0, y))] == LDL [(mempty, (0, (\f -> f y)))] <*> u
    { <*> def x2, func ap, mempty <> w = w, d + 0 = d }
forall (w, (d, u')) in u => LDL [(w, (d, u' y))] == LDL [(w, (d, u' y)))]
    { reflexivity }
True
```

```
Composition
pure (.) <*> u <*> v <*> w == u <*> (v <*> w)
    { pure def }
LDL [(mempty, (0, (.)))] <*> u <*> v <*> w == u <*> (v <*> w)
    { <*> def , mempty <> w = w, d + 0 = d }
forall (w', (d', u')) in u => LDL [(w', (d', (u' . )))] <*> v <*> w == u <*> (v <*> w)
    { <*> def, uap = LDL [(mempty, (0, (.)))] <*> u }
forall (w', (d', (u' . ))) in uap, forall (w'', (d'', v')) in v => LDL [(w' <> w'',
     (d'+d'', (u' . v')))] <*> w == u <*> (v <*> w)
    { <*> def, uvap = LDL [(mempty, (0, (.)))] <*> u <*> v }
forall (w' <> w'', (d'+d'', (u' . v'))) in uvap, forall (w''', (d''', w')) in w => LDL
     [(w' <> w'' <> w''', (d'+d''+d''', (u' . v') w'))]
    { (f . g) x = f (g x) }
    { left = LDL [(w' <> w'' <> w''', (d'+d''+d''', u' (v' w')))] }
    { <*> def }
forall (w'', (d'', v')) in v, forall (w''', (d''', w')) in w => u <*> LDL [(w'' <> w''',
     (d''+d''', v' w'))]
    { <*> def, vap = v <*> w }
forall (w', (d', u')) in u, forall (w'' <> w''', (d''+d''', v' w')) in vap => LDL [(w'
     <> w'' <> w''', (d'+d''+d''', u' (v' w')))]
    { right = LDL [(w' <> w'' <> w''', (d'+d''+d''', u' (v' w')))] = left }
    { reflexivity }
```

```
True
```

8.3.3 Monad

```
Left Identity
return a >>= f == f a
    {return def}
 LDL [(mempty, Duration (0, a)] >>= f == f a
    { >>= def, f a = LDL [(w, Duration (x, a))], mempty <> a = a, 0 + d = d }
 LDL [(w, Duration (x, a))] = LDL [(w, Duration (x, a))]
    { reflexivity }
True
```

```
Right Identity
m >>= return == m
    { m = LDL [(w, Duration (d, x))] }
 LDL [(w, Duration (d, x))] >>= return == LDL [(w, Duration (d, x))]
    { reflexivity }
True
```

```
Associativity
(m >>= f) >>= g == m >>= (\x -> f x >>= g)
    { m = LDL [w, Duration (d, x)], f x = LDL [(w0, Duration (d0, b))], >>= def }
LDL [(w <> w0, Duration (d + d0, b))] >>= g == LDL [w, Duration (d, x)] >>= (\x -> f x
     >>= g)
    { g x = LDL [(w1, Duration (d1, v))], >>= def x3}
LDL [(w <> w0 <> w1, Duration (d + d0 + d1 , v))] == LDL [(w <> w0 <> w1, Duration (d +
     d0 + d1 , v))]
    { reflexivity }
True
```