Monad stack van Aff en Location

|  |  |  |
| --- | --- | --- |
| Joop Ringelberg | 27-09-17 | Versie: 1 |

# Introductie

De monad transformer LocationT maakt het mogelijk om een monad stack te maken van bijvoorbeeld Aff en Location. De onderstaande drie types zijn allemaal gebaseerd op Aff. AsyncDomeinFileM combineert de effecten die nodig zijn om (asynchroon) een DomeinFile op te halen. AsyncPropdefsM voegt daar een effect aan toe dat nodig is om Resources op te zoeken in het globale register. En NestedLocation tenslotte, specialiseert AsyncPropdefsM door te stelen dat het variabele type a zich in een Location bevindt.

type AsyncDomeinFileM e = Aff (gm :: GLOBALMAP, avar :: AVAR, ajax :: AJAX | e)

type AsyncPropDefsM e = AsyncDomeinFileM (st :: ST ResourceIndex | e)

type NestedLocation e a = AsyncPropDefsM e (Location a)

De naam NestedLocation is gekozen omdat we hier een monad in een monad hebben genesteld: Location in Aff. In plaats daarvan kunnen we ook monads stapelen. Daar gebruiken we LocationT voor:

type StackedLocation e a = LocationT (AsyncPropDefsM e) a

Merk het verschil tussen NestedLocation en StackedLocation op: de eerste stopt a expliciet in Location, de tweede doet dat impliciet.

# StackedLocation

LocationT (AsyncPropDefsM e) is net zozeer een monad als bijvoorbeeld Aff zelf. De diverse functies die horen bij de instanties van LocationT van Functor, Applicative en Monad, maken en consumeren Locations. Hoe worden die onderling verbonden? Wat zijn de dependencies?

## StackedLocation als Functor

Als functie f wordt gemapt over een StackedLocation, wordt f toegepast op de waarde in de locatie. Het resultaat wordt verpakt in een nieuwe Location. Uiteraard is het eindresultaat van de map diezelfde Location, maar verpakt in Aff.

Deze nieuwe Location is afhankelijk van de oorspronkelijke. Bijvoorbeeld:

stackedLoc\_1 :: StackedLocation Integer

stackedLoc\_1 = pure 1

f :: Integer -> String

f = show

r :: StackedLocation String

r = map f stackedLoc\_1

De afhankelijkheden:

f

map f stackedLoc\_1

## StackedLocation als Applicative

De pure functie verpakt een waarde in een Location en wikkelt die in Aff:

pure :: forall a e. a -> NestedLocation e a

De apply functie neemt een functie en een waarde, beide in NestedLocation. Apply pakt beide uit en houdt dan een functie en een waarde in een Location over. Vervolgens past hij de apply functie van de Apply instantie Location toe op beiden. Het resultaat is een nieuwe Location en die wordt in Aff gewikkeld. Bijvoorbeeld:

stackedLoc\_1 :: StackedLocation Integer

stackedLoc\_1 = pure 1

stackedLoc\_f :: Location (Integer -> String)

f = pure show

r :: StackedLocation String

r = apply stackedLoc\_f stackedLoc\_1

Apply maakt de nieuwe Location afhankelijk van beide argumenten. In beeld:

apply f 1

apply stackedLoc\_f stackedLoc\_1

apply f 1

## StackedLocation als Monad

Bind neemt een waarde in een NestedLocation en past daar een functie van die waarde naar een resultaat in een NestedLocation op toe. De Location die het resultaat bevat, is afhankelijk van de Location die als eerste argument werd meegegeven (gewikkeld in Aff). Bijvoorbeeld:

stackedLoc\_1 :: StackedLocation Integer

stackedLoc\_1 = pure 1

f :: Integer -> StackedLocation String

f = pure <<< show

r :: StackedLocation String

r = bind stackedLoc\_1 f

De afhankelijkheden in beeld:

Bind f

Bind stackedLoc\_1 f

do

x <- stackedLoc\_1

f x

stackedLoc\_1 >>= f

# Het effect van Kleisli composition

Twee functies die een resultaat in de NestedLocation e monad leveren, kunnen met Kleisli composition verbonden worden tot een nieuwe functie:

f :: Integer -> NestedLocation e Integer

f x = x + 1

g :: Integer -> NestedLocation e String

g = show

h :: Integer -> NestedLocation e String

h = f >=> g

y :: NestedLocation e String

y = h (pure 1)

De definitie van Kleisli composition:

composeKleisli :: forall a b c m. Bind m => (a -> m b) -> (b -> m c) -> a -> m c

composeKleisli f g a = f a >>= g

composeKleisli f g a = do

x <- f a

g x

composeKleisli past de Monad bind functie toe. Daaruit leiden we af dat het eindresultaat van (f >=> g) a afhankelijk is van resultaat van f a.

fun x = do

y <- f x

g y

“g”

fun = f >=> g

Is f a afhankelijk van a? Dat hangt af van de implementatie van f. Zo ja, dan is het resultaat van (f >=> g) a indirect afhankelijk van a.

Als we Kleisli compositie meerdere keren toepassen:

fun = f >=> g >=> h

ontstaat een herkenbaar patroon als we fun als volgt schrijven:

fun x = do

y <- f x

z <- g y

h z

Elke regel van de do-constructie representeert een monadische waarde – dus in het geval van NestedLocation óók een Location. De drie locaties zijn onderling afhankelijk zoals weergegeven in het diagram hieronder. Waarom? Wel, omdat >=> rechts associatief is, is f >=> g >=> h eigenlijk f >=> (g >=> h). Hieraan zie je dat het resultaat van de hele expressie toegepast op x, afhankelijk is van de toepassing van f op x – zoals in het diagram.

fun x = do

y <- f x

z <- g y

h z

“h”

## Naamgeving binnen do-expressies

Een do-expressie met n regels wordt door de Purescript compiler herschreven naar n-1 bind expressies, waarbij evenzoveel anonieme functies worden gegenereerd. Dit verklaart waarom het *label* van de verbinding tussen de bovenste twee locaties in het bovenstaande diagram leeg is – een anonieme functie heeft immers geen naam. Bekijk nu de onderstaande twee regels:

f >=> (g >=> h) x

f >=> (p >=> q) x

Het onderstaande diagram geeft de locaties en verbindingen:

fun2 x = do

y <- f x

z <- g y

h z

fun1 x = do

y <- f x

z <- p y

q z

“h”

“q”

“”

“”

Twee labels zijn anoniem! Dit betekent natuurlijk, dat de uitgaande pijlen van de bovenste locatie niet van elkaar onderscheiden kunnen worden. Met andere woorden, zodra fun2 toegepast wordt op x, zal bind ontdekken dat er al een dependency is naar een locatie met hetzelfde label – en volkomen onterecht het resultaat van fun1 teruggeven.

## Name preserving Kleisli compose

De module Location bevat een versie van Kleisli compose die de naam van de eerste functie geeft aan de gehele compositie. Bijvoorbeeld:

f >=> (g >==> h)

geeft het volgende diagram:

f >=> (g >==> h) a

“h”

“g”

# Een functie liften met <$> en <\*>

Een functie van twee argumenten kunnen we als volgt liften naar Applicative waarden:

(+) <$> pure 1 <\*> pure 2

Dit kan uiteraard óók met de Applicative instantie van NestedLocation. Door de opeenvolging van map en apply ontstaat dan een gelaagde structuur van Locations.

1

(+)<$>

(+)<$>1

(+)<$>1<\*>2

(+)<$>1<\*>2

(+)<$>1<\*>2

2

(+) <$> pure 1 <\*> pure 2

Van bijzonder belang is de naamgeving van de verbindingen tussen de locaties. De verbinding tussen het eerste argument en de locatie met de partieel toegepaste functie krijgt een naam volgens het schema:

{naam van de functie}<$>{naam van locatie 1e argument}

De andere twee verbindingen krijgen als naam de volledige expressie:

{naam van de functie}<$>{naam van locatie 1e argument}<\*>{naam van locatie 1e argument}

De naam van de locatie van de partieel toegepaste functie zou voldoende onderscheidend zijn bij de lijst van afhankelijke locaties van de locatie van het tweede argument. Maar dat geldt niet voor de afhankelijke locaties van de partieel toegepaste functie. In het voorbeeld: we kunnen 1 bij oneindig veel andere getallen optellen. Als we voor al die mogelijke sommen het label (+)<$>1<\*> zouden gebruiken, overschreven we telkens de vorige waarde.