Abstract

Toon Nolten

July 2015

<toon.nolten@student.kuleuven.be>

Not concrete

1 Ask not what types can do for you

1.1 Inleiding

Dependent types zijn heel krachtig maar ook heel nieuw en theoretisch Daardoor worden ze nog niet veel gebruikt, voor sommigen is het te intimiderend, voor anderen lijkt het te experimenteel. Met een eenvoudig voorbeeld van het gebruik van dependent types laat ik zien dat ze niet intimiderend moeten zijn en dat hun experimentele aard niet altijd in de weg moet staan.

```
_o_ : {A : Set}{B : A -> Set}{C : (x : A) -> B x -> Set}
      (f : {x : A}(y : B x) -> C x y)(g : (x : A) -> B x)
      (x : A) -> C x (g x)
(f o g) x = f (g x)
```

Het probleem dat we oplossen is het opstellen van red-black trees en de oplossing die we gebruiken is die van Okasaki [1] met aanpassingen waar nodig voor de nieuwe types, op deze manier staat de code niet in de weg van het onderwerp terwijl de toepassing niet te ver afligt van een realistisch probleem.

1.2 Red-black trees in Agda

Het eerste voorbeeld is in Agda [2], een taal die begonnen is met een dependent typesysteem. Als we red-black trees willen maken hebben we een datatype nodig voor die bomen, Okasaki gebruik hier het volgende type dat redelijk typisch is voor haskell:

```
data Color = R | B
data Tree elt = E | T Color (Tree elt) elt (Tree elt)
```

Dit type laat toe om elke geldige boom voor te stellen, jammer genoeg laat het ook toe om ongeldige bomen voor te stellen e.g. `T R (T R E 1 E) 2 E`, `T B (T B E 1 E) 2 E`. Geldige bomen moeten voldoen aan twee invarianten:

1. Geen enkele rode knoop heeft een rode ouder
2. Elk pad van de wortel naar een lege knoop bevat hetzelfde aantal zwarte knopen

(hierbij beschouwen we lege knopen als zwart)

[1] Functional pearl

[2] Agda is functionele programmeertaal met dependent types gebaseerd op martin-löf typetheorie.

In de oplossing van Okasaki moeten we dus altijd goed opletten dat we de invarianten behouden. In Agda daarentegen zijn de types krachtig genoeg om deze invarianten op te nemen. Zo zal het typesysteem ons een foutmelding geven wanneer we een ongeldige boom proberen op te stellen.

Het type voor de bomen in Agda, *A* is het type van de elementen:

```
data Color : Set where
  R B : Color

Height = ℕ

data Tree : Color → Height → Set a where
  E : Tree B 0
  R : ∀{h} → Tree B h → A → Tree B h → Tree R h
  B : ∀{cl cr h} → Tree cl h → A → Tree cr h → Tree B (suc h)
```

In dit type zit naast de kleur van een knoop ook de hoogte (het aantal zwarte knopen tot een lege knoop) vervat. Samen zorgen deze ervoor dat we beide invarianten hebben opgelegd. De eerste invariant is voldaan omdat de enige manier om een rode knoop te maken *R* is en zoals we kunnen zien moeten de kinderen die we aan deze constructor meegeven zwart zijn, we kunnen dus onmogelijk een rode knoop met rode kinderen maken. De tweede invariant is voldaan door te eisen dat kinderen van een knoop van gelijke hoogte moeten zijn, hier *h*.

Omdat de invarianten nu in het type zitten moeten we ons geen zorgen meer maken of we ze wel of niet verbreken, als we ze verbreken krijgen we gewoon een foutmelding bij het type checken. Type checking wordt dus ook meer een deel van de code schrijven. I.p.v. af en toe code te compileren waarbij er dan type checking plaatsvindt, gelijkt het meer op programmeren met een REPL [3] ingebouwd in de editor. Het typesysteem geeft feedback zoals welke gevallen er nog niet behandeld zijn voor de input (alle functies moeten totaal zijn) en geeft ook aan welk type er op bepaalde plaatsen verwacht wordt en welke elementen er in de context beschikbaar zijn om eventueel tot een waarde van dat type te komen.

Door gebruik te maken van een krachtiger type krijgen we alle voordelen van een statische correctheidscontrole, we moeten dus ook niet vertrouwen op de volledigheid van een testsuite. Een preciezer type brengt ook wel nadelen mee, vaak worden tijdens de loop van een functie tijdelijk invarianten verbroken soms voor meerdere functieoproepen bij recursie, zolang het eindresultaat voldoet aan de invarianten is dit aanvaardbaar en soms kan het ook niet anders. Dit is ook het geval in de code van Okasaki, de *ins* functie kan een ongeldige boom teruggeven daarom zijn de recursieve oproepen naar *ins* afgeschermd door *balance*. In de Agda implementatie kan dit niet, het is onmogelijk om een ongeldige boom op te stellen met dit type. Om het *insert* algoritme te kunnen implementeren hebben we dus een extra type nodig dat de ongeldige bomen kan voorstellen. Dit vergt dus meer werk maar we kunnen tegelijkertijd preciezer zijn in hoe de bomen af wijken. Als we dan een verandering doen waardoor de bomen op de verkeerde manier afwijken zal het typesysteem ons waarschuwen.

De afwijkende bomen die we moeten voorstellen hebben een zwarte wortel met een rood kind dat een rood kind heeft. En zulk een afwijkende boom wordt altijd als argument aan *balance* meegegeven. Door hier twee types voor te gebruiken worden sommige dingen eenvoudiger, pattern matching over meerdere niveaus van een boom... geen geneste IR... weten exact waar de R R...

```
data IRTree : Height → Set a where
  IRl : ∀{h} → Tree R h → A → Tree B h → IRTree h
  IRr : ∀{h} → Tree B h → A → Tree R h → IRTree h

data OutOfBalance : Height → Set a where
  _◀_◀_ : ∀{c h} → IRTree h → A → Tree c h → OutOfBalance h
  _▶_▶_ : ∀{c h} → Tree c h → A → IRTree h → OutOfBalance h
```

Omdat we nu soms zowel een geldige als een ongeldige boom kunnen teruggeven of moeten kunnen ontvangen, hebben we nog één extra type nodig:

```
data Treeish : Color → Height → Set a where
  RB : ∀{c h} → Tree c h → Treeish c h
  IR : ∀{h} → IRTree h → Treeish R h
```

[3] Wikipedia?

Nu kunnen we de implementatie van de functies bespreken. De *balance* functie ziet er heel gelijkaardig uit dankzij een voorzichtige formulering van het argumenttype:

```
balance : ∀{h} → OutOfBalance h → Tree R (suc h)
balance (IRl (R a x b) y c ◀ z ◀ d) = R (B a x b) y (B c z d)
balance (IRr a x (R b y c) ◀ z ◀ d) = R (B a x b) y (B c z d)
balance (a ▶ x ▶ IRl (R b y c) z d) = R (B a x b) y (B c z d)
balance (a ▶ x ▶ IRr b y (R c z d)) = R (B a x b) y (B c z d)
```

Wat opvalt is dat de vijfde vergelijking weggevallen is, de *catch-all* in de implementatie van Okasaki. Omdat ons type nu zegt dat we een ongebalanceerde boom moeten hebben, is het niet mogelijk dat we een gebalanceerde boom krijgen die we gewoon terug kunnen geven. Dit zorgt er ook wel voor dat we op de plaats waar we *balance* oproepen, moeten zorgen dat dit ook echt nodig is. (Voordeel bij debugging omdat ge niet moet checken dat er in balance niets gebeurt)

De hulpfuncties voor insert zijn wel een stuk langer geworden. *blacken* moet zowel een geldige als een ongeldige boom kunnen krijgen en de hoogte van een boom kan met één toenemen afhankelijk van de kleur van de wortel van het argument vandaar het conditionele returntype.

```
blacken : ∀{c h} → (Treeish c h)
        → (if c =ᶜ B then Tree B h else Tree B (suc h))
blacken (RB E) = E
blacken (RB (R l b r)) = (B l b r)
blacken (RB (B l b r)) = (B l b r)
blacken (IR (IRl l b r)) = (B l b r)
blacken (IR (IRr l b r)) = (B l b r)
```

Omdat we de gevallen voor geldige en ongeldige, lege, zwarte en rode en links of rechts infrarode bomen apart moeten behandelen is deze functie een stuk langer dan de *one-liner* van Okasaki, dat is de prijs die we betalen voor precisie.

De *ins* functie heeft een extra argument, namelijk het element dat ge-insert moet worden omdat het geen locale definitie is. Het probleem met een locale definitie in agda is dat die maar op één vergelijking van toepassing is. Het returntype is ook preciezer, *ins* geeft ofwel een geldige ofwel een ongeldige boom terug. Het returntype is een dependent pair omdat de kleur van het resultaat zowel rood als zwart kan zijn en dit werkt als een existentiële kwantor.

```
ins : ∀{c h} → (a : A) → (t : Tree c h)
    → Σ[ c' ∈ Color ] (if c =ᶜ B then (Tree c' h) else (Treeish c' h))
ins a E = R , R E a E
--
ins a (R _ b _) with   a ≤ b
ins a (R l _ _) | LT   with ins a l
ins _ (R _ b r) | LT   | R , t = R , IR (IRl t b r)
ins _ (R _ b r) | LT   | B , t = R , (RB (R t b r))
ins _ (R l b r) | EQ   = R , RB (R l b r)
ins a (R _ _ r) | GT   with ins a r
ins _ (R l b _) | GT   | R , t = R , (IR (IRr l b t))
ins _ (R l b _) | GT   | B , t = R , (RB (R l b t))
--
ins a (B _ b _) with   a ≤ b
ins a (B l _ _) | LT   with ins a l
ins _ (B {R} _ b r) |   LT | c , RB t = B , B t b r
ins _ (B {R} _ b r) |   LT | .R , IR t = R , balance (t ◀ b ◀ r)
ins _ (B {B} _ b r) |   LT | c , t = B , B t b r
ins _ (B l b r) | EQ   = B , B l b r
ins a (B _ _ r) | GT   with ins a r
ins _ (B {cr = R} l b   _) | GT | c , RB t = B , B l b t
ins _ (B {cr = R} l b   _) | GT | .R , IR t = R , balance (l ▶ b ▶ t)
ins _ (B {cr = B} l b   _) | GT | c , t = B , B l b t
```

De code is een stuk langer maar dat is vooral omdat we onderscheid moeten maken tussen de constructors voor *Tree*, of het resultaat een geldige of ongeldige boom is en of we balance wel of niet nodig hebben.

Wat niet in de types is opgenomen is de, nogal belangrijke, invariant dat de waarden in een zoekboom gesorteerd moeten zijn...

Agda heeft geen polymorfisme, wat in haskell met polymorfisme wordt gedaan, wordt in Agda gewoonlijk met impliciete typeargumenten bereikt.

```
id :: a -> a
id x = x
```

```
id : {A : Set} → A → A
id x = x
```

Om toe te laten dat de bomen met eender welke vergelijkbare elementen te gebruiken zijn, is de module geparametriseerd met een waarde van het type *StrictTotalOrder*. Zo'n waarde is een record met daarin ondermeer het type (*Carrier*) waarvoor de orde opgesteld is en een vergelijkingsfunctie (*compare*) die bepaald of a < b, a = b of a > b is. Door *A* gelijk te stellen aan *Carrier* en LT, EQ en GT te gebruiken voor de output van *compare* maken we de code toch nog gemakkelijk leesbaar.

1.3 Red-black trees in Haskell

Het tweede voorbeeld is geschreven in Haskell [4], omdat Haskell met een aantal GHC [5] extensies tussen een gewone getypeerde functionele programmeertaal en een dependently typed taal in zit. Wat belangrijk is om te beseffen is dat Haskell al een typesysteem heeft met types, kinds en een sort, wat natuurlijk moet blijven werken en daarbovenop complexe features heeft waar rekening mee gehouden moet worden, e.g. *GADTs* werken niet met *deriving* tenzij de *GADT* syntax in de eerste plaats overbodig was. (Voorbeeld over instance Eq in de code) Met *GADTs* kunnen we essentieel hetzelfde type implementeren voor de bomen als in Agda:

```
data Nat = Z | S Nat deriving (Show, Eq, Ord)

data Color = R | B deriving (Show, Eq, Ord)

data Tree :: Color -> Nat -> * -> * where
  ET :: Tree B Z a
  RT :: Tree B  h a -> a -> Tree B  h a -> Tree R h a
  BT :: Tree cl h a -> a -> Tree cr h a -> Tree B (S h) a
```

Het *Tree* type is de eerste *GADT* die we tegenkomen en hiervoor hebben we meteen twee extensies nodig, namelijk *GADTs* en *KindSignatures*. Een kind signature is als het ware een type signature voor een type. De *GADT* extensie laat ons toe om de constructors verschillende returntypes te geven, e.g. de *ET* constructor creëert een waarde van het polymorfe type *Tree B Z a* terwijl *RT* een waarde opstelt van het polymorfe type *Tree R h a*. In Haskell is er geen concept van een geparametriseerde module dus op die manier kunnen we niet aan het type van de elementen komen, in plaat daarvan maken we gebruik van polymorfisme. We hebben niet alleen het type nodig maar ook een functie om elementen van een type te vergelijken, hiervoor gebruiken we waar nodig de *Ord* constraint. In Haskell kan je niet zomaar kinds definiëren dus waar komen de kinds in de signature voor *Tree* vandaan? Die hebben we te danken aan datatype promotion uit de *DataKinds* extensie. Het type Nat wordt automatisch gepromoveerd to een kind met dezelfde naam, de constructors van dat type worden gepromoveerd tot types met als kind de nieuwgepromoveerde kind. Dit is nodig omdat dependent types draaien rond types die afhangen van waardes en in Haskell is er een strikte scheiding tussen waardes en types, om met typeniveau waardes te kunnen werken, moeten we dus onze waarde constructors spiegelen op typeniveau. De andere types zijn nagenoeg indentiek aan die uit Agda, Haskell heeft geen ondersteuning voor ternaire infix operatoren zoals voor *OutOfBalance* en we moeten steunen op polymorfisme voor het elementtype:

```
data IRTree :: Nat -> * -> * where
  IRl :: Tree R h a -> a -> Tree B h a -> IRTree h a
  IRr :: Tree B h a -> a -> Tree R h a -> IRTree h a

data OutOfBalance :: Nat -> * -> * where
  (:<:) :: IRTree h a -> a -> Tree c h a -> OutOfBalance h a
```

4

5

```
  (:>:) :: Tree c h a -> a -> IRTree h a -> OutOfBalance h a

data Treeish :: Color -> Nat -> * -> * where
  RB :: Tree c h a -> Treeish c h a
  IR :: IRTree h a -> Treeish R h a
```

De *balance* functie is opnieuw nagenoeg indentiek:

```
balance :: OutOfBalance h a -> Tree R (S h) a
balance ((:<:) (IRl (RT a x b) y c) z d) = RT (BT a x b) y (BT c z d)
balance ((:<:) (IRr a x (RT b y c)) z d) = RT (BT a x b) y (BT c z d)
balance ((:>:) a x (IRl (RT b y c) z d)) = RT (BT a x b) y (BT c z d)
balance ((:>:) a x (IRr b y (RT c z d))) = RT (BT a x b) y (BT c z d)
```

blacken daarentegen krijgt een ander returntype. Dezelfde techniek gebruiken als in Agda is zeer lastig in haskell omdat functies op typeniveau, i.e. type families, waardes op typeniveau nodig hebben die in dit geval komen van een andere functie op typeniveau, kortom dit wordt snel heel ingewikkeld. De gemakkelijke oplossing is om gewoon een disjuncte som van de returntypes terug te geven, in Haskell met *Either*:

```
blacken :: Treeish c h a -> Either (Tree B h a) (Tree B (S h) a)
blacken (RB ET) = Left ET
blacken (RB (RT l b r)) = Right (BT l b r)
blacken (RB (BT l b r)) = Left (BT l b r)
blacken (IR (IRl l b r)) = Right (BT l b r)
blacken (IR (IRr l b r)) = Right (BT l b r)
```

Op deze manier verliezen we gedeeltelijk het determinisme over het returntype van *blacken* dus daar waar we *blacken* gebruiken moeten we bereid zijn om een boom van beide types te verwerken. In Haskell geeft dit geen probleem omdat we partiële functies kunnen definiëren, in Agda daarentegen zouden we dan ook de gevallen die eigenlijk niet kunnen voorkomen maar die het type niet uitsluit moeten implementeren, wat soms vervelend en altijd onnodig is.

De *ins* functie heeft opnieuw een disjuncte som als type, deze keer omdat de *Color* existentiëel gekwantificeerd moet zijn, dit werkt dus wel voor een kleur maar zou niet werken voor bvb. natuurlijke getallen: (Existential types zouden kunnen helpen?)

```
ins :: Ord a => a -> Tree c h a -> Either (Treeish R h a) (Treeish B h a)
ins a ET = Left $ RB (RT ET a ET)
--
ins a (RT l b r)
  | a < b , Left (RB t) <- ins a l = Left $ IR (IRl t b r)
  | a < b , Right (RB t) <- ins a l = Left $ RB (RT t b r)
  | a == b = Left $ RB (RT l b r)
  | a > b , Left (RB t) <- ins a r = Left $ IR (IRr l b t)
  | a > b , Right (RB t) <- ins a r = Left $ RB (RT l b t)
--
ins a (BT l b r)
  | a < b , Left (RB t) <- ins a l = Right $ RB (BT t b r)
  | a < b , Left (IR t) <- ins a l = Left $ RB (balance ((:<:) t b r))
  | a < b , Right (RB t) <- ins a l = Right $ RB (BT t b r)
  | a == b = Right $ RB (BT l b r)
  | a > b , Left (RB t) <- ins a r = Right $ RB (BT l b t)
  | a > b , Left (IR t) <- ins a r = Left $ RB (balance ((:>:) l b t))
  | a > b , Right (RB t) <- ins a r = Right $ RB (BT l b t)
```

Hier zien we opnieuw iets wat mogelijk is omdat Haskell partiële functies toelaat: het returntype is altijd een *Treeish* ook al is een zwarte *Treeish* altijd een geldige boom. In Agda zou dit heel vervelend zijn, niet voor het returntype maar wel voor de functie die het resultaat moet verwerken. Agda zou in de recursieve oproepen naar *ins*, in bepaalde gevallen, niet kunnen uitsluiten dat een *IRTree* onmogelijk is en omdat Agda totale definities verwacht, moeten we dan een nutteloze waarde opstellen, wat alleen maar verward als we ooit terug naar de code moeten kijken. (geprobeerd in Treeisher.agda) De recursieve oproepen van *ins* moesten op een heel specifieke manier gebeuren, namelijk in een pattern guard (voorbeelden van hoe het niet werkte...)

De *insert* functie is deze keer nog eenvoudiger, opnieuw moet de recursieve oproep in een pattern guard, de rest is voordehandliggend:

```
insert :: Ord a => a -> Tree c h a -> Either (Tree B h a) (Tree B (S h) a)
insert a t
  | Left t' <- ins a t = blacken t'
  | Right t' <- ins a t = blacken t'
```

(Nog vermelden dat een eenvoudige typeclass instance definiëren, zoals voor Eq, nu verre van triviaal is omdat we de pattern matching een handje moeten toesteken omwille van de vrije constraints op kleur.)

1.4 Conclusie

Het is duidelijk dat dependent types expressiever zijn, ze laten toe om bepaalde invarianten statisch te verifiëren zonder dit voor alle invarianten op te leggen. (vb. okasaki zonder dependent types in agda) In de voorbeelden bvb. zijn de meest kenmerkende invarianten van red-black trees in de types opgenomen maar de belangrijkste niet, namelijk dat de elementen op volgorde moeten zitten. De technieken uit dependently typed talen zijn ook toepasbaar in Haskell, weliswaar in een meer beperkte vorm. De waarde van de statische verificatie moet dus nog beter afgewogen worden.