KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Programmeren en Bewijzen met Dependently Typed Talen

Toon Nolten

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen

Promotoren:

Dr. Dominique Devriese
Prof. dr. ir. Frank Piessens

Assessoren:

Prof. dr. Marc Denecker
Dr. Thomas Heyman

Begeleider:

Jesper Cockx

Academiejaar 2014 – 2015

Voorwoord

Ik heb dit thesisonderwerp gekozen omdat ik erg geïnteresseerd ben in programmeertalen. Van jongs af aan heb ik me altijd afgevraagd hoe dingen werken. Eén van de grootste raadsels was hoe een computer nu eigenlijk werkt. Ik heb redelijk snel geleerd dat een computer eigenlijk niets meer is dan een machine die een stroom van bits omzet in een andere stroom van bits. Niet dat ik daarmee volledig begreep hoe een computer werkt op het niveau van bits en bytes, maar ik kon me er toch iets bij voorstellen. Waar bij mij het interessantste probleem zat was de omzetting van een programma zoals wij het zouden schrijven naar zo een stroom van bits. Stilaan heb ik dan geleerd wat een compiler is en dat bracht eigenlijk alleen meer problemen met zich mee: "Hoe kan de C compiler in C geschreven zijn?" Dit maar om aan te geven dat ik het een fascinerend onderwerp vind. De interesse in dependent types is eigenlijk eerder toevallig gekomen. Ik had de beschrijving van het onderwerp al eens gelezen en daarbij moest ik meteen denken aan hoe het typesysteem van Java mij al had tegengewerkt in vorige opdrachten. Maar ik dacht tegelijk ook aan Haskell wat een veel aangenamer typesysteem heeft. En die dissonantie is eigenlijk de reden dat ik voor dit onderwerp gekozen heb. Waarom is het ene typesysteem eerder aangenaam om mee te werken en het andere niet? En belangrijker nog: "Hoe ver kan een typesysteem gaan?"

Ik wil ook nog een aantal mensen bedanken. In de eerste plaats mijn begeleider Jesper en mijn promotor Dominique, zij hebben mij tegelijk op het juiste pad gezet en mij veel zelfstandigheid gegeven in wat ik juist wou doen en hoe ik het wou doen. Ook heb ik veel gehad aan discussies in de #agda en #haskell IRC kanalen op het freenode netwerk waarbij ik vermijd namen te noemen omdat de kans groot is dat ik iemand vergeet. Verder wil ik mijn vader bedanken die altijd bereid was te luisteren naar een al te vaak ingewikkelde uitleg wanneer ik me een nieuw concept probeerde eigen te maken. Natuurlijk kan ik niet vergeten mijn moeder te bedanken voor het rusteloze aandringen om nu toch eindelijk is aan die thesis te beginnen werken. Tenslotte wil ik mijn grootvaders bedanken die me met hun uitgebreide kennis geïnspireerd hebben om altijd vragen te blijven stellen.

Toon Nolten

Inhoudsopgave

Samenvatting

Dependent types vormen een krachtig middel om bepaalde eigenschappen te coderen in het typesysteem. Op deze manier moeten we niet telkens aan alle eigenschappen proberen denken. Wat de computer voor ons kan doen, moeten we zelf niet doen. In programmeertalen is er al lang een tendens om meer door de computer zelf te laten doen, bijvoorbeeld het geheugenbeheer met garbage collectors. Wat typesystemen betreft is dit jammer genoeg nog niet het geval. Er wordt wel onderzoek verricht maar dit komt zelden terecht in een van de talen die voor alledaagse taken worden gebruikt. Haskell is één van de uitzonderingen op deze regel. Het is een taal die zijn nut in de praktijk al bewezen heeft. En die tegelijkertijd nog het onderwerp is van onderzoek om betere technieken te vinden voor onder andere typesystemen en concurrency. Agda staat in vergelijking nog in de kinderschoenen maar dit maakt het ook mogelijk om op een sneller tempo te experimenteren.

In deze thesis proberen we aan de hand van twee gevalstudies een beter zicht te krijgen op programmeren met dependent types. Waarvoor het nuttig kan zijn: meer concepten uitdrukken op een type safe manier en statisch eigenschappen verifiëren bijvoorbeeld. Wat er aan schort: geen inferentie van types, *verbose*. En hoe een taal die eigenlijk geen dependent types heeft zich verhoudt tot een dependently typed taal.

Hoofdstuk 1

Inleiding

In deze thesis worden twee programmeertalen met een sterk typesysteem vergeleken. Dit met een nadruk op het typesysteem van de talen, eerder dan de talen zelf. Het gaat hier om twee functionele programmeertalen: Agda [1], een dependently typed programmeertaal, en Haskell [2], een taal met een Hindley-Milner [3] typesysteem.

1.1 Typesystemen

Een typesysteem is een verzameling van regels waaraan code in een bepaalde programmeertaal moet voldoen. Er zijn verschillende redenen om zulke regels op te leggen. De belangrijkste is type safety; dit betekent dat de computer nooit onzinnige bewerkingen zal uitvoeren. Wat een onzinnige bewerking juist is, hangt af van het typesysteem. Gewoonlijk worden bewerkingen zoals optelling enkel gedefinieerd op getallen. Een voorbeeld van een onzinnige bewerking zou dan een optelling kunnen zijn van een getal en een lijst. Een andere vorm van bescherming die onder type safety valt is memory safety. Een belangrijke vorm hiervan is bescherming tegen buffer overflows: deze hebben vaak security vulnerabilities tot gevolg maar als het typesysteem over genoeg informatie beschikt, kan gegarandeerd worden dat ze niet kunnen voorkomen. Een andere reden voor een typesysteem is bijvoorbeeld optimalisatie: als de compiler meer informatie heeft kunnen er soms betere optimalisaties gedaan worden. Types kunnen ook een vorm van documentatie zijn: een typesynoniem kan bijvoorbeeld verduidelijken dat een functie een naam verwacht in plaats van gewoon een string, wat het type is voor een opeenvolging van karakters.

Er is een grote variatie aan bestaande typesystemen. Het typesysteem voor de programmeertaal C [4] heeft eenvoudige types die in direct verband staan met de voorstelling van de bijhorende waarden op het niveau van bits en bytes. Het typesysteem van Java [5] daarentegen is volledig gericht op de objectgeoriënteerde metafoor: een object wordt gedefinieerd door een klasse en die klasse *is* tevens het type voor dat object. Het Hindley-Milner typesysteem wordt vaak gebruikt als basis voor het typesysteem van functionele programmeertalen. Het meest merkwaardige kenmerk van dit typesysteem is dat het meest algemene type voor een waarde altijd geïnfereert kan worden. Het resultaat hiervan is dat een programma geschreven

kan worden zonder enige type annotatie en het typesysteem toch kan garanderen dat type safety behouden blijft. Het typesysteem van Haskell is gebaseerd op het Hindley-Milner typesysteem maar heeft een groot aantal uitbreidingen hierop. Zulke uitbreidingen maken het typesysteem expressiever.

1.2 Dependent Types

Dependent types zijn nauw verbonden met de Curry-Howard correspondence [6]. Ze breiden het verband tussen types en propositielogica uit tot een predicatenlogica door het mogelijk te maken om de logische kwantoren voor te stellen. Het verschil met andere typesystemen is dat een dependent type kan afhangen van waarden, zo kan het type van een lijst afhangen van de lengte van die lijst. Deze op het eerste zicht kleine verandering heeft een grote impact op de expressiviteit van het typesysteem. Plotseling wordt het mogelijk om ontzettend veel eigenschappen van de code op te nemen in de types waardoor ze statisch gegarandeerd worden. Schendingen van de eigenschappen die we gecodeerd hebben in de types zijn dus schendingen van de type safety en het typesysteem laat zulke programma's niet toe.

De eigenschappen die we opnemen in de types gaan van heel eenvoudig, bijvoorbeeld de lengte van een lijst, tot eerder ingewikkeld, bijvoorbeeld invarianten van zoekbomen. Fundamenteel is er geen limiet, wat in de logica uit te drukken is, is ook in de types uit te drukken. Echter als we ons bezig houden met programmeren, eerder dan bewijzen formaliseren, zijn de eigenschappen die we wensen af te dwingen vaak redelijk eenvoudig. We gebruiken de types eerder om eigenschappen die we in talen zonder dependent types impliciet laten - en dus in het achterhoofd moeten houden - expliciet te maken en zo door het typesysteem te laten nakijken.

De typentheorie waarop Agda gebaseerd is, gelijkt op de intuïtionistische typentheorie van Per Martin-Löf [7]. In deze thesis laten we dit echter terzijde en bekijken we Agda als een programmeertaal met dependent types in tegenstelling tot Haskell, wat als voorbeeld dient voor een programmeertaal die in de praktijk toegepast wordt voor grootschalige systemen. Deze veralgemening schiet natuurlijk tekort, Agda is geen perfecte voorstelling van *alle* programmeertalen met dependent types en Haskell is niet de enige programmeertaal die op industriële schaal gebruikt kan worden met een sterk typesysteem.

1.3 Vergelijking

De vergelijking tussen Agda en Haskell wordt gemaakt aan de hand van twee concrete gevalstudies. Het gaat hier om twee probleemstellingen die eerst geïmplementeerd zijn in Agda en daarna als het ware vertaald zijn naar Haskell. Dit zorgt ervoor dat Agda als het ware bevoordeeld is om een elegante implementatie te hebben maar dat is irrelevant omdat het niet de bedoeling is een waardeoordeel te vellen. De gevalstudies dienen in de eerste plaats om te illustreren wat we precies met dependent types kunnen uitdrukken dat anders niet in de taal zelf uit te drukken is. De vergelijking tussen de twee talen is er vooral om aan te tonen dat deze manier

van programmeren ook in een courante taal al mogelijk is. In hoofdstuk 2 overlopen we de features die we gebruiken uit Agda en Haskell.

Hoofdstuk 2

Programmeertalen

2.1 Inleiding

In dit hoofdstuk beschrijven we de programmeertalen die gebruikt zijn in de gevalstudies. Er moet ergens een grens getrokken worden die bepaald wat wel en niet uitgelegd wordt, deze thesis veronderstelt dat de lezer vertrouwd is met getypeerd functioneel programmeren. Omdat Agda minder bekend is, is de uitleg hierover uitgebreider dan die over Haskell. Het enige dat we voor Haskell moeten uitleggen zijn namelijk de extensies van de Glasgow Haskell Compiler [8], GHC, die we gebruiken. Merk op dat overal waar code getoond wordt, het symbool ↪ gebruikt wordt om aan te duiden dat een regel die te lang was op die regel verder gaat, dit resulteert niet altijd in geldige code dus bij het overnemen moet hierop gelet worden.

2.2 Agda

Agda is een dependently typed programmeertaal. Talen met dependent types zijn dankzij de Curry-Howard correspondence ook bruikbaar als bewijsassistent. In vele andere talen met dependent types ligt de nadruk ook eerder op het gebruik als bewijsassistent dan wel als programmeertaal.

2.2.1 Dependent Types

Het belangrijkste verschil tussen Agda en andere functionele programmeertalen is het typesysteem. Een dependent type is een type dat afhangt van een waarde. De voorgaande zin legt eigenlijk alles uit wat belangrijk is aan dependent types maar iemand die nog niet vertrouwd is met dependent types zal hier weinig van opsteken. Vandaar overlopen we een aantal voorbeelden in Agda: eerst om de syntax te verduidelijken, daarna om te illustreren wat dependent types nu eigenlijk zijn.

De definitie van nieuwe types gebeurt gelijkaardig als in andere getypeerde functionele programmeertalen. Een `data` sleutelwoord wordt gevolgd door de naam van het type, een dubbelepunt, het type van het type, dan het sleutelwoord `where`

en op de volgende regels de constructors met hun type. Nemen we als voorbeeld een definitie voor natuurlijke getallen:

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

Dit geeft ons een unaire voorstelling van de natuurlijke getallen, het getal 2 stellen we bijvoorbeeld voor als volgt: `suc (suc zero)`. Een verschil met andere talen is dat we het type van het type moeten opgeven. In dependently typed talen is de scheiding tussen types en waarden fundamenteel opgeheven. Het type van de meeste eenvoudige types is in Agda `Set`, wat eigenlijk $\texttt{Set}_0$ is. Het type van $\texttt{Set}_0$ is $\texttt{Set}_1$ en deze hiërarchie gaat in theorie oneindig ver door. Haskell heeft gelijkaardige concepten maar daar gaat de hiërarchie niet erg ver door. De hiërarchie gaat als volgt in Haskell, met de Engelse termen omdat ze niet allemaal een goede vertaling hebben: een value heeft een type, een type heeft een *kind* [9], een *kind* heeft een sort en hier stopt de hiërarchie. Het dichtste equivalent van `Set` is in Haskell het *kind* $*$, $*$ heeft nog sort `BOX` maar `BOX` heeft zelf sort `BOX`. Merk op dat `BOX` enkel en alleen een intellectueel concept is, het is niet uit te drukken in Haskell zelf. Het tweede verschil met een type declaratie in een taal zoals Haskell is dat we het type van elke constructor moeten opgeven, voor `Nat` is dit heel eenvoudig. In het volgende voorbeeld wordt duidelijker waarom we deze types moeten specificeren.

Dit voorbeeld wordt vaak gebruikt om het concept van dependent types te illustreren. Deze code komt uit de Agda Standard Library [10] maar is licht aangepast om het voorbeeld eenvoudiger te maken.

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

head : ∀ {n} {A : Set} → Vec A (suc n) → A
head (x :: xs) = x
```

`Vec` is het type voor lijsten met een vaste lengte, vanaf nu noemen we dit vectors. Hier zien we twee nieuwe dingen, het type `Vec` verwacht nog twee argumenten. Het eerste is de parameter voor de dubbelepunt, `A` met als type `Set` dus `A` is een eenvoudig type. Het tweede is de index van het type $\mathbb{N}$, dit is de naam van het type voor natuurlijke getallen uit de standard library, het verschil tussen een index en een parameter is dat een index voor elke constructor kan variëren terwijl een parameter voor alle constructors hetzelfde is. Een type met indices noemen we ook wel een inductive family [11]. De lege vector, `[]`, heeft lengte nul, het type is `Vec A zero`: het is dus een vector van elementen van type `A` met lengte `zero`. De constructor die langere vectoren maakt, verwacht een element, `x` van type `A`, een vector met elementen van hetzelfde type `A` en een bepaalde lengte `n`, namelijk `xs` met als type `Vec A n`, en geeft een vector terug waarvan de lengte één groter is, `Vec A (suc n)`. De `head` functie, die het eerste element van een vector terug geeft kan dan eisen

dat ze enkel werkt op vectoren met een lengte groter dan nul, vectoren met type `Vec A (suc n)`. Als de lengte niet in het type opgenomen is, kan een functie er ook geen voorwaarden aan opleggen. Zo moet de `head` functie voor gewone lijsten in Haskell een fout opwerpen wanneer ze opgeroepen wordt met een lege lijst als argument. Dit is een heel eenvoudig voorbeeld en hieruit blijkt niet welke van de twee een betere manier is om lijsten voor te stellen. Het laat wel zien wat het betekent voor een type om af te hangen van een waarde. Wat we ook zien, zowel in het type van `_::_` als `head` is `∀ {n}`, de universele kwantor zorgt dat `n` eender wat kan zijn zolang het voldoet aan de eisen in de rest van het type. De accolades rond `n` en in het type van `head` ook rond het type van het volgende argument, `{A : Set}`, duidt aan dat deze argumenten impliciet zijn, ze worden afgeleid uit de context, in dit geval bijvoorbeeld uit het type van het vector argument. Er is nog een ander soort impliciete argumenten in Agda, *instance arguments*, deze worden aangeduid met dubbele accolades en worden op een andere manier ingevuld, de typechecker zoekt naar een concrete *instance* in de context, als er verschillende mogelijkheden zijn leidt dat tot een typefout. Oorspronkelijk zijn *instance arguments* toegevoegd aan Agda als een eenvoudig alternatief voor typeclasses uit Haskell.

Een meer uitgebreid voorbeeld uit een artikel dat het nut van dependent types goed uitlegt [12], laat zien dat dependent types zeer expressief zijn. Het type `RA` laat toe om een relationele algebra expressie op te stellen die correct is door constructie. Het artikel beargumenteert dat het in Haskell niet mogelijk is om een interface naar een database te voorzien die evenveel statische garanties biedt en even volledig - joins zijn zonder dependent types moeilijk te typeren - is zonder gebruik te maken van preprocessing of experimentele features. Dit betekent niet dat er geen libraries bestaan voor Haskell die bepaalde eigenschappen statisch of dynamisch opleggen, maar wel dat die beperkter zijn en minder statisch kunnen zijn. Statische correctheid is handiger omdat ze geen uitvoerige testen noodzakelijk maakt.

```
data RA : Schema → Set where
  Read    : ∀ {s} → Handle s → RA s
  Union   : ∀ {s} → RA s → RA s → RA s
  Diff    : ∀ {s} → RA s → RA s → RA s
  Product : ∀ {s s'} → {_ : So (disjoint s s')} → RA s → RA s'
          → RA (append s s')
  Project : ∀ {s} → (s' : Schema) → {_ : So (sub s' s)} → RA s → RA s'
  Select  : ∀ {s} → Expr s BOOL → RA s → RA s
```

Het type `Schema` stelt een schema van een relatie voor, door dit als index op te nemen, kunnen we bepaalde eisen stellen aan de schema's waarop een relationele expressie werkt. De `Read` constructor gelijkt een beetje op een lege lijst omdat die aan de basis van elke relationele expressie moet liggen, voor alle andere constructors van een relationele expressie hebben we al een relationele expressie nodig. `Read` bevat de informatie die aangeeft in welke database de relatie te vinden is in een `Handle`. Zo'n `Handle` heeft tevens een schema als index en kunnen we enkel opstellen als de database inderdaad een relatie heeft die aan het schema voldoet. Voor de unie van twee relaties, `Union`, en het verschil tussen twee relaties, `Diff`, eisen we

dat de schemas overeenkomen: een relatie kan maar een unie zijn van twee relaties als alle attributen overeenkomen. Eigenlijk moeten de schema's niet hetzelfde zijn maar mogen ze permutaties van elkaar zijn maar dit kan opgelost worden door de implementatie van `Schema` als een data type dat dezelfde vorm heeft in welke volgorde de elementen ook worden toegevoegd. De constructor voor het cartesisch product, `Product`, heeft een impliciet argument van het type `So (disjoint s s')`, de underscore duidt aan dat we het argument niet gebruiken in de rest van het type maar moet er staan omdat we een type van een impliciet argument niet zonder een variabele kunnen opgeven. Het type `So (disjoint s s')` zorgt ervoor dat de schemas `s` en `s'` disjunct zijn omdat een relatie geen twee attributen met dezelfde naam kan hebben en een cartesisch product geen join is. Verder zien we dat `Product` een relationele expressie opstelt waarvan het resulterende schema de combinatie van de disjuncte schema's is. Voor de projectie, `Project`, eisen we dat de attributen die we uit een relatie willen projecteren inderdaad aanwezig zijn in de relatie. Het belangrijkste kenmerk van de `Select` constructor kunnen we helaas niet uitleggen zonder meer van het artikel over te nemen maar is niet belangrijk in de rest van dit werk.

2.2.2 Syntax

Wat ook opmerkelijk is aan Agda, is de vrijheid die er is door de flexibele syntax. Agda legt geen regels op in verband met hoofdletters voor types en constructors en kleine letters voor functies, dit mes snijdt natuurlijk aan twee kanten omdat types en functies minder gemakkelijk uit elkaar te houden zijn. Wat waarschijnlijk de belangrijkste reden is om zulke regels niet af te dwingen is dat er eigenlijk geen verschil is tussen waarden en types, een functie is ook een waarde, het is dus ook niet logisch om artificieel een verschil op te leggen in de schrijfwijze. Wat naamgeving betreft is nog een belangrijk kenmerk dat alle unicode karakters toegelaten zijn. Hier wordt in de standard library ook veel gebruik van gemaakt. Praktisch betekent dit dus dat er goede ondersteuning moet zijn van de tekstverwerker en het lettertype waarin je Agda code schrijft.

Een belangrijker kenmerk van de syntax in Agda is dat types, constructors en functies gedefinieerd kunnen worden met mixfix notatie. Voor vectoren hebben we al een infix constructor gezien, een tweede voorbeeld van zulke notatie is dit:

```
if_then_else_ : {A : Set} → Bool → A → A → A
if true then x else y = x
if false then x else y = y
```

Deze functie kunnen we nu op twee manieren gebruiken. Met prefix notatie als we de naam met underscores overnemen als volgt: `if_then_else_ true x y`. Of zoals in de definitie in de mixfix vorm. Soms maakt dit de code gemakkelijker leesbaar. Wat wel noodzakelijk wordt door zo'n flexibele notatie is het veelvuldig gebruik van spaties, voor `if_then_else_` maakt dit weinig uit want daar zijn de spaties logisch maar stel bijvoorbeeld dat dit een functie is: `[_]`, dan moeten er spaties rond het

argument, `[ x ]`, wat in het begin vreemd aanvoelt maar nodig is omdat `[x]` een geldige naam zou kunnen zijn voor een functie.

Er is nog één expressie die in andere talen gewoonlijk niet voorkomt, de zogenaamde *with* expressie. Hiermee kunnen we een *pattern match* doen op een tussentijds resultaat. In de tweede gevalstudie wordt dit duidelijker door een voorbeeld.

2.2.3 Modules

Agda heeft, net als Haskell, modules om code in te kunnen delen in verschillende logische eenheden die eventueel over verschillende bestanden verdeeld kunnen worden. In Agda kunnen we modules ook parametriseren, dit gaat in Haskell niet maar is wel terug te vinden in bijvoorbeeld de ML [13] [14] familie van talen, die net als Haskell statisch getypeerd en functioneel zijn. Om een geparametriseerde module te gebruiken moeten we er waarden aan meegeven overeenkomstig de types van de parameters. Op deze manier kunnen we er bijvoorbeeld voor zorgen dat een module met sorteerfuncties enkel gebruikt wordt voor elementen die een orderelatie hebben.

2.2.4 Versie

De gebruikte versies van Agda en de standard library in de rest van deze thesis zijn respectievelijk `2.4.2.2` en `0.9`.

2.3 Haskell

Omdat we redelijk veel vergen van het typesysteem, komen we niet toe met standaard Haskell zoals bijvoorbeeld geïmplementeerd in GHC maar hebben we een aantal extensies nodig van GHC.

2.3.1 Waarden op typeniveau

Haskell heeft geen dependent types maar we hebben waarden nodig op het niveau van types, in Haskell moeten we die dus voorstellen door types, en die types moeten zelf een type hebben, wat in Haskell dus een *kind* is. Normaal kunnen we in Haskell geen *kinds* definiëren maar met de extensie DataKinds [15] gaat dit wel. Deze extensie laat ons toe om zelf *kinds* te definiëren met bijhorende types door onze gewone types te promoveren tot *kinds* en hun constructors te promoveren tot types. Een voorbeeld maakt dit duidelijker:

```
data Nat = Z | S Nat

two :: Nat
two = S (S Z)

type_level_two :: S (S Z)
```

We definiëren gewoon een type voor natuurlijke getallen en door de DataKinds extensie wordt dit type een *kind* en de constructors, types. Het *kind* `Nat` is dus

het *kind* van twee types namelijk `Z` en `S Nat`. De `Nat` in `S Nat` is opnieuw het *kind* `Nat` niet het type. Omdat dezelfde namen voor verschillende concepten af en toe verwarrend kunnen zijn en het niet onmogelijk is dat we al een type `Z` gedefinieerd hadden, worden types en constructors ook altijd gepromoveerd tot *kinds* en types die beginnen met een apostrophe. Het type `Nat` wordt bijvoorbeeld altijd gepromoveerd tot de *kind* `'Nat` en de constructor `Z` tot het type `'Z`. Met de DataKinds extensie kunnen we dus waarden voorstellen op typeniveau. Waar de extensie nog tekort schiet is dat de waarde `Z` en het type `Z` volledig los van elkaar staan, buiten de naam. Er is ook geen waarde van het type `Z`. In één van de volgende hoofdstukken gebruiken we een techniek waarmee we de waarden op waardeniveau en typeniveau ongeveer kunnen verbinden.

2.3.2 Inductive families

In Haskell hebben we types die kunnen afhangen van andere types, een voorbeeld hiervan is het `Maybe a` type en dit geeft ons de indices uit Agda. Wat nog mist is de mogelijkheid om het type voor iedere constructor te variëren. Met behulp van generalised algebraic data types [16] uit de GADTs extensie kunnen we dit wel. De KindSignatures extensie laat ons toe om bepaalde termen te annoteren met een *kind* zoals we nu bepaalde termen een type annotatie kunnen geven. Met deze twee extensies kunnen we eenvoudige inductive families implementeren in Haskell, als voorbeeld beschouwen we het type voor vectors:

```
data Vec :: * -> Nat -> * where
  V0 :: Vec a Z
  (:>) :: a -> Vec a n -> Vec a (S n)
```

Het belangrijkste verschil met de definitie in Agda is dat het type van de elementen nu niet hetzelfde zou moeten zijn voor elke constructor. Haskell laat ons bijvoorbeeld toe te zeggen dat `V0` het type `Vec Int Z` heeft en dat `(:>)` een vector met type `Vec Char (S n)` oplevert. We moeten dus zelf beter opletten bij het definiëren van de types van de constructors.

2.3.3 Impliciete resolutie van een relatie

Een relatie tussen twee types kunnen we voorstellen met een nieuw type, laten we als voorbeeld de relatie tussen vlaggen en landen nemen:

```
data Flag = DrieKleur | StarsAndStripes | UnionJack

data Country = Belgium | USofA | UnitedKingdom

data FlagCountry :: Flag -> Country -> * where
  Belg :: FlagCountry DrieKleur Belgium
  Amer :: FlagCountry StarsAndStripes USofA
  Engl :: FlagCountry UnionJack UnitedKingdom
```

Als we nu in een type een vlag kennen en het land willen weten kunnen we dit type gebruiken om dit te weten te komen: `flag -> FlagCountry flag country -> country`, dit is eigenlijk niet geldig in Haskell maar het gaat om het idee. Dit is soms te expliciet omdat we wel in het type van de relatie geïnteresseerd zijn maar niet in de waarde zelf. In Haskell kunnen we gebruikmaken van typeclasses om informatie impliciet te houden, de typeclasses uit standaard Haskell laten maar één parameter toe. Omdat we voor een relatie twee parameters nodig hebben, maken we gebruik van de MultiParamTypeClasses extensie. De FlexibleInstances extensie laat ons toe om type variables te gebruiken in de `instance` declaraties. De bijhorende typeclass voor het vorige voorbeeld zou er als volgt uitzien:

```
class FlgCntry (f :: Flag) (c :: Country)
instance FlgCntry DrieKleur Belgium
instance FlgCntry StarsAndStripes USofA
instance FlgCntry UnionJack UnitedKingdom
```

Het type van een functie die hiervan gebruik maakt, wordt niet minder expliciet maar we moeten geen argument meer meegeven waar weinig informatie in zit: `FlgCntry flag country => flag -> country`, dit is weer geen geldige Haskell code maar het gaat om het idee.

2.3.4 Versie

De gebruikte versie van GHC in deze thesis is `7.6.3`.

2.4 Besluit

Met de kennis van de twee talen die we in dit hoofdstuk opgebouwd hebben, kunnen we beginnen aan de gevalstudies. In de eerste gevalstudie bekijken we een formalisering van de paden die een spelfiguur aflegt om zo fouten te kunnen voorkomen. In de tweede gevalstudie implementeren we een soort zoekbomen, red-black trees, met statische verificatie van de invarianten.

Hoofdstuk 3

Case: Verified Koopa Troopa Movement

3.1 Inleiding

Deze eerste gevalstudie is geïnspireerd door de populaire Mario [17] franchise van Nintendo [18], in het bijzonder Super Mario Bros. [19]. Er zijn verschillende vijanden in het spel maar hier beperken we ons tot de schildpadachtige Koopa Troopas. Zoals wel vaker voorkomt, bevatten vele van de Mariospellen zogenaamde *glitches*, fouten die uitgebuit kunnen worden om bepaalde acties uit te voeren die normaal moeilijk of onmogelijk zijn. Zo is er bijvoorbeeld een fout met een Koopa Troopa waardoor Mario aan het einde van het level over de vlag kan springen, dit is normaal niet mogelijk. De precieze fout is moeilijk uit te leggen maar ze is mogelijk omdat een Koopa Troopa als het ware onder het level kan rondlopen, dit is in figuur 3.1 te zien. Deze fout is eigenlijk de precieze inspiratie van deze gevalstudie.

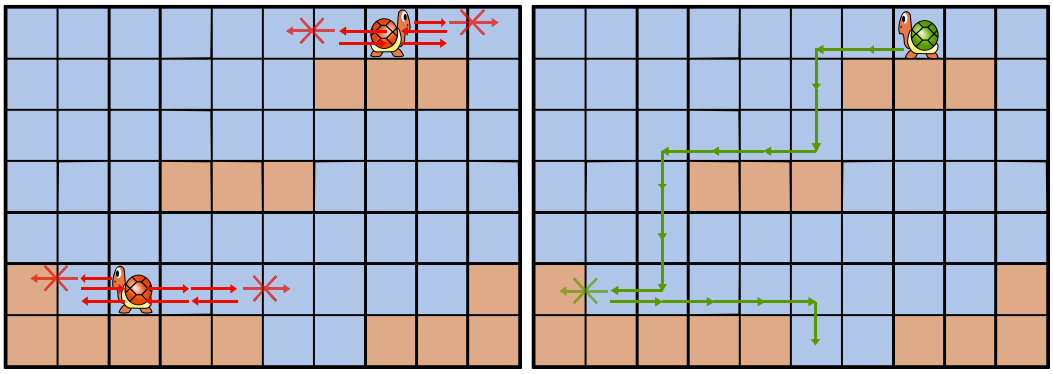
De Koopa Troopas komen voor in twee kleuren: rood en groen. Ze lopen in één richting tot ze een obstakel tegenkomen, dan keren ze om. Voor alle Koopa Troopas is een muur een obstakel, enkel voor de rode Koopa Troopas is een afgrond een obstakel. Dit heeft tot gevolg dat rode Koopa Troopas heen en weer patrouilleren en groene Koopa Troopas eerder trapsgewijs naar beneden vallen tot ze uit het beeld verdwijnen. Door deze regels uit te drukken in de types, in plaats van in de spellogica, kunnen we een fout zoals die in figuur 3.1 te zien is, voorkomen. In figuur 3.2 zijn een aantal voorbeeldpaden geïllustreerd.

3.2 Koopa Troopas in Agda

Om het programmeren met dependent types zo duidelijk mogelijk te illustreren wordt alle code stap voor stap bekeken in begrijpelijke stukken.

De code begint met een module declaratie, deze is niet verplicht maar wel nuttig omdat ze het mogelijk maakt de code te gebruiken in andere programma's. Daarna importeren we een aantal eenvoudige types uit de standard library.

Figuur 3.1: Onder Mario is nog net het hoofd van een Koopa Troopa te zien [20]



Figuur 3.2: Mogelijke paden voor rode en groene Koopa Troopas

```
module koopa where
  open import Data.Nat
  open import Data.Fin renaming (_+_ to _F+_; _<_ to _F<_; suc to
   ↪  fsuc;
    zero to fzero)
  open import Data.Vec renaming (map to vmap; lookup to vlookup;
                                 replicate to vreplicate)
  open import Data.Unit
  open import Data.Empty
```

`Data.Nat` bevat een unaire voorstelling van natuurlijke getallen die we gaan gebruiken voor coördinaten. `Data.Fin` definieert een type voor begrensde natuurlijke getallen, dit gebruiken we zodat we bij index operaties niet buiten het bereik van een lijst kunnen gaan. `Data.Vec` definieert lijsten met een vaste lengte, vectors dus. `Data.Unit` definieert een type, $\top$ ook wel top genoemd, met één waarde namelijk `tt`. En, ten laatste, `Data.Empty` definieert een leeg type, $\bot$ ook wel bottom genoemd, waarvoor dus geen waarden bestaan. Dit is anders dan in Haskell waar je eigenlijk geen leeg type kunt hebben, elk type bevat daar minstens *bottom*, wat verschillende dingen kan betekenen, bijvoorbeeld niet eindigen of een fout in de uitvoering.

Het volgende stuk is een geneste module declaratie waarin het type `Matrix` wordt gedefinieerd als een vector van vectoren en een functie die een element uit een `Matrix` projecteert. `Matrix` is een inductive family zoals eerder besproken in sectie 2.2.1. Omdat `Matrix` hier gedefinieerd is als een vector van vectoren kunnen we voor de `lookup` functie de `lookup` functies voor vectoren gebruiken. De volgorde van de indices speelt hierbij een rol maar omdat de indices van het type `Fin n` zijn, kan Agda een fout geven als we ze omwisselen. Vervolgens wordt de module geopend zodat we het type en de functie kunnen gebruiken zonder de namen te moeten kwalificeren met de naam van de module.

```
  module Matrix where
    data Matrix (A : Set) : ℕ → ℕ → Set where
      Mat : {w h : ℕ} → Vec (Vec A w) h → Matrix A w h

    lookup : ∀ {w h} {A : Set} → Fin h → Fin w → Matrix A w h → A
    lookup row column (Mat rows) = vlookup column (vlookup row rows)
  open Matrix
```

Hierna begint de oplossing van het probleem eigenlijk pas echt. We definiëren een aantal types waarmee we de Koopa Troopas en de levels kunnen voorstellen.

```
  data Color : Set where
    Green : Color
    Red : Color

  data KoopaTroopa : Color → Set where
    _KT : (c : Color) → KoopaTroopa c
```

Koopa Troopas kunnen twee kleuren hebben dus we definiëren een type `Color` met een constructor voor elke kleur. Agda laat ons toe om constructors met hetzelfde type te groeperen als volgt: `Green Red : Color`. Het type `KoopaTroopa` is geïndexeerd op `Color`, dit wil dus zeggen dat er een type is voor elke `Color`. De constructor maakt gebruik van de Agda syntax voor mixfix notatie, in dit geval is `_KT` dus een postfix constructor. De `_KT` constructor verwacht een `Color` en maakt dan een waarde van het type `KoopaTroopa c` waar die `c` dus de waarde van het argument is. Een groene Koopa Troopa kunnen we dus voorstellen als `Green KT` en heeft het type `KoopaTroopa Green`, later wordt duidelijk waarom het belangrijk is dat de kleur in het type zit.

Het volgende stuk bevat nog een aantal type declaraties. Hier definiëren we posities met alle informatie die we later nodig hebben om te bepalen op welke posities een Koopa Troopa mag staan.

```
data Material : Set where
  gas    : Material
  -- liquid : Material
  solid  : Material

data Clearance : Set where
  Low  : Clearance
  High : Clearance
  Ultimate  : Clearance

record Position : Set where
  constructor pos
  field
    x   : ℕ
    y   : ℕ
    mat : Material
    clr : Clearance
```

Elke positie is ofwel lucht, `gas`, ofwel vast, `solid`: grond of muur of iets dergelijks. Het type `Material` stelt deze toestanden voor. Het type `Clearance` gebruiken we om te bepalen wie of wat waar mag komen. Een positie met `Clearance Low` kan door eender welke Koopa Troopa betreden worden maar een positie met `Clearance High` kan enkel door groene Koopa Troopas betreden worden en gebruiken we om posities aan te duiden waar de enige logische beweging vallen is. De `Clearance Ultimate` gebruiken we voor posities waar geen enkele Koopa Troopa mag komen. Een positie stellen we voor door een *record* met twee velden voor een horizontale en een verticale coördinaat, een veld dat het `Material` van de positie aangeeft en een veld voor de `Clearance`. De constructor `pos` kunnen we gebruiken om een positie op te stellen, een voorbeeld van een positie kan dus zijn: `pos 3 5 gas Low`, let hierbij op dat de cijfers door Agda begrepen worden als natuurlijke getallen, hiermee kunnen we ook enkel natuurlijke getallen voorstellen.

Om de Koopa Troopas een `Clearance` te geven moeten we ze op een of andere manier differentiëren en het enige dat verschilt tussen de Koopa Troopas is hun kleur wat dus de meest logische bepalende factor van de `Clearance` wordt.

```
data _c>_ : Color → Clearance → Set where
  <red>   : ∀ {c} → c c> Low
  <green> : Green c> High
```

In plaats van groen en rood gelijk te stellen aan respectievelijk een hoge en een lage `Clearance`, werkt de constructor `<red>` voor beide kleuren terwijl `<green>` enkel voor groen werkt. Dit zorgt ervoor dat we later niet moeten zorgen dat overal waar een lage `Clearance` toegelaten is ook een hoge `Clearance` toegelaten is, een groene Koopa Troopa krijgt gewoon de `Clearance` die hij nodig heeft. We zien hier ook dat de mixfix notatie even goed voor types te gebruiken is, in dit geval gebruiken we dit om het type voor te stellen als een pijl, `_c>_`, van een kleur naar een `Clearance`.

De volgende twee definities zijn eigenlijk het belangrijkste deel van de oplossing en steunen op alle voorgaande concepten.

```
data _follows_⟨_⟩ : Position → Position → Color → Set where
  stay  : ∀ {c x y} → pos x y gas Low follows pos x y gas Low ⟨ c
   ↪ ⟩
  next  : ∀ {c cl x y}{{_ : c c> cl}} →
          pos (suc x) y gas cl follows pos x y gas Low ⟨ c ⟩
  back  : ∀ {c cl x y}{{_ : c c> cl}} →
          pos x y gas cl follows pos (suc x) y gas Low ⟨ c ⟩
  -- jump  : ∀ {c x y} → pos x (suc y) gas High follows pos x y
   ↪ gas Low ⟨ c ⟩
  fall  : ∀ {c cl x y} → pos x y gas cl follows pos x (suc y) gas
   ↪ High ⟨ c ⟩
```

Het `_follows_⟨_⟩` type is op het eerste zicht waarschijnlijk een beetje vreemd, dit is omdat de waarden van het type eigenlijk minder belangrijk zijn dan het type zelf. Het type drukt uit welke positie kan volgen op welke positie rekening houdend met een kleur. Omdat hier de types van de constructors belangrijk zijn, overlopen we ze één voor één. De `stay` constructor drukt uit dat een Koopa Troopa altijd kan blijven staan zolang hij in lucht staat met een lage `Clearance`. De `Material` moet `gas` zijn omdat een Koopa Troopa niet in grond of muren mag staan en de `Clearance` moet `Low` zijn omdat we willen dat ook rode Koopa Troopas kunnen stilstaan. Eigenlijk is stilstaan niet echt een nodige *beweging* maar hiermee kunnen we later de eindpositie van een pad expliciet maken. De volgende twee constructors `next` en `back` bekijken we tegelijk omdat ze heel gelijkaardig zijn. Ze drukken uit dat een positie met een bepaalde `Clearance` enkel kan volgen op een horizontaal vorige, respectievelijk volgende, positie als de kleur van de Koopa Troopa die `Clearance` oplevert. Ook moet de voorgaande positie een lage `Clearance` hebben, dit is omdat we de hoge `Clearance` gebruiken voor posities waar de enige mogelijke beweging

vallen is. Het `Material` voor de posities moet nog steeds `gas` zijn omdat Koopa Troopas niet in muren kunnen lopen. De laatste constructor is `fall` en die kan enkel gebruikt worden als beweging van een positie met een hoge `Clearance`. Aangezien enkel groene Koopa Troopas op een positie met een hoge `Clearance` kunnen geraken, zijn dat ook de enige die kunnen vallen. Nogmaals kunnen we alleen vallen van een positie in lucht naar een positie in lucht, Koopa Troopas kunnen dus niet de grond in vallen.

```
infixr 5 _↠⟨_⟩_
data Path {c : Color} (Koopa : KoopaTroopa c) :
     Position → Position → Set where
  []  : ∀ {p} → Path Koopa p p
  _↠⟨_⟩_ : {q r : Position} → (p : Position) → q follows p ⟨ c ⟩
             → (qs : Path Koopa q r) → Path Koopa p r
```

Het type waarmee we de paden voorstellen die Koopa Troopas kunnen afleggen, `Path`, moet ervoor zorgen dat enkel geldige paden worden opgesteld voor een bepaalde Koopa Troopa. De parameter voor de kleur is impliciet omdat die volgt uit het type van de Koopa Troopa. Verder hangt de geldigheid van een pad af van de Koopa Troopa en kan een pad maar voor één Koopa Troopa tegelijk opgesteld worden, dus is er een parameter waar we een Koopa Troopa verwachten. Een pad gaat van een beginpositie naar een eindpositie en die moeten kunnen variëren voor de verschillende constructors, dus dat zijn indices. De constructor voor een leeg pad, `[]`, moet een begin- en een eindpositie hebben, maar die kunnen we impliciet verkrijgen uit de vorige positie van het pad en als er geen vorige positie is, uit het verwachte type op de plaats waar we de constructor gebruiken. De constructor om langere paden op te stellen maakt weer gebruik van mixfix notatie, de `infixr` declaratie zorgt ervoor dat de constructor rechts associatief is, waardoor we minder haakjes zullen nodig hebben. De constructor verwacht een positie `p`, een pad van `q` naar `r`, `qs`, en een bewijs dat `q` kan volgen op `p` voor de kleur van de Koopa Troopa en resulteert in een pad van `p` naar `r`. `_↠⟨_⟩_` behoudt de geldigheid van een pad door het argument van het type `_follows_⟨_⟩` en de enige manier om een pad te beginnen is met een leeg pad dat sowieso geldig is. Door deze eigenschappen zijn alle paden geldig bij wijze van constructie.

We kunnen nu paden beginnen opstellen maar een belangrijk probleem is nog dat de geldigheid van de paden afhangt van een goede bepaling van de `Clearance` van elke positie. Wat volgt is eerst een klein voorbeeld van een pad en daarna een mogelijke oplossing voor een juiste bepaling van de `Clearance` voor elke positie.

```
ex_path : Path (Red KT) (pos 0 0 gas Low) (pos 0 0 gas Low)
ex_path = pos 0 0 gas Low ↠⟨ next ⟩
          pos 1 0 gas Low ↠⟨ back ⟩
          pos 0 0 gas Low ↠⟨ stay ⟩ []
```

Om ervoor te zorgen dat de `Clearance` juist bepaald wordt, gaan we een functie gebruiken die de regels voor de bepaling van de `Clearance` van een positie uitdrukt.

```
matterToPosVec []           []         _ _ = []
matterToPosVec (mat :: mats) (under :: unders) x y =
  pos x y mat cl :: matterToPosVec mats unders (x + 1) y
    where
      clearance : Material → Material → Clearance
      clearance gas   gas   = High
      clearance gas   solid = Low
      clearance solid _     = Ultimate
      cl = clearance mat under
```

De `matterToPosVec` functie verwacht twee vectoren van dezelfde lengte, met `Materials` in en twee coördinaten die aangeven bij welke positie het eerste `Material` in de eerste vector hoort. Het resultaat is een vector van dezelfde lengte met posities in met de juiste coördinaten, de `Materials` uit de eerste vector en de juiste `Clearance` die afhangt van het `Material` van de positie onder een positie, vandaar dat we twee vectoren in de invoer nodig hebben. Het basisgeval is wanneer de vectoren leeg zijn, in het andere geval bepaalt de functie `clearance` wat de juiste `Clearance` voor de positie is aan de hand van het `Material` van de positie zelf en van de positie er onder. Lucht boven lucht heeft `Clearance High` omdat daar alleen gevallen kan worden, lucht boven grond heeft `Clearance Low` omdat elke Koopa Troopa moet kunnen staan. Alle grond en muur posities hebben `Clearance Ultimate` omdat geen enkele Koopa Troopa zich in een muur of in de grond mag bevinden.

Door vectoren te geven van alle `Materials` op een horizontale rij en de rij daaronder kunnen we nu de `Clearances` juist bepalen. Voor de voorstelling van een level kunnen we dus een vector van zulke vectoren gebruiken. Uiteindelijk willen we een matrix van posities die het level voorstelt maar omdat het voor de recursie beter uitkomt als we met een vector van vectoren werken voor de invoer, gebruiken we geen matrix voor de invoer.

```
matterToPosVecs : {w h : ℕ} → Vec (Vec Material w) h → Vec (Vec
 ↪  Position w) h
matterToPosVecs [] = []
matterToPosVecs (_::_ {y} mats matss) =
  matterToPosVec mats (unders matss gas) 0 y :: matterToPosVecs
   ↪  matss
  where
    unders : ∀ {m n ℓ}{A : Set ℓ} → Vec (Vec A m) n → A → Vec A m
    unders [] fallback = vreplicate fallback
    unders (us :: _) _ = us

matsToMat : {w h : ℕ} → Vec (Vec Material w) h → Matrix Position
 ↪  w h
matsToMat matss = Mat (reverse (matterToPosVecs matss))
```

De functie `matterToPosVecs` maakt gebruik van de vorige functie om een vector van vectoren van `Materials` om te zetten in een vector van vectoren van posities.

Het basisgeval is de lege vector en die wordt omgezet in een lege vector. In het recursieve geval zien we dat de `_::_` constructor voor vectoren gebruikt wordt in de prefix vorm, dit is omdat we willen *pattern matchen* op een derde impliciet argument en drie argumenten gaan niet goed samen met een binaire infix constructor. We gebruiken `matterToPosVec` op de eerste vector in de vector van vectoren en op de vector daaronder, die we verkrijgen uit de functie `unders`. `unders` is eigenlijk heel gelijkaardig aan de `head` functie maar is specifieker voor een vector van vectoren en heeft een *fallback* argument dat we teruggeven als we de rij onder de laatste rij nodig hebben. Het enige dat dan nog rest is om de recursieve oproep te doen op de rest van de vector van vectoren. De `matsToMat` functie maakt van de vector van vectoren van posities, die we terugkrijgen uit `matterToPosVecs`, een matrix waarbij de rijen eerst van volgorde gewisseld worden omdat de index $(0, 0)$ bij een matrix linksboven zit en in de levelvoorstelling links onder.

Met deze functies kunnen we gemakkelijk een level opstellen door de juiste `Materials` in de juiste volgorde op te geven, hierbij wordt dan meteen de juiste `Clearance` bepaalt voor elke positie. Om onze notatie iets visueler te maken definiëren we nog twee functies met een unicode symbool.

```
□_ : ∀{n} → Vec Material n → Vec Material (suc n)
□ xs = gas :: xs
■_ : ∀{n} → Vec Material n → Vec Material (suc n)
■ xs = solid :: xs
infixr 5 □_
infixr 5 ■_
example_level : Matrix Position 10 7
example_level = matsToMat (
  □ □ □ □ □ □ □ □ □ □ [] ::
  □ □ □ □ □ □ ■ ■ ■ □ [] ::
  □ □ □ □ □ □ □ □ □ □ [] ::
  □ □ □ ■ ■ ■ □ □ □ □ [] ::
  □ □ □ □ □ □ □ □ □ □ [] ::
  ■ □ □ □ □ □ □ □ □ ■ [] ::
  ■ ■ ■ ■ ■ □ □ ■ ■ ■ [] :: [])
```

De functies `□_` en `■_` zijn gewoon afkortingen voor respectievelijk een vector die begint met `gas` en nog een staart verwacht en een vector die begint met `solid` en nog een staart verwacht. Deze functies zijn gedefinieerd met een underscore ook al zijn ze gewoon prefix functies omdat de `infixr` declaratie enkel van toepassing is op mixfix functies. Zoals we kunnen zien is het level, dat tevens in figuur 3.2 te zien is, redelijk herkenbaar ook al stellen we het gewoon voor met tekst, dit is een voordeel van werken met de volledige unicode karakterset.

Het enige dat ons nu nog rest te definiëren zijn een aantal hulp functies om natuurlijke getallen om te zetten in getallen van het type `Fin n` omdat we die niet met cijfers kunnen schrijven.

```
_<'_ : ℕ → ℕ → Set
m <' zero = ⊥
zero <' suc n = ⊤
suc m <' suc n = m <' n

fromNat : ∀ {n}(k : ℕ){_ : k <' n} → Fin n
fromNat {zero} k {}
fromNat {suc n} zero = fzero
fromNat {suc n} (suc k) {p} = fsuc (fromNat k {p})

f : ∀ {n}(k : ℕ){_ : k <' n} → Fin n
f = fromNat

p : (x : Fin 10) → (y : Fin 7) → Position
p x y = lookup y x example_level
```

De eerste functie, `_<'_`, vergelijkt twee natuurlijke getallen en geeft een type terug, in een taal waar types geen waarden zijn gaat dit gewoonlijk niet. Als het eerste getal strikt kleiner is dan het tweede is dit type $\top$, anders is het type $\bot$. Een element van het type $\top$ kan altijd impliciet ingevuld worden, er is er namelijk maar één: `tt`. Een element van het type $\bot$ kan nooit gegeven worden want er zijn er geen. Van deze functie maken we gebruik in het type van `fromNat`, een functie die een natuurlijk getal omzet in een element van `Fin n`. We kunnen een natuurlijk getal `k` maar omzetten in een `Fin n` als `k` strikt kleiner is dan `n`. Het type `k <' n` is ofwel gelijk aan het type $\top$ ofwel aan het type $\bot$, in het eerste geval moeten we dus geen expliciete waarde geven, in het tweede geval kunnen we geen waarde geven en dus kunnen we de functie ook niet oproepen met een `k` die we nooit kunnen omzetten in een `Fin n`. Op deze manier kan de typechecker ons waarschuwen wanneer we dit ergens wel proberen te doen. In deze functie zien we nog iets nieuws, de eerste vergelijking heeft geen rechterzijde. Dit is omdat er geen waarde bestaat van het type `Fin 0`, dat isomorf is met het type $\bot$. Omdat de `n` in `Fin n` een natuurlijk getal is, is `0` niet uitgesloten en omdat Agda een totale programmeertaal is moeten we voor elke mogelijke waarde van het argument `n` een antwoord bieden. In dit geval is het type `k <' n` sowieso gelijk aan $\bot$ en omdat we geen waarde van dat type kunnen krijgen kunnen we gebruik maken van het zogenaamde *absurd pattern*: `{}` of `()`. Het *absurd pattern* geeft aan dat er onmogelijk een waarde van het juiste type bestaat, de typechecker moet dit wel kunnen nagaan dus dit werkt alleen voor types die eenvoudig genoeg zijn. Als Agda overtuigd is dat er geen waarde bestaat, moeten we ook geen rechterzijde opgeven. In de derde vergelijking zien we nog dat het *bewijs* dat `k` kleiner is dan `n` expliciet doorgegeven wordt en dus geldig blijft voor het recursieve geval, het is tenslotte ook maar een eenvoudige waarde, `tt`. De functie `f` is slechts een afkorting voor `fromNat`. De functie `p` is een afkorting voor `lookup` waarbij de coördinaten omgewisseld zijn voor de leesbaarheid en het level niet moet opgegeven worden omdat het in alle voorbeelden hetzelfde is.

Met al deze extra definities kunnen we nu paden op een compacte manier definiëren zoals we in de rest van de voorbeelden zien.

```
red_path_one : Path (Red KT) (p (f 7) (f 6)) (p (f 8) (f 6))
red_path_one = p (f 7) (f 6) ↠⟨ back ⟩
               p (f 6) (f 6) ↠⟨ next ⟩
               p (f 7) (f 6) ↠⟨ next ⟩
               p (f 8) (f 6) ↠⟨ stay ⟩ []

red_path_two : Path (Red KT) (p (f 2) (f 1)) (p (f 3) (f 1))
red_path_two = p (f 2) (f 1) ↠⟨ back ⟩
               p (f 1) (f 1) ↠⟨ next ⟩
               p (f 2) (f 1) ↠⟨ next ⟩
               p (f 3) (f 1) ↠⟨ next ⟩
               p (f 4) (f 1) ↠⟨ back ⟩
               p (f 3) (f 1) ↠⟨ stay ⟩ []
```

`red_path_one` en `red_path_two` stellen de paden voor die door de rode Koopa Troopas afgelegd wordt in figuur 3.2, het vakje linksonder heeft coördinaat $(0,0)$, de x-as loopt horizontaal en de y-as verticaal. Voorbeelden van paden waar niets mis mee is, zijn eigenlijk niet zo interessant, die kunnen in het originele spel ook voorgesteld worden. Het wordt pas interessant als we proberen om een ongeldig pad op te stellen, want dat is wat we proberen te voorkomen.

```
-- Type error shows up 'late' because 'cons' is right associative
red_nopath_one : Path (Red KT) (p (f 1) (f 1)) (p (f 0) (f 1))
red_nopath_one = p (f 1) (f 1) ↠⟨ back ⟩
                 p (f 0) (f 1) ↠⟨ stay ⟩ []

-- Red KoopaTroopa can't step into a wall
red_nopath_two : Path (Red KT) (p (f 1) (f 1)) (p (f 0) (f 1))
red_nopath_two = p (f 1) (f 1) ↠⟨ back ⟩ []

-- Red Koopa Troopa can't step into air
red_nopath_three : Path (Red KT) (p (f 4) (f 1)) (p (f 5) (f 1))
red_nopath_three = p (f 4) (f 1) ↠⟨ next ⟩ []
```

De fout bij `red_nopath_one` ziet eruit als volgt:

```
gas != solid of type Material
when checking that the expression stay has type
pos 0 (suc zero) gas Low follows p (f 0) (f 1) ⟨ Red ⟩
```

Agda geeft aan dat de positie in het type van `stay` niet overeenkomt met de eigenlijke positie, door `stay` te gebruiken op positie $(0,1)$ zou het type `pos 0 1 solid Ultimate follows pos 0 1 solid Ultimate ⟨ Red ⟩` moeten zijn maar

dat is niet toegelaten in de definitie van de constructor. Eigenlijk verwachten we een fout, daar waar we een muur in lopen en hier krijgen we de fout pas te zien als we al in een muur staan. Dit is omdat de `_→»⟨_⟩_` constructor rechts associatief is, dus de eerste fout die Agda tegenkomt, zit zo ver mogelijk vanachter in het pad.

In de fout voor `red_nopath_two` kunnen we zien dat de beweging waarbij we een muur zouden inlopen wel als fout wordt gezien:

```
gas != solid of type Material
when checking that the expression p (f 1) (f 1) →» ⟨ back ⟩ [] has
type Path (Red KT) (p (f 1) (f 1)) (p (f 0) (f 1))
```

Deze fout is zo goed als hetzelfde als de vorige, alleen gaat het nu om de `back` constructor en is de expressie waar de fout zich bevindt niet even ver gereduceerd omdat het type van `back` ingewikkelder is.

Een pad voor een rode Koopa Troopa moet ook ongeldig zijn als een Koopa Troopa zou vallen bij het afleggen van het pad:

```
Low != High of type Clearance
when checking that the expression p (f 4) (f 1) →» ⟨ next ⟩ [] has
type Path (Red KT) (p (f 4) (f 1)) (p (f 5) (f 1))
```

Als een rode Koopa Troopa van een afgrond afloopt, is het niet het `Material` dat in de weg zit, het is namelijk allemaal lucht, maar wel het feit dat een rode Koopa Troopa niet de juiste `Clearance` heeft.

Voor een groene Koopa Troopa kunnen we opnieuw geldige paden opstellen.

```
-- Any path that is valid for red Koopa Troopas, is also valid for
 ↪  green
-- Koopa Troopas because we did not constrain Koopa Troopas to only
 ↪  turn
-- when there is an obstacle
green_path_one : Path (Green KT) (p (f 7) (f 6)) (p (f 8) (f 6))
green_path_one = p (f 7) (f 6) →» ⟨ back ⟩
                 p (f 6) (f 6) →» ⟨ next ⟩
                 p (f 7) (f 6) →» ⟨ next ⟩
                 p (f 8) (f 6) →» ⟨ stay ⟩ []

green_path_two : Path (Green KT) (p (f 7) (f 6)) (p (f 5) (f 0))
green_path_two = p (f 7) (f 6) →» ⟨ back ⟩
                 p (f 6) (f 6) →» ⟨ back ⟩
                 p (f 5) (f 6) →» ⟨ fall ⟩
                 p (f 5) (f 5) →» ⟨ fall ⟩
                 p (f 5) (f 4) →» ⟨ back ⟩
                 p (f 4) (f 4) →» ⟨ back ⟩
                 p (f 3) (f 4) →» ⟨ back ⟩
                 p (f 2) (f 4) →» ⟨ fall ⟩
```

```
p (f 2) (f 3) ↠⟨ fall ⟩
p (f 2) (f 2) ↠⟨ fall ⟩
p (f 2) (f 1) ↠⟨ back ⟩
p (f 1) (f 1) ↠⟨ next ⟩
p (f 2) (f 1) ↠⟨ next ⟩
p (f 3) (f 1) ↠⟨ next ⟩
p (f 4) (f 1) ↠⟨ next ⟩
p (f 5) (f 1) ↠⟨ fall ⟩ []
```

In `green_path_one` zien we één van de geldige paden voor rode Koopa Troopas terugkomen. Een pad dat geldig is voor een rode Koopa Troopa is ook geldig voor een groene Koopa Troopa. In het oorspronkelijke spel zijn Koopa Troopas ook obstakels voor elkaar, op die manier zou een groene Koopa Troopa dus beperkt kunnen worden tot een pad dat een rode ook kan afleggen. Evenzeer is een pad voor een rode Koopa Troopa die omkeert voor er een obstakel in de weg zit, mogelijk. Opnieuw kan dit nuttig zijn, bijvoorbeeld om de Koopa Troopas actief de speler te laten achtervolgen als die in de buurt komt, waarbij ze dus vroeger dan nodig omkeren. In `green_path_two` zien we het langere pad dat in figuur 3.2 geïllustreerd is.

Voor een groene Koopa Troopa is het geen fout als het pad van een afgrond af gaat en dan naar beneden valt. Maar er treedt zoals verwacht nog wel een fout op als we proberen een muur in te lopen.

```
-- Green Koopa Troopa can't step into a wall
green_nopath_one : Path (Green KT) (p (f 1) (f 1)) (p (f 0) (f 1))
green_nopath_one = p (f 1) (f 1) ↠⟨ back ⟩ []
```

```
gas != solid of type Material
when checking that the expression p (f 1) (f 1) ↠⟨ back ⟩ [] has
type Path (Green KT) (p (f 1) (f 1)) (p (f 0) (f 1))
```

Deze fout is identiek aan die voor `red_nopath_two`. In het volgende deel bespreken we de implementatie van dit model in Haskell.

3.3 Koopa Troopas in Haskell

In Haskell beginnen we de code met de LANGUAGE pragma's die besproken zijn in hoofdstuk 2. Daarna komt de module declaratie, hier zien we voor de eerste keer dat Haskell redelijk strikte regels heeft over naamgeving, modules, types en constructors moeten met een hoofdletter beginnen en functies met een kleine letter.

```
{-# LANGUAGE GADTs, DataKinds, KindSignatures #-}
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}

module Koopa where
```

Om te beginnen moeten we een aantal types die we in Agda uit de standard library kunnen halen, zelf implementeren, er is bijvoorbeeld geen erg grote nood aan een unaire voorstelling van natuurlijke getallen in Haskell. De naamgeving en technieken die we hier gebruiken komen uit een artikel [21] over dependently typed programmeren in Haskell met de titel: "Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming." Het wordt redelijk snel duidelijk waarom het artikel die titel heeft.

```
data Nat = Z | S Nat
data Natty :: Nat -> * where
  Zy :: Natty Z
  Sy :: Natty n -> Natty (S n)
natter :: Natty n -> Nat
natter Zy = Z
natter (Sy n) = S (natter n)
data NATTY :: * where
  Nat :: Natty n -> NATTY
nattyer :: Nat -> NATTY
nattyer Z = Nat Zy
nattyer (S n) = case nattyer n of Nat m -> Nat (Sy m)
```

We beginnen met het definiëren van onze unaire natuurlijke getallen, dankzij de datatype promotion van de DataKinds extensie, hebben we nu ook meteen unaire natuurlijke getallen op het typeniveau. Wat we nog missen is een verbinding tussen waarden op het waardeniveau en waarden op het typeniveau zoals die er wel is in Agda, daar kunnen we een argument van de functie ook in het type gebruiken. In Haskell kunnen we dit niet echt doen, maar we kunnen het wel imiteren met een zogenaamde singleton constructie. Dit zien we in de definitie van `Natty`, elke waarde die we opstellen heeft een ander type: `Zy` heeft type `Natty Z` en `Sy Zy` heeft type `Natty (S Z)`. Omdat elk type `Natty n` maar één waarde bevat, heeft deze constructie de benaming singleton gekregen. De functie `natter`, ongeveer te lezen als "meer Nat-achtig," zet een `Natty n` om in een `Nat`, hierbij werpen we als het ware informatie op het typeniveau weg. De omgekeerde conversie gaat niet zomaar, er is geen manier om statisch te bepalen wat de `n` in het return type `Natty n` zou moeten zijn. Om dit bij benadering toch te kunnen doen hebben we een derde type nodig, `NATTY`. Dit type verstopt als het ware de typeniveau informatie van een `Natty n`. Nu kunnen we de functie `nattyer`, "meer Natty-achtig," wel schrijven. De definitie is eenvoudig, in het recursieve geval moeten we een *pattern match* doen op de recursieve oproep om met de eigenlijke waarde van de `Natty` te kunnen werken.

Haskell ontbreekt logischerwijs tevens een type voor begrensde natuurlijke getallen. Deze keer kunnen we de extra types niet definiëren omdat Haskell het `Fin` type niet kan promoveren. De DataKinds extensie promoveert enkel types met argumenten met *kind* $*$ en `Fin` heeft een argument met *kind* `Nat`.

```
data Fin :: Nat -> * where
  Zf :: Fin (S n)
```

```
  Sf :: Fin n -> Fin (S n)
intToFin :: Natty n -> Integer -> Fin n
intToFin Zy _ = error "Fin Z is an empty type"
intToFin (Sy n) i
  | i <  0 = error "Negative Integers cannot be represented by a
  ↪  finite natural"
  | i == 0 = Zf
  | i >  0 = Sf (intToFin n (i-1))
```

De definitie voor `Fin` is identiek aan die uit de Agda standard library voor zover dat mogelijk is. De functie `intToFin` dient ongeveer hetzelfde doel als de `fromNat` functie uit de Agda code. Omdat Haskell geen cijfers toelaat voor natuurlijke getallen, vertrekken we van het `Integer` type. Dit brengt met zich mee dat we negatieve getallen toelaten, hiervoor werpen we gewoon een fout op.

Vervolgens is er het type `Vec` dat ook weer logischerwijs niet in Haskell aanwezig is met de bijhorende functies.

```
data Vec :: * -> Nat -> * where
  V0   :: Vec a Z
  (:>) :: a -> Vec a n -> Vec a (S n)
infixr 5 :>

vlookup :: Fin n -> Vec a n -> a
vlookup (Zf)   (a :> _)  = a
vlookup (Sf n) (_ :> as) = vlookup n as

vreplicate :: Natty n -> a -> Vec a n
vreplicate Zy     _ = V0
vreplicate (Sy n) a = a :> vreplicate n a

vreverse :: Vec a n -> Vec a n
vreverse V0 = V0
vreverse (a :> V0) = a :> V0
vreverse (a :> as) = vreverse' as a
  where
    vreverse' :: Vec a n -> a -> Vec a (S n)
    vreverse' V0 x' = x' :> V0
    vreverse' (x :> xs) x' = x :> vreverse' xs x'
```

Haskell laat binaire infix constructors toe door hun naam te beginnen met een dubbelepunt, toevallig kunnen we zo bijna dezelfde notatie gebruiken als in Agda. `vlookup` haalt een element uit een vector met behulp van een begrensd getal dus we sluiten uit dat een index te groot of te klein is. `vreplicate` stelt een vector op met een bepaald aantal herhalingen van een element, we hebben hier een `Natty n` nodig omdat we de waarde `n` moeten kennen voor het type van de vector die we willen teruggeven. `vreverse` keert de volgorde van de elementen in een vector om.

Nu kunnen we ook het `Matrix` type definiëren. Dit type en de voorgaande types zijn niet gedefinieerd in submodules zoals we `Matrix` gedefinieerd hebben in Agda omdat Haskell geen concept heeft van submodules. We zouden deze types allemaal in modules in hun eigen bestanden kunnen definiëren maar omdat het weinig code is en slechts tot voorbeeld dient, is dit achterwege gelaten.

```
data Matrix :: * -> Nat -> Nat -> * where
  Mat :: Vec (Vec a w) h -> Matrix a w h

mlookup :: Fin h -> Fin w -> Matrix a w h -> a
mlookup row column (Mat rows) = vlookup column (vlookup row rows)
```

Zowel het `Matrix` type als de `mlookup` functie gelijken sterk op hun tegenhangers in Agda. Het grootste verschil is dat we in Haskell met polymorfisme werken daar waar we in Agda gewoonlijk impliciete argumenten gebruiken.

Voor het `Color` type hebben we weer het type zelf en een singleton type nodig, `Colorry`, omdat we in het `KoopaTroopa` type de kleur uit het argument nodig hebben op het typeniveau.

```
data Color = Green | Red
data Colorry :: Color -> * where
  Greeny :: Colorry Green
  Redy   :: Colorry Red

data KoopaTroopa :: Color -> * where
  KT :: Colorry c -> KoopaTroopa c
```

Wat we hierbij ook kunnen opmerken is dat het type `KoopaTroopa` eigenlijk hetzelfde is als `Colorry` op de constructors na, en hier zit het probleem: als we een constructor willen die een kleur argument neemt en gebruik maakt van de bijhorende waarde op het typeniveau moet dat argument een singleton type hebben.

Voor de `Material` en `Clearance` types hebben we dezelfde constructies nodig als voor de natuurlijke getallen.

```
data Material = Gas | Solid
data Matty :: Material -> * where
  Gasy   :: Matty Gas
  Solidy :: Matty Solid
data MATTY :: * where
  Mater :: Matty m -> MATTY
mattyer :: Material -> MATTY
mattyer Gas = Mater Gasy
mattyer Solid = Mater Solidy

data Clearance = Low | High | Ultimate
data Clearry :: Clearance -> * where
```

```
  Lowy  :: Clearry Low
  Highy :: Clearry High
  Ultimatey  :: Clearry Ultimate
data CLEARRY :: * where
  Clear :: Clearry cl -> CLEARRY
clearryer :: Clearance -> CLEARRY
clearryer Low = Clear Lowy
clearryer High = Clear Highy
clearryer Ultimate = Clear Ultimatey
```

Het veelvoud aan types dat we op deze manier krijgen is conceptueel geen probleem maar praktisch is het wel lastig. Soms hebben we een waarde van het ene type en hebben we één van de andere types nodig dus moeten we een functie gebruiken om van een `Matty` een `Material` te maken, jammer genoeg gaat deze omzetting niet altijd in de omgekeerde richting. Om de omzetting van `Material` naar `Matty` toch te doen hebben we het type `MATTY` maar bij deze omzetting verliezen we de informatie op het typeniveau waarvoor we `Matty` net gemaakt hebben. Dit is niet altijd een probleem maar komt wel voortdurend terug als we met deze constructies werken.

Omdat we in het type voor een pad gebruik willen maken van de waarden van de posities, hebben we voor posities weer dezelfde constructie nodig, alleen is deze nu uitgebreider.

```
data Position = Pos { getX      :: Nat
                    , getY      :: Nat
                    , matter :: Material
                    , clr    :: Clearance
                    }
data Positionny :: Position -> * where
  Posy :: Natty x -> Natty y -> Matty m -> Clearry cl
            -> Positionny (Pos x y m cl)
data POSITIONNY :: * where
  Posit :: Positionny p -> POSITIONNY
positionnyer :: Position -> POSITIONNY
positionnyer (Pos x y m cl)
  | Nat xY <- nattyer x
  , Nat yY <- nattyer y
  , Mater mY <- mattyer m
  , Clear clY <- clearryer cl
  = Posit (Posy xY yY mY clY)
```

In de functie `positionnyer` zien we een geval waar we de waarden uit de types `NATTY`, `MATTY` en `CLEARRY` als het ware kunnen uitpakken, dit is alleen mogelijk omdat we ze daarna terug inpakken in een waarde met type `POSITIONNY`.

Het volgende concept dat we moeten uitdrukken is de relatie tussen een kleur en de bijhorende `Clearance`, in Agda gebruiken we hiervoor een type en *instance*

arguments. Omdat we niet telkens expliciet een waarde willen meegeven die toch niet gebruikt wordt, wat we gebruiken is immers het type, gebruiken we een typeclass in Haskell.

```
class CoClr (c :: Color) (cl :: Clearance)
instance CoClr c Low
instance CoClr Green High
```

Op deze manier kunnen we met een typeclass constraint aan de informatie geraken die we nodig hebben, zonder een expliciet argument.

Voor het `Follows` type is de implementatie redelijk eenvoudig. Buiten de typeclass constraint in plaats van een *instance argument* en het polymorfisme in plaats van impliciete argumenten, is het grootste verschil de syntax. Haskell laat binaire infix constructors toe en met de extensie TypeOperators [22] ook types en typeclasses maar een ternaire mixfix notatie is er niet. We gebruiken dus gewoon een type met drie parameters.

```
data Follows :: Position -> Position -> Color -> * where
  Stay :: Follows (Pos x y Gas Low) (Pos x y Gas Low) c
  Next :: CoClr c cl => Follows (Pos (S x) y Gas cl) (Pos x y Gas
   ↪  Low) c
  Back :: CoClr c cl => Follows (Pos x y Gas cl) (Pos (S x) y Gas
   ↪  Low) c
  Fall :: Follows (Pos x y Gas cl) (Pos x (S y) Gas High) c
```

Voor het type `Path` komen we een probleem tegen.

```
data Path :: Color -> Position -> Position -> * where
  P0 :: Path c p p
  Pcons :: Positionny p -> Follows q p c -> Path c q r -> Path c p r
```

`Path` kan geen parameter hebben met *kind* `KoopaTroopa` omdat het type `KoopaTroopa` een parameter heeft met een *kind*, namelijk `Color`, dat niet `*` is. Zo'n type kan door de DataKinds extensie niet gepromoveerd worden dus kunnen we het ook niet als *kind* gebruiken.

Op dit punt kunnen we hetzelfde pad voorstellen als in de Agda code.

```
exPath :: Path Red (Pos Z Z Gas Low) (Pos Z Z Gas Low)
exPath = Pcons (Posy Zy Zy Gasy Lowy) Next
        (Pcons (Posy (Sy Zy) Zy Gasy Lowy) Back
        (Pcons (Posy Zy Zy Gasy Lowy) Stay P0))
```

`exPath` is heel gelijkaardig aan `ex_path` uit Agda maar de unaire voorstelling die we moeten gebruiken voor de getallen doet wel afbreuk aan de leesbaarheid.

De functies om een `Matrix` op te stellen aan de hand van een vector van vectoren van `Material`s zijn redelijk eenvoudig. We hebben wel expliciete argumenten nodig

voor de breedte en hoogte maar eigenlijk om een andere reden dan bij vorige gebruiken van de singleton types. In dit geval hebben we de informatie al op het typeniveau en hebben we ze in `matterToPosVecs` nodig op het niveau van waarden. De enige manier om dit te bereiken is met expliciete argumenten van een singleton type.

```
matterToPosVec :: Vec Material n -> Vec Material n -> Nat -> Nat
                  -> Vec Position n
matterToPosVec V0 V0 _ _ = V0
matterToPosVec (mat :> mats) (under :> unders) x y =
  Pos x y mat cl :> matterToPosVec mats unders (S x) y
    where
      clearance :: Material -> Material -> Clearance
      clearance Gas   Gas   = High
      clearance Gas   Solid = Low
      clearance Solid _     = Ultimate
      cl = clearance mat under

matterToPosVecs :: Natty w -> Natty h -> Vec (Vec Material w) h
                   -> Vec (Vec Position w) h
matterToPosVecs _ _ V0 = V0
matterToPosVecs w (Sy z) (mats :> matss) =
  matterToPosVec mats (unders w matss Gas) Z y  :> matterToPosVecs w
  ↪ z matss
    where
      y = natter (Sy z)
      unders :: Natty m -> Vec (Vec a m) n -> a -> Vec a m
      unders m V0 fallback = vreplicate m fallback
      unders _ (us :> _) _ = us

mattersToMatrix :: Natty w -> Natty h -> Vec (Vec Material w) h
                   -> Matrix Position w h
mattersToMatrix w h matss = Mat (vreverse (matterToPosVecs w h
 ↪ matss))
```

Twee functies uit Agda die we nog missen in Haskell zijn de functies die een `Material` voorstellen met een symbool. In Haskell kunnen we deze wel definiëren zoals in Agda, mits we geldige karakters gebruiken, namen van functies moeten uit alfanumerieke karakters bestaan en niet alle unicode karakters zijn alfanumeriek. Maar omdat we geen fixity declaratie kunnen geven voor een functie en geen unaire operatoren kunnen definiëren, is er geen manier om deze functies rechts associatief te maken. Als we de constructor `(:>)` dus mee zouden opnemen in de functies zouden we overal `$` moeten gebruiken waar nu de constructor staat.

```
o :: Material
o = Gas
```

```
c :: Material
c = Solid
```

Het voorbeeld level ziet er hetzelfde uit ook al is dit minder duidelijk door de symbolen die we gebruiken.

```
exampleLevel :: Matrix Position
                      (S(S(S(S(S(S(S(S(S(S Z)))))))))) -- 10
                      (S(S(S(S(S(S(S Z)))))))) -- 7
exampleLevel = mattersToMatrix
                 (Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy))))))))) -- 10
                 (Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy)))))))) -- 7
                 (
  (o :> o :> o :> o :> o :> o :> o :> o :> o :> V0) :>
  (o :> o :> o :> o :> o :> c :> c :> c :> o :> V0) :>
  (o :> o :> o :> o :> o :> o :> o :> o :> o :> V0) :>
  (o :> o :> o :> c :> c :> c :> o :> o :> o :> o :> V0) :>
  (o :> o :> o :> o :> o :> o :> o :> o :> o :> o :> V0) :>
  (c :> o :> o :> o :> o :> o :> o :> o :> o :> c :> V0) :>
  (c :> c :> c :> c :> c :> o :> o :> c :> c :> c :> V0) :> V0)
```

De laatste functies die we nog niet gedefinieerd hebben zijn de functies om onze notatie van de voorbeelden te verkorten.

```
ix :: Integer -> Fin (S(S(S(S(S(S(S(S(S Z)))))))))) -- 10
ix = intToFin (Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy))))))))) -- 10

iy :: Integer -> Fin (S(S(S(S(S(S(S Z)))))))) -- 7
iy = intToFin (Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy)))))))) -- 7

p :: Fin (S(S(S(S(S(S(S(S(S Z)))))))))
     -> Fin (S(S(S(S(S(S(S Z)))))))
     -> POSITIONNY
p x y | Pos x' y' m' cl' <- mlookup y x exampleLevel
      , Nat xy <- nattyer x'
      , Nat yy <- nattyer y'
      , Mater my <- mattyer m'
      , Clear cly <- clearryer cl'
      = Posit (Posy xy yy my cly)
```

Omdat we de bovengrens niet impliciet kunnen laten voor `intToFin`, definiëren we twee functies met de juiste grenzen voor horizontale en verticale coördinaten, `ix` en `iy`. De `p` functie uit Agda zorgt voor een probleem in Haskell. Deze functie zou namelijk een `Positionny p` moeten teruggeven maar in de `Matrix` die het level voorstelt zitten enkel `Position`s. We weten dat de omzetting van `Position` naar `Positionny` niet zomaar kan en dat we dus een `POSITIONNY` moeten teruggeven.

Het probleem hiermee duikt pas op in de voorbeelden: we kunnen een `POSITIONNY` uitpakken om te werken met de `Positionny` die erin zit zolang we die `Positionny` ofwel verbruiken in een functie, ofwel terug verpakken in een type zoals `POSITIONNY`. In `p` kunnen we geen van beide doen, het `Path` heeft de `Positionny` nodig dus we kunnen die niet omzetten in iets anders en het type `Path` heeft ook de informatie uit het type van de `Positionny` nodig en daar kunnen we niet aan.

Dit probleem is wel op te lossen maar de oplossing is niet eenvoudig. Het probleem is eigenlijk dat we het type van een element niet uit de `Matrix` kunnen halen zoals in Agda. Om dit in Haskell te kunnen doen zou het type `Matrix` een overeenkomstig singleton type moeten hebben. Maar `Matrix` heeft parameters met *kind* `Nat` en kan dus niet gepromoveerd worden. Dit kunnen we oplossen door natuurlijke getallen te definiëren met *kind* $*$: een leeg type `Z` en een leeg type `S n` bijvoorbeeld. Maar we komen hetzelfde probleem tegen in het type van de constructor `Mat`, het vector type daar is ook niet te promoveren omdat `Vec` weer een parameter van *kind* `Nat` heeft. Dit kunnen we oplossen door het *kind* `Nat` in de definitie van `Vec` te vervangen door het *kind* $*$. Op deze manier kan de DataKinds extensie het `Matrix` type promoveren en kunnen we er een singleton voor definiëren. Dit is nog niet de hele oplossing en ze is ook niet geïmplementeerd omdat we hierbij gedeeltelijk garanties verliezen. Plotseling is `S Bool` een geldig type en `Vec Z Char` ook. We zijn dus als het ware *unkinded* aan het programmeren, analoog aan untyped.

Om dit probleem te omzeilen definiëren we een aantal `Positionny`s alsof we ze uit de `Matrix` voor het voorbeeld level hebben gehaald.

```
p01 = Posy Zy (Sy Zy) Solidy Ultimatey
p11 = Posy (Sy Zy) (Sy Zy) Gasy Lowy
p21 = Posy (Sy(Sy Zy)) (Sy Zy) Gasy Lowy
p22 = Posy (Sy(Sy Zy)) (Sy(Sy Zy)) Gasy Highy
p23 = Posy (Sy(Sy Zy)) (Sy(Sy(Sy Zy))) Gasy Highy
p24 = Posy (Sy(Sy Zy)) (Sy(Sy(Sy(Sy Zy)))) Gasy Highy
p31 = Posy (Sy(Sy(Sy Zy))) (Sy Zy) Gasy Lowy
p34 = Posy (Sy(Sy(Sy Zy))) (Sy(Sy(Sy(Sy Zy)))) Gasy Lowy
p41 = Posy (Sy(Sy(Sy(Sy Zy)))) (Sy Zy) Gasy Lowy
p44 = Posy (Sy(Sy(Sy(Sy Zy)))) (Sy(Sy(Sy(Sy Zy)))) Gasy Lowy
p51 = Posy (Sy(Sy(Sy(Sy(Sy Zy))))) (Sy Zy) Gasy Highy
p54 = Posy (Sy(Sy(Sy(Sy(Sy Zy))))) (Sy(Sy(Sy(Sy Zy)))) Gasy Lowy
p55 = Posy (Sy(Sy(Sy(Sy(Sy Zy))))) (Sy(Sy(Sy(Sy(Sy Zy))))) Gasy Highy
p56 = Posy (Sy(Sy(Sy(Sy(Sy Zy))))) (Sy(Sy(Sy(Sy(Sy(Sy Zy)))))) Gasy
 ↪  Highy
p66 = Posy (Sy(Sy(Sy(Sy(Sy(Sy Zy))))))
           (Sy(Sy(Sy(Sy(Sy(Sy Zy)))))) Gasy Lowy
p76 = Posy (Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy)))))))
           (Sy(Sy(Sy(Sy(Sy(Sy Zy)))))) Gasy Lowy
p86 = Posy (Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy))))))))
           (Sy(Sy(Sy(Sy(Sy(Sy Zy)))))) Gasy Lowy
```

Met deze posities kunnen we alle voorbeelden die we hadden uitdrukken in Haskell.

```
redPathOne :: Path Red (Pos (S(S(S(S(S(S Z)))))))
                            (S(S(S(S(S(S Z))))))  Gas Low)
                       (Pos (S(S(S(S(S(S(S Z))))))))
                            (S(S(S(S(S(S Z))))))  Gas Low)
redPathOne = Pcons p76 Back
           $ Pcons p66 Next
           $ Pcons p76 Next
           $ Pcons p86 Stay P0

redPathTwo :: Path Red (Pos (S(S Z))     (S Z) Gas Low)
                       (Pos (S(S(S Z))) (S Z) Gas Low)
redPathTwo = Pcons p21 Back
           $ Pcons p11 Next
           $ Pcons p21 Next
           $ Pcons p31 Next
           $ Pcons p41 Back
           $ Pcons p31 Stay P0
```

De paden komen overeen met die in de Agda code. Net zoals in Agda is het geen verassing dat we geldige paden kunnen uitdrukken. Het is interessanter om te kijken naar de fout die we krijgen bij de ongeldige paden.

```
-- Because Stay only allows a position with material Gas to follow
 ↪  from
-- a position with material Gas, this is a type error
redNoPathOne :: Path Red (Pos (S Z) (S Z) Gas Low)
                         (Pos Z (S Z) Solid Ultimate)
redNoPathOne = Pcons p11 Back
             $ Pcons p01 Stay P0

-- Red Koopa Troopa can't step into a wall
-- This is for the same reason as in redNoPathOne but with the Back
-- constructor
redNoPathTwo :: Path Red (Pos (S Z) (S Z) Gas Low)
                         (Pos Z (S Z) Solid Ultimate)
redNoPathTwo = Pcons p11 Back P0

-- Red Koopa Troopa can't step into air
-- Here the problem is that there is no instance for CoClr Red High
 ↪  which
-- is necessary for Next, however the solution GHC suggests is to add
 ↪  it
```

```
-- while it was actually not defined on purpose
redNoPathThree :: Path Red (Pos (S(S(S(S Z)))) (S Z) Gas Low)
                           (Pos (S(S(S(S(S Z))))) (S Z) Gas High)
redNoPathThree = Pcons p41 Next P0
```

De fout voor `redNoPathOne` is een stuk langer dan die voor `red_nopath_one`, dit heeft onder andere te maken met het ontwerp van Agda. Agda is eigenlijk gemaakt om interactief te werken met de typechecker terwijl Haskell eerder los van het schrijven wordt gecompileerd. Agda stopt bij de eerste fout die de typechecker tegenkomt terwijl Haskell doorgaat en probeert alle fouten te vinden.

```
koopa.hs:246:16:
    Couldn't match type 'Gas with 'Solid
    Expected type: Path
                     'Red
                     ('Pos ('S 'Z) ('S 'Z) 'Gas 'Low)
                     ('Pos 'Z ('S 'Z) 'Solid 'Ultimate)
      Actual type: Path
                     'Red ('Pos ('S 'Z) ('S 'Z) 'Gas 'Low)
                          ('Pos 'Z ('S 'Z) 'Gas 'Low)
    In the expression: Pcons p11 Back $ Pcons p01 Stay P0
    In an equation for `redNoPathOne':
        redNoPathOne = Pcons p11 Back $ Pcons p01 Stay P0

koopa.hs:247:22:
    Couldn't match type 'Solid with 'Gas
    Expected type: Positionny ('Pos 'Z ('S 'Z) 'Gas 'Ultimate)
      Actual type: Positionny ('Pos 'Z ('S 'Z) 'Solid 'Ultimate)
    In the first argument of `Pcons', namely `p01'
    In the second argument of `($)', namely `Pcons p01 Stay P0'
    In the expression: Pcons p11 Back $ Pcons p01 Stay P0

koopa.hs:247:26:
    Couldj't match type 'Low with 'Ultimate
    Expected type: Follows
                     ('Pos 'Z ('S 'Z) 'Gas 'Low)
                     ('Pos 'Z ('S 'Z) 'Gas 'Ultimate) 'Red
      Actual type: Follows
                     ('Pos 'Z ('S 'Z) 'Gas 'Low)
                     ('Pos 'Z ('S 'Z) 'Gas 'Low) 'Red
    In the second argument of `Pcons', namely `Stay'
    In the second argument of `($)', namely `Pcons p01 Stay P0'
    In the expression: Pcons p11 Back $ Pcons p01 Stay P0
```

De eerste fout gaat over het pad als een geheel en zegt dat het eigenlijke `Material`, `Solid`, niet overeenkomt met wat het moet zijn, `Gas`. De tweede fout zegt eigenlijk hetzelfde maar dan over de specifieke positie, Haskell leidt wel af dat de `Material Gas` moet zijn maar niet dat de `Clearance Ultimate` niet zou mogen voorkomen. De derde fout gaat wel over de `Clearance` voor een rode Koopa Troopa. Deze drie zijn

wel heel volledig maar we moeten goed opletten dat die fouten niet allemaal dezelfde oorzaak hebben, zoals hier. Merken we ook nog op dat de eerste fout alleen verschijnt als we de type signature voor `redNoPathOne` geven. We moeten dus goed oppassen met inferentie omdat we op die manier wel eens iets anders kunnen schrijven dan we bedoelen.

De fout voor `redNoPathTwo` is veel korter.

```
koopa.hs:254:31:
    Couldn't match type 'Gas with 'Solid
    Expected type: Path
                     'Red
                     ('Pos 'Z ('S 'Z) 'Gas 'Ultimate)
                     ('Pos 'Z ('S 'Z) 'Solid 'Ultimate)
      Actual type: Path
                     'Red
                     ('Pos 'Z ('S 'Z) 'Gas 'Ultimate)
                     ('Pos 'Z ('S 'Z) 'Gas 'Ultimate)
    In the third argument of 'Pcons', namely 'P0'
    In the expression: Pcons p11 Back P0
    In an equation for 'redNoPathTwo': redNoPathTwo = Pcons p11 Back P0
```

Opnieuw komt het `Material` dat we verwachten in de type signature niet overeen met het type dat door `Pcons` wordt afgedwongen. Als we de type signature niet opgeven krijgen we ook weer een andere fout.

De fout voor `redNoPathThree` is nog korter en is ook anders dan de vorige fouten die allemaal ongeveer hetzelfde waren.

```
koopa.hs:262:28:
    No instance for (CoClr 'Red 'High) arising from a use of 'Next'
    Possible fix: add an instance declaration for (CoClr 'Red 'High)
    In the second argument of 'Pcons', namely 'Next'
    In the expression: Pcons p41 Next P0
    In an equation for 'redNoPathThree':
        redNoPathThree = Pcons p41 Next P0
```

Deze keer heeft de fout niets met `Material`s te maken maar wel met de `Clearance`, zoals verwacht. De rode Koopa Troopa heeft geen `Clearance High` dus de juiste *instance* kan niet gevonden worden. De oplossing die Haskell voorstelt is wel fout, het was een bewuste keuze om die *instance* niet te definiëren voor rode Koopa Troopas. Om dit soort verwarrend advies te vermijden zouden gesloten typeclasses nuttig zijn.

We kunnen ook geldige paden voor groene Koopa Troopas opgeven met dezelfde opmerkingen als bij de code in Agda: elk pad dat geldig is voor een rode Koopa Troopa is ook geldig voor groene Koopa Troopas.

```
-- Any path that is valid for red Koopa Troopas, is also valid for
 ↪  green
-- Koopa Troopas because we did not constrain Koopa Troopas to only
 ↪  turn
```

```
-- when there is an obstacle
greenPathOne :: Path Green (Pos (S(S(S(S(S(S(S Z)))))))
                                (S(S(S(S(S(S Z))))))) Gas Low)
                           (Pos (S(S(S(S(S(S(S(S Z))))))))
                                (S(S(S(S(S(S Z))))))) Gas Low)
greenPathOne = Pcons p76 Back
             $ Pcons p66 Next
             $ Pcons p76 Next
             $ Pcons p86 Stay P0

greenPathTwo :: Path Green (Pos (S(S(S(S(S(S(S Z)))))))
                                (S(S(S(S(S(S Z))))))) Gas Low)
                           (Pos (S(S(S(S(S Z))))) Z Gas Low)
greenPathTwo = Pcons p76 Back
             $ Pcons p66 Back
             $ Pcons p56 Fall
             $ Pcons p55 Fall
             $ Pcons p54 Back
             $ Pcons p44 Back
             $ Pcons p34 Back
             $ Pcons p24 Fall
             $ Pcons p23 Fall
             $ Pcons p22 Fall
             $ Pcons p21 Back
             $ Pcons p11 Next
             $ Pcons p21 Next
             $ Pcons p31 Next
             $ Pcons p41 Next
             $ Pcons p51 Fall P0
```

Ten slotte is er nog het foute pad voor een groene Koopa Troopa.

```
-- Green Koopa Troopa can't step into a wall
-- Exactly the same as for redNoPathTwo
greenNoPathOne :: Path Green (Pos (S Z) (S Z) Gas Low)
                             (Pos Z (S Z) Solid Ultimate)
greenNoPathOne = Pcons p11 Back P0
```

Met dezelfde fout als voor `redNoPathTwo`.

```
koopa.hs:300:33:
    Couldn't match type 'Gas with 'Solid
    Expected type: Path
                     'Green
                     ('Pos 'Z ('S 'Z) 'Gas 'Ultimate)
                     ('Pos 'Z ('S 'Z) 'Solid 'Ultimate)
```

```
  Actual type: Path
                 'Green
                 ('Pos 'Z ('S 'Z) 'Gas 'Ultimate)
                 ('Pos 'Z ('S 'Z) 'Gas 'Ultimate)
In the third argument of `Pcons', namely `P0'
In the expression: Pcons p11 Back P0
In an equation for `greenNoPathOne':
    greenNoPathOne = Pcons p11 Back P0
```

3.4 Besluit

We hebben een voorbeeld gezien van wat het is om met dependent types te programmeren. Voor de garanties die we krijgen van de statische verificatie betalen we vooral in de omvang van de code. Voor een spel zoals Mario is dit gewoonlijk niet de moeite maar er zijn genoeg voorbeelden waar statische verificatie wel die extra moeite waard is. We kunnen bijvoorbeeld denken aan applicaties in de ruimtevaart of productie van geïntegreerde circuits waar een fout in de programmatuur ontzettend duur kan zijn. Of een veiligheidssysteem waar levens van afhangen. Enkel voor zulke systemen is het extra werk voor een volledige statische verificatie aanvaardbaar. Maar dependent types laten toe om maar gedeeltelijke verificatie te doen, bijvoorbeeld werken met vectoren, in plaats van lijsten en begrensde getallen als indices, voorkomt elke vorm van indexeringsfouten.

We hebben ook een duidelijk verschil gezien tussen programmeren in een *volwaardige* dependently typed taal en een eerder praktische taal met een typesysteem dat aan expressiviteit aan het winnen is. Ik denk dat de titel van het artikel over dependently typed programmeren in Haskell ondertussen wel duidelijk is. Haskell zit al redelijk dicht bij dependent types maar verschilt nog genoeg om het heel moeilijk te maken om op deze manier te werken. In Haskell kan het extra werk nog steeds aanvaardbaar zijn afhankelijk van de kost van fouten maar is het waarschijnlijk eerder voordelig om te werken met gedeeltelijke verificatie. Wat hier wel tegenover staat is dat Haskell met deze technieken in de praktijk bruikbaar is, daar waar Agda nog een lange weg te gaan heeft voordat dit het geval is. Ook is een ontwikkeling als het singleton package [23] het vermelden waard, dit zou gedeeltelijk voor een vereenvoudiging kunnen zorgen wat de veelheid aan types betreft die we momenteel nodig hebben.

De code voor dit hoofdstuk is in zijn geheel terug te vinden in bijlagen A en B. In het volgende hoofdstuk bekijken we een tweede gevalstudie waar we de tekortkomingen van Haskell tegenover volwaardige dependent types op een andere manier oplossen.

Hoofdstuk 4

Case: Verified Red-Black Trees

4.1 Inleiding

Deze tweede gevalstudie gaat over een implementatie van red-black trees [24]. Red-black trees zijn een soort binaire zoekbomen, door één extra bit aan informatie bij te houden voor elke knoop zijn ze bij benadering gebalanceerd. Deze benaderende balans is goed genoeg om $\mathcal{O}(\log n)$ tijdscomplexiteit te garanderen voor: het opzoeken van een element, een element toe te voegen en een element te verwijderen. Zoekbomen worden vaak gebruikt om meer abstracte datastructuren te implementeren, bijvoorbeeld verzamelingen en associatieve arrays. Vooral voor de geordende varianten van die datastructuren kunnen zoekbomen voordeliger zijn voor de implementatie dan hashtabellen omdat zoekbomen per definitie geordend zijn.

Chris Okasaki heeft een elegante implementatie van red-black trees [25] in een functionele programmeertaal geschreven, in Haskell. De implementaties in dit hoofdstuk zijn hierop gebaseerd. Zijn implementatie is gebaseerd op een bondige formulering van de red-black tree invarianten. Deze gaan als volgt:

Invariant 1 Geen enkele rode knoop heeft een rode ouder.

Invariant 2 Elk pad van de wortel naar een blad bevat hetzelfde aantal zwarte knopen.

De implementatie van Okasaki wordt hier herhaald zodat het eenvoudig is om te zien wat van het oorspronkelijk algoritme komt en wat toegevoegd is om de twee invarianten statisch te garanderen.

```
module RedBlackTree where

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

type Set a = Tree a
```

```
empty :: Set elt
empty = E

member :: Ord elt => elt -> Set elt -> Bool
member x E = False
member x (T _ a y b) | x <  y = member x a
                     | x == y = True
                     | x >  y = member x b

-- originally without type signature
balance :: Color -> Tree elt -> elt
           -> Tree elt -> Tree elt
balance B (T R (T R a x b) y c) z d =
  T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d =
  T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) =
  T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) =
  T R (T B a x b) y (T B c z d)
balance color a x b = T color a x b

insert :: Ord elt => elt -> Set elt
          -> Set elt
insert x s = makeBlack (ins s)
  where
    ins E = T R E x E
    ins (T color a y b)
      | x <  y = balance color (ins a) y b
      | x == y = T color a y b
      | x >  y = balance color a y (ins b)

    makeBlack (T _ a y b) = T B a y b
```

Omdat deze implementatie zo kort en eenvoudig is kan bijna iedereen deze code waarschijnlijk begrijpen. Maar omdat datastructuren zo fundamenteel zijn aan alles wat programma's doen, is het belangrijk dat ze correct zijn. Eén datastructuur kan nog wel correct gehouden worden door goed na te denken over aanpassingen en een mentaal model van de werking bij te houden maar vanaf dat er een tiental datastructuren zijn, is er bijna niemand die ervoor kan zorgen dat die correct zijn en blijven. Gewoonlijk wordt er daarom een suite van tests opgesteld die regelmatig uitgevoerd dienen te worden, iedere keer dat er een fout gevonden wordt, worden er tests toegevoegd die die fout zouden detecteren als ze ooit terugkomt. Dit is een bewezen methode maar omvat onmiskenbaar een hoop extra werk: tests

moeten geschreven worden en onderhouden omdat bijvoorbeeld de interface tot het programma kan veranderen. Een zwakheid van deze aanpak is dat de tests vaak niet alle mogelijke gevallen afdekken: we moeten vertrouwen op ons vermogen om te redeneren over de code om speciale gevallen te ontdekken waar een grote kans voor bestaat dat er iets mis gaat. Om dit op te lossen zijn er gerandomiseerde testprogramma's zoals QuickCheck [26] bedacht, die het vinden van speciale gevallen automatiseren. In de implementaties in dit hoofdstuk daarentegen maken we gebruik van de expressiviteit van dependent types om de invarianten door het typesysteem te laten nakijken en zo de code statisch te verifiëren.

4.2 Red-Black Trees in Agda

De code van Okasaki is heel elegant maar ze bevat heel weinig statische garanties. Het `Tree` type bijvoorbeeld voorkomt helemaal geen misbruik: `T R (T R E 1 E) 2 E` is een geldige boom, net zoals `T B (T B E 1 E) 2 E` terwijl beiden niet voldoen aan de invarianten van red-black trees. Dit betekent dat we goed moeten opletten als we een functie schrijven die een `Tree` moet teruggeven omdat we geen waarschuwing krijgen als we een ongeldige boom proberen op te stellen. Om nog maar niet te vermelden hoe we met ongeldige bomen als invoer moeten omspringen? De gemakkelijkste oplossing is om er niets aan te doen, maar dan kan het programma crashen omdat we ergens in een functie impliciet aannamen dat de invoer een geldige red-black tree zou zijn. Wat mogelijk nog erger is dan een crash is geen crash, een ongeldige boom gaat door het programma en elke functie waarmee die in aanraking komt kan er iets aan verandert hebben, op het einde merken we dat de uitvoer niet correct is en nu moeten we uitzoeken waar het precies mis gegaan is. De expressiviteit van dependent types kan ons hierbij helpen, als we ervoor zorgen dat een ongeldige boom niet opgesteld kan worden, kunnen we ook geen problemen hebben met ongeldige bomen. Het `Tree` type in Agda is eigenlijk een exacte definitie van wat een red-black tree is terwijl het `Tree` type van Okasaki eender welke binaire boom met rood en zwart gekleurde knopen voorstelt.

```
module RedBlackTree where

open import Data.Bool using (Bool; true; false) renaming (T to So;
 ↪  not to ¬)
_⇒_ : ∀{ℓ₁ ℓ₂} → Set ℓ₁ → Set ℓ₂ → Set _
P ⇒ T = {{p : P}} → T
infixr 3 _⇒_

if_then_else_ : ∀{ℓ}{A : Set ℓ} b → (So b ⇒ A) → (So (¬ b) ⇒ A) →
 ↪  A
if true  then t else f = t
if false then t else f = f
```

```
open import Data.Nat hiding (_<_; _≤_; _≟_; compare)
  renaming (decTotalOrder to ℕ-DTO)
open import Level hiding (suc)
open import Relation.Binary hiding (_⇒_)

open import Data.Sum using (_⊎_) renaming (inj₁ to h+0; inj₂ to h+1)
open import Data.Product using (Σ; Σ-syntax; _,_; proj₁; proj₂)
```

We beginnen met een aantal imports en de definities van implicatie en mixfix *if then else*, alles dat we hier zien wordt uitgelegd daar waar het voor het eerst gebruikt wordt.

```
module RBTree {a ℓ}(order : StrictTotalOrder a ℓ ℓ) where

  open module sto = StrictTotalOrder order
  A = Carrier

  pattern LT = tri< _ _ _
  pattern EQ = tri≈ _ _ _
  pattern GT = tri> _ _ _
  _≤_ = compare
```

We definiëren een geparametriseerde module, `RBTree`, met als parameter een *record* van het type `StrictTotalOrder`. De `a` en ℓ parameters zijn levels en zorgen ervoor dat de elementen van onze boom ook elementen van een type in één van de grotere universa kunnen zijn: $\mathtt{Set}_1$, $\mathtt{Set}_2$, etc. Door een waarde van het type `StrictTotalOrder` te eisen, kunnen we er zeker van zijn dat we de elementen kunnen ordenen en de zoekbomen dus nuttig kunnen invullen. De twee volgende regels zorgen ervoor dat we in de rest van de code in de module kunnen verwijzen naar dit type met een strikte totale orderelatie als `A`. De *pattern* declaraties zorgen ervoor dat we met een compactere notatie kunnen werken voor het resultaat van een vergelijking. Op deze manier verbergen we een aantal bewijzen die we niet echt nodig hebben. Ten slotte kiezen we een symbool om de vergelijkingsfunctie te verkorten. Het symbool staat voor de relatie "kleiner dan of gelijk aan" maar het resultaat van de functie is `LT` als het eerste argument voor het tweede komt, `EQ` als de elementen gelijk zijn en `GT` als het eerste element na het tweede komt.

Nu kunnen we net als Okasaki beginnen met definiëren wat een kleur is en wat een red-black tree is.

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

  _=ᶜ_ : Color → Color → Bool
  R =ᶜ R = true
  B =ᶜ B = true
```

```
_ =ᶜ _ = false

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)
```

Het type `Color` is heel eenvoudig, de functie `_=ᶜ_` is niet meer dan een vergelijking van twee kleuren op gelijkheid met een Booleaans resultaat. We definiëren ook nog een synoniem voor natuurlijke getallen, `Height`, wat het aantal zwarte knopen op een pad van de wortel naar een willekeurig blad voorstelt. Het type `Tree` is ook nog redelijk eenvoudig maar dwingt wel meteen de twee invarianten af. De constructor voor een rode knoop, `R`, laat enkel kinderen toe met een zwarte wortel, op die manier kan een rode knoop nooit een rode ouder hebben. Dit zorgt dat invariant 1 behouden blijft. Overigens moeten de twee kinderen dezelfde hoogte hebben om invariant 2 te behouden. De constructor, `B`, eist eveneens dat de kinderen een gelijke hoogte hebben zodat invariant 2 behouden blijft. Meer is er niet nodig om een type op te stellen dat enkel correcte red-black trees kan voorstellen.

Het grootste voordeel van dit preciezere type is tegelijk ook een nadeel: algoritmes verbreken vaak tijdelijk een of meerdere invarianten omdat het soms eenvoudiger is om iets kapot te maken en daarna in stappen te herstellen dan om alles in één keer goed te doen. Dit is vooral het geval wanneer een verandering een andere verandering tot gevolg kan hebben met een domino-effect. Okasaki maakt gebruik van deze techniek in zijn `ins` functie: `ins` zou een ongeldige boom kunnen teruggeven en daarom zijn de oproepen naar `ins` als het ware afgeschermd door `makeBlack` of `balance`. De `ins` functie kan een boom teruggeven met een rode wortel en één rood kind, dit noemen we ook wel een infrarode boom. Een infrarode boom kan altijd hersteld worden door de wortel zwart te kleuren maar dit vergroot de hoogte met één. Tijdens het toevoegen van een element vermijden we liever veranderingen van hoogte omdat als één kind hoger wordt dan het andere, we de boom moeten herbalanceren. De recursieve oproepen naar `ins` zijn daarom afgeschermd door oproepen naar `balance`. Het is tevens het algoritme voor herbalancering waar de oplossing van Okasaki zijn elegantie vandaan haalt. Als het resultaat van een recursieve oproep naar `ins` een infrarode boom is, dan moet de wortel van het kind waarin we de waarde hebben toegevoegd, rood zijn geweest en dat kan alleen als de ouder van dat kind zwart was. Dit zijn nu net de voorwaarden voor de `balance` functie van Okasaki. Omdat we zulk een tijdelijke schending van de invarianten niet kunnen voorstellen in ons `Tree` type, hebben we een nieuw type nodig dat infrarode bomen kan voorstellen.

```
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
```

Omdat we nu twee verschillende types hebben voor red-black trees en infrarode bomen kunnen we de `balance` functie niet implementeren zoals Okasaki. Soms zou het type van het linker argument infrarood moeten zijn en het rechtse red-black en soms omgekeerd. We kunnen dit gedrag wel vatten in een nieuw type.

```
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
```

Dankzij de flexibele mixfix syntax van Agda kunnen we nu deze toepassing uit Haskell, `balance B infra b c`, als volgt schrijven in Agda: `balance (infra ◂ b ◂ c)`. We hebben ook geen nood meer aan het kleur argument, waarom komt later aan bod. We komen nog één type te kort, sommige functies zoals `ins` kunnen zowel een geldige als een ongeldige boom teruggeven. Omdat deze voorgesteld worden door twee aparte types hebben we een type nodig dat werkt als een disjuncte som van die twee types.

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

Meer precisie kost meer werk, wat bij Okasaki één type was, zijn er nu vier. Hoe preciezer we willen zijn hoe meer we expliciet moeten maken: dit zal ook duidelijk worden in de implementaties van de functies. De eerste functie die we bekijken is `balance`.

```
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)
```

De implementatie gelijkt sterk op die van Okasaki maar er zijn subtiele verschillen. Het kleur argument is niet langer nodig, Okasaki gebruikt dit eigenlijk alleen in het geval dat hij niets doet met de invoer om de argumenten terug samen te stellen tot een geldige boom met dezelfde kleur wortel. Omdat onze functie enkel bomen aanvaard die gebalanceerd *moeten* worden en een gebalanceerde boom altijd rood is, weten we op voorhand wat de kleur van de boom is die we teruggeven. Dit kunnen we alleen maar doen omdat ons type nu strikt genoeg is om geldige bomen uit te sluiten van `balance`. Dit betekent wel dat we op de plaats waar we `balance` willen gebruiken, zeker moeten zijn dat we een ongeldige boom hebben. Om die beslissing te kunnen maken moeten we vaak meer gevallen van mekaar scheiden maar dit heeft ook voordelen. In definities zoals die van Okasaki worden sommige functies soms uitgevoerd zonder dat ze iets doen, als er dan een fout optreedt, moeten ook deze functies nagegaan worden omdat we niet op voorhand weten of de functies de waarde inderdaad onveranderd laten.

De functies die Okasaki met een locale definitie implementeert in `insert` zijn meer veranderd. Ze zijn niet meer lokaal gedefinieerd omdat de `where` expressie uit Agda maar geldt voor één vergelijking. De eenvoudigste van die functies is `blacken`, die hetzelfde doet als `makeBlack`.

```
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)
```

Het enige dat `blacken` doet, is de wortel van een boom zwart kleuren. Dit wordt bemoeilijkt omdat we dit zowel voor een geldige als voor een ongeldige boom moeten doen, vandaar het type `Treeish` voor het argument. Omdat we een nieuwe knoop teruggeven met een andere constructor moeten we de invoer volledig uit elkaar halen en terug samenstellen in het resultaat, daarom zijn er zoveel meer vergelijkingen. Dit is de eerste keer dat we zien dat het type van het resultaat een functie oproep is. In dit geval hangt het af van de kleur van de wortel van de invoer of de boom hoger wordt of niet: als een rode wortel zwart wordt gekleurd, verhoogt de boom. Daarom gebruiken we `if_then_else_` in het type en omdat we met dependent types werken is dit geen probleem: het resultaat van een functie kan een type zijn en dit type kunnen we ook meteen gebruiken als type.

Omdat `ins` niet meer lokaal gedefinieerd is moeten we het argument voor het element dat we gaan toevoegen expliciet maken. De functie `ins` geeft een red-black tree terug als de invoer een zwarte boom was en geeft een red-black tree of een infrarode boom terug, in de vorm van een `Treeish`, als de invoer een rode boom was. De hoogte van het resultaat is dezelfde als die van de invoer, wat enkel verzekerd kan worden omdat we een infrarode boom kunnen teruggeven: het is het zwart maken van de wortel dat de hoogte kan veranderen. De kleur van de boom die we teruggeven, kunnen we niet zomaar bepalen, daarom is de kleur existentieel gekwantificeerd met een dependent pair, een koppel waarvan het type van het tweede element kan afhangen van de waarde van het eerste. Dit betekent wel dat we altijd een koppel van een kleur en een boom moeten teruggeven.

```
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
```

Wat meteen opvalt is dat de definitie van `ins` veel langer is dan in de implementatie van Okasaki. Dit is vooral omdat we veel preciezer moeten zijn. Wat voor boom we teruggeven en of we moeten balanceren of niet, maakt veel uit. De `with` expressie laat ons toe om de tussentijdse resultaten te inspecteren die we nodig hebben om te beslissen wat voor boom we teruggeven en of we moeten herbalanceren. De code leest als volgt, bijvoorbeeld voor het derde deel waar we `ins` toepassen op een boom met een zwarte wortel, `(B _ b _)`. Als het element dat we wensen toe te voegen, `a`, kleiner, `LT`, is dan het element in de wortel, `b`, en de toevoeging van het element aan het linkse kind, `ins a l`, heeft als resultaat `c , RB t` dan is de boom die we teruggeven zwart, `B , B t b r`. Als daarentegen het resultaat van `ins a l`, `.R , IR t` is, dan is het resultaat sowieso een rode boom en moeten we herbalanceren, `R , balance (t ◂ b ◂ r)`. De *dot patterns* in Agda, zoals `.R`, betekenen dat die waarde de enige mogelijke waarde is wat door Agda nagegaan wordt.

De `insert` functie die we nu kunnen definiëren is preciezer dan die van Okasaki, maar is niet zo precies mogelijk: de mogelijke hoogtes van het resultaat zijn beperkt maar de hoogte is niet volledig bepaald. In de import declaratie voor de disjuncte som, `Sum`, hebben we de constructors een naam gegeven die aangeeft of een boom van gelijke hoogte is als de invoer, `h+0` of één hoger geworden is `h+1` dit maakt het resultaat iets makkelijker leesbaar. Dat het resultaat een disjuncte som is, is een toegeving. Het probleem met preciezer zijn, is dat het moeilijk op voorhand te bepalen is of een boom hoger zal worden na het toevoegen van een element. Een functie die dit bepaalt, is wel gemakkelijk te schrijven en kunnen we gebruiken in het type maar Agda kan niet redeneren over zo'n complexe functie en in combinatie met de totaliteitsvoorwaarde voor functies in Agda zorgt dit voor problemen waar een invoer, die eigenlijk niet kan gegeven worden, ervoor zorgt dat we een uitvoer zouden moeten geven die we niet kunnen opstellen. Een oplossing hiervoor kan waarschijnlijk gevonden worden met een bewijs dat duidelijk maakt voor Agda wat wel en niet kan voorkomen. De aanzienlijke moeilijkheid om zo een bewijs op te stellen tegenover de

extra precisie die we verkrijgen voor de hoogte is in dit geval niet de moeite waard gevonden. Hier hebben we dus gekozen voor eenvoud over precisie.

```
insert : ∀{c h} → (a : A) → (t : Tree c h)
         → Tree B h ⊎ Tree B (suc h)
insert {R} a t with ins a t
... | R , t' = h+1 (blacken t')
... | B , t' = h+0 (blacken t')
insert {B} a t with ins a t
... | R , t' = h+1 (blacken (RB t'))
... | B , t' = h+0 (blacken (RB t'))
```

De `insert` functie neemt een waarde en een red-black tree en geeft een red-black tree terug van dezelfde hoogte of één hoger. De gevallen voor een rode en een zwarte boom moeten we opsplitsen omdat `ins` respectievelijk een `Tree` of een `Treeish` teruggeeft.

Nu rest er ons enkel nog de definities voor verzamelingen op basis van de bomen te geven. Ze zijn niet even handig geformuleerd als die van Okasaki maar de focus van deze gevalstudie ligt hier ook niet op.

```
-- Simple Set Operations
set : ∀{c h} → Set _
set {c}{h} = Tree c h

empty : set
empty = E

member : ∀{c h} → (a : A) → set {c}{h} → Bool
member a E = false
member a (R l b r) with a ≤ b
... | LT = member a l
... | EQ = true
... | GT = member a r
member a (B l b r) with a ≤ b
... | LT = member a l
... | EQ = true
... | GT = member a r
```

Ten slotte is er nog een voorbeeld van hoe we met deze bomen kunnen werken, het interessantste hieraan zijn de bewijsjes onderaan die aantonen dat een bepaalde boom gelijk is aan een toevoeging van een element aan een andere boom.

```
open import Relation.Binary.Properties.DecTotalOrder ℕ-DTO
ℕ-STO : StrictTotalOrder _ _ _
ℕ-STO = strictTotalOrder
```

```
open module rbtree = RBTree ℕ-STO

t0 : Tree B 2
t0 = B (R (B E 1 E) 2 (B (R E 3 E) 5 (R E 7 E)))
       8
       (B E 9 (R E 10 E))

t1 : Tree B 3
t1 = B (B (B E 1 E) 2 (B E 3 E))
       4
       (B (B E 5 (R E 7 E)) 8 (B E 9 (R E 10 E)))

t2 : Tree B 3
t2 = B (B (B E 1 E) 2 (B E 3 E))
       4
       (B (R (B E 5 E) 6 (B E 7 E)) 8 (B E 9 (R E 10 E)))

open import Relation.Binary.PropositionalEquality
t1≡t0+4 : h+1 t1 ≡ insert 4 t0
t1≡t0+4 = refl

t2≡t1+6 : h+0 t2 ≡ insert 6 t1
t2≡t1+6 = refl
```

4.3 Red-Black Trees in Haskell

In Haskell beginnen we opnieuw met de declaratie van de LANGUAGE pragma's die we nodig hebben en de definitie van natuurlijke getallen. Deze keer gebruiken we geen singleton types dus blijft het eenvoudiger. Ook de kleuren zijn nog steeds eenvoudig.

```
{-# LANGUAGE GADTs, DataKinds, KindSignatures #-}
module RedBlackTree where

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

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

Het `Tree` type is vrij gelijkaardig aan het `Tree` type uit de Agda code.

```
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 grootste verschil, buiten de namen van de constructors, is dat het `Tree` type een extra parameter heeft voor het polymorfisme. In Agda konden we door de parameter van de module als het ware overal aannemen dat het type van de elementen gekend was. Haskell heeft geen geparametriseerde modules dus maken we al onze types polymorf en voegen we constraints toe aan de functies die elementen moeten kunnen vergelijken.

Wat we nog niet zijn tegengekomen is het implementeren van een typeclass voor een GADT, dit kan wel eens verrassend moeilijk zijn. Daarom tonen we hier een voorbeeld voor de `Eq` typeclass.

```
instance Eq a => Eq (Tree c h a) where
  ET == ET = True
  RT l a r == RT m b s = a == b && l == m && r == s
  -- Black trees need further pattern matching because of cl and cr
  BT ET a ET == BT ET b ET =
    a == b
  BT ET a r@(RT {}) == BT ET b s@(RT {}) =
    a == b && r == s
  --
  BT l@(RT {}) a ET == BT m@(RT {}) b ET =
    a == b && l == m
  BT l@(RT {}) a r@(RT {}) == BT m@(RT {}) b s@(RT {}) =
    a == b && l == m && r == s
  BT l@(RT {}) a r@(BT {}) == BT m@(RT {}) b s@(BT {}) =
    a == b && l == m && r == s
  --
  BT l@(BT {}) a r@(RT {}) == BT m@(BT {}) b s@(RT {}) =
    a == b && l == m && r == s
  BT l@(BT {}) a r@(BT {}) == BT m@(BT {}) b s@(BT {}) =
    a == b && l == m && r == s
  _ == _ = False
```

Omdat de kleuren van de kinderen van een zwarte knoop onbepaald zijn moeten we veel meer *pattern matches* doen omdat Haskell anders niet genoeg kan afleiden over de hoogtes van de kinderen. Het enige nut van deze typeclass is om te kunnen testen of een boom die we uit `insert` krijgen gelijk is aan een andere boom wat alleen tijdens de uitvoering kan bepaald worden.

Omdat het `Tree` type opnieuw specifieker is dan bij Okasaki, komen we hetzelfde probleem tegen en hebben we een aantal extra types nodig.

```
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
  (:>:) :: 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
```

Op een paar verschillen na, zoals de GADTs waar we in Agda inductive families gebruiken, het polymorfisme en het feit dat Haskell geen mixfix notatie toelaat. Gelijken deze types sterk op de types uit de Agda implementatie.

De `balance` functie is opnieuw zeer gelijkaardig.

```
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)
```

De functie `blacken` heeft een iets minder precies type. Functies op het niveau van types zijn in Haskell veel meer beperkt, net omdat er een verschil is tussen waarden en types, terwijl in Agda alle types waarden zijn. De TypeFamilies extensie [27] van GHC biedt iets wat lijkt op een functie op het typeniveau. Hier kiezen we opnieuw voor een eenvoudigere oplossing door het type minder precies te maken.

```
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)
```

Het type dat we teruggeven is nu een disjuncte som, we moeten dus expliciet aangeven of het resultaat dezelfde hoogte heeft of één hoger geworden is.

De `ins` functie heeft ook een iets minder streng type, we geven nu altijd een `Treeish` terug in plaats van af en toe een `Tree`. In dit geval vervangen we het dependent pair uit Agda door een disjuncte som. Dit gaat alleen als het type waarover het dependent pair existentieel kwantificeert, exact twee elementen heeft. In deze functie moeten we ook de volgorde van elementen kunnen bepalen, daarom leggen we een `Ord` constraint op, op het type van de elementen.

```
-- Surprisingly difficult to find the right formulation
-- (ins in a pattern guard)
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)
```

De functie maakt nu gebruik van *pattern guards* [28] omdat `ins` altijd een `Treeish` teruggeeft en we aan de boom of infrarode boom moeten geraken die daar in zit. Dit is niet gewoon een kwestie van *pattern matchen* en het is enkel gelukt met *pattern guards*, `case` expressies zouden dus ook moeten werken. Dat het type dat `ins` teruggeeft altijd `Treeish` is, zou problemen geven in Agda. Omdat Agda een totale programmeertaal is, zouden we als we willen *pattern matchen* op een functie die een `Treeish` teruggeeft, een *match* moeten voorzien voor elk geval dat mogelijk is volgens het type. Concreet zou dit willen zeggen voor `ins` dat we een *match* moeten voorzien voor een infrarode boom, ook al geven we als invoer een zwarte boom, wat eigenlijk niet kan voorkomen. In het kort komt het erop neer dat we in Agda een functie zouden moeten definiëren om het geval te dekken dat niet kan voorkomen. Dit heeft natuurlijk niet veel zin en het is logischer om het type preciezer te maken. In Haskell geeft dit geen probleem omdat we partiële functies kunnen definiëren. Omdat er niet gecontroleerd wordt op totaliteit kan het wel zijn dat we ergens een geval over het hoofd zien dat toch kan voorkomen, dus moeten we beter opletten.

De `insert` functie is terug eenvoudiger omdat `ins` nu altijd een `Treeish` teruggeeft. Het type is ongeveer dat uit Agda, op polymorfisme en de `Ord` constraint na. En we gebruiken weer een *pattern guard* om het tussentijds resultaat uit te pakken vooraleer we het aan `blacken` kunnen geven.

```
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'
```

De functies voor verzamelingen hebben ongeveer hetzelfde nadeel als in Agda.

```
-- Simple Set operations
-- Partial Type Signatures might allow 'hiding' the color and height
type Set c h a = Tree c h a

empty :: Set B Z a
empty = ET

member :: Ord a => a -> Set c h a -> Bool
member _ ET = False
member a (RT l b r)
  | a < b = member a l
  | a == b = True
  | a > b = member a r
member a (BT l b r)
  | a < b = member a l
  | a == b = True
  | a > b = member a r
```

En tenslotte kunnen we opnieuw een voorbeeld geven van het gebruik, jammer genoeg zijn de *bewijzen* deze keer alleen na te gaan door ze uit te voeren.

```
t0 :: Tree B (S (S Z)) Integer
t0 = BT (RT (BT ET 1 ET) 2 (BT (RT ET 3 ET) 5 (RT ET 7 ET)))
        8
        (BT ET 9 (RT ET 10 ET))

t1 :: Tree B (S (S (S Z))) Integer
t1 = BT (BT (BT ET 1 ET) 2 (BT ET 3 ET))
        4
        (BT (BT ET 5 (RT ET 7 ET)) 8 (BT ET 9 (RT ET 10 ET)))

t2 :: Tree B (S (S (S Z))) Integer
t2 = BT (BT (BT ET 1 ET) 2 (BT ET 3 ET))
        4
        (BT (RT (BT ET 5 ET) 6 (BT ET 7 ET)) 8 (BT ET 9 (RT ET 10
         ↪  ET)))

-- Would a proof with refl and equality require the entire tree at
 ↪  type
-- level?
t1_is_t0_plus_4 :: Bool
t1_is_t0_plus_4 = t1 == t0_plus_4
  where Right t0_plus_4 = insert 4 t0

t2_is_t1_plus_6 :: Bool
```

```
t2_is_t1_plus_6 = t2 == t1_plus_6
  where Left t1_plus_6 = insert 6 t1
```

4.4 Besluit

Uit deze gevalstudie is duidelijk dat we altijd kunnen kiezen hoever we de verificatie doordrijven. De belangrijke invarianten over de volgorde van element in een zoekboom zijn hier niet opgenomen in het typesysteem. Dit kan verschillende motivaties hebben, misschien vinden we het te eenvoudig om de vergelijkingen voor volgorde juist te doen en hebben we geen behoefte aan de zekerheid die we zouden krijgen van een statische verificatie. Ook hebben we gezien dat als we af en toe een type een beetje verzwakken, dat de code dan heel eenvoudig kan blijven. In deze gevalstudie was de code in Haskell helemaal niet moeilijker te schrijven omdat we gebruik gemaakt hebben van het feit dat Haskell partiële functies toelaat. Verder zijn we in onze opzet geslaagd om ervoor te zorgen dat bepaalde functies enkel geldige bomen terug kunnen geven. In een interface van een library met datastructuren kan dit handig van pas komen omdat we kunnen garanderen aan de gebruiker dat wat hij krijgt altijd een geldige datastructuur is.

Wat betreft de implementatie van zoekbomen is deze gevalstudie eerder een voorbeeld van een kleine stap richting formalisatie. Af en toe heeft iemand een heel goed idee en in dit geval is er bijvoorbeeld een artikel van Conor McBride [29] met de titel: “How to Keep Your Neighbours in Order.” In dit artikel formuleert McBride een elegante en heel erg algemene voorstelling voor geördende bomen en andere datastructuren.

De volledige code voor dit hoofdstuk is terug te vinden in bijlagen C.2 en D.2. In het volgende hoofdstuk zien we nog een besluit over het geheel.

Hoofdstuk 5

Besluit

Dependent types zijn expressief en kunnen vele eigenschappen die we anders ongezegd laten, duidelijk en correct formaliseren zonder veel moeite. De afweging tussen moeite en statische garanties verdwijnt zeker niet maar is well redelijk flexibel te bepalen. Gewoon met vectoren werken in plaats van lijsten kost eigenlijk geen moeite en kan wel al mooie eigenschappen opleveren. Uiteindelijk moet de afweging voorlopig nog vaak gemaakt worden op basis van ervaring. Dependent types zijn nog niet zolang terug te vinden in *echte* programmeertalen en er zijn daardoor nog geen regels die we kunnen volgen over hoeveel statische verificatie het grootste voordeel biedt tegenover de kleinste inspanning.

Wat we ook gezien hebben is dat, ook al zijn talen met dependent types nog niet klaar voor algemeen gebruik, de ideeën die we erin opdoen gedeeltelijk toepasbaar zijn in een taal zoals Haskell. Leren programmeren met dependent types is dus geen tijdverspilling. Het heeft een dieper inzicht in het gebruik van types in andere talen tot gevolg. En de technieken kunnen af en toe ook overgezet worden naar die talen.

De gevalstudies hebben laten zien dat bepaalde concepten helemaal niet moeilijk te formaliseren zijn met dependent types. En dat door redelijk eenvoudige formalisaties toch redelijk wat fouten kunnen worden voorkomen. We hebben ook gezien dat Haskell redelijk ver te drijven is naar het dependently typed programmeren, ook al wordt het steeds lastiger hoe dichter we bij dependent types proberen komen. Hoewel ontwikkelingen zoals het singleton package voor verbetering kunnen zorgen. Wat misschien nog de belangrijkste conclusie uit de gevalstudies is, is dat als we een kleine toegeving doen op de precisie van de types, we af en toe een grote vermindering van inspanning kunnen krijgen. En, zolang de meeste eigenschappen nog in het typesysteem uitgedrukt zijn, kunnen we die uit ons hoofd zetten en nadenken over belangrijkere problemen.

Alle code die besproken is, is beschikbaar in de bijlagen alsook op het internet `https://github.com/toonn/pbdtt`.

Bijlagen

Bijlage A

KoopaTroopas in Agda

```
{-

    Verified Koopa Troopa Movement
             Toon Nolten

-}

module koopa where
  open import Data.Nat
  open import Data.Fin renaming (_+_ to _F+_; _<_ to _F<_; suc to fsuc;
    zero to fzero)
  open import Data.Vec renaming (map to vmap; lookup to vlookup;
                                 replicate to vreplicate)
  open import Data.Unit
  open import Data.Empty

  module Matrix where
    data Matrix (A : Set) : ℕ → ℕ → Set where
      Mat : {w h : ℕ} → Vec (Vec A w) h → Matrix A w h

    lookup : ∀ {w h} {A : Set} → Fin h → Fin w → Matrix A w h → A
    lookup row column (Mat rows) = vlookup column (vlookup row rows)
  open Matrix

  data Color : Set where
    Green : Color
    Red : Color

  data KoopaTroopa : Color → Set where
    _KT : (c : Color) → KoopaTroopa c

  data Material : Set where
    gas    : Material
    -- liquid : Material
    solid  : Material
```

```
data Clearance : Set where
  Low  : Clearance
  High : Clearance
  Ultimate  : Clearance

record Position : Set where
  constructor pos
  field
    x   : ℕ
    y   : ℕ
    mat : Material
    clr : Clearance

data _c>_ : Color → Clearance → Set where
  <red>   : ∀ {c} → c c> Low
  <green> : Green c> High

data _follows_⟨_⟩ : Position → Position → Color → Set where
  stay  : ∀ {c x y} → pos x y gas Low follows pos x y gas Low ⟨ c ⟩
  next  : ∀ {c cl x y}{{_ : c c> cl}} →
          pos (suc x) y gas cl follows pos x y gas Low ⟨ c ⟩
  back  : ∀ {c cl x y}{{_ : c c> cl}} →
          pos x y gas cl follows pos (suc x) y gas Low ⟨ c ⟩
  -- jump  : ∀ {c x y} → pos x (suc y) gas High follows pos x y gas Low ⟨
   ↪  c ⟩
  fall  : ∀ {c cl x y} → pos x y gas cl follows pos x (suc y) gas High ⟨
   ↪  c ⟩


infixr 5 _↠⟨_⟩_
data Path {c : Color} (Koopa : KoopaTroopa c) :
     Position → Position → Set where
  []  : ∀ {p} → Path Koopa p p
  _↠⟨_⟩_ : {q r : Position} → (p : Position) → q follows p ⟨ c ⟩
             → (qs : Path Koopa q r) → Path Koopa p r

ex_path : Path (Red KT) (pos 0 0 gas Low) (pos 0 0 gas Low)
ex_path = pos 0 0 gas Low ↠⟨ next ⟩
          pos 1 0 gas Low ↠⟨ back ⟩
          pos 0 0 gas Low ↠⟨ stay ⟩ []

matterToPosVec : {n : ℕ} → Vec Material n → Vec Material n → ℕ → ℕ
                 → Vec Position n
matterToPosVec []           []        _ _ = []
matterToPosVec (mat :: mats) (under :: unders) x y =
  pos x y mat cl :: matterToPosVec mats unders (x + 1) y
    where
      clearance : Material → Material → Clearance
      clearance gas   gas   = High
```

```
      clearance gas   solid = Low
      clearance solid _     = Ultimate
      cl = clearance mat under

matterToPosVecs : {w h : ℕ} → Vec (Vec Material w) h → Vec (Vec Position
 ↪  w) h
matterToPosVecs [] = []
matterToPosVecs (_::_ {y} mats matss) =
  matterToPosVec mats (unders matss gas) 0 y :: matterToPosVecs matss
  where
    unders : ∀ {m n ℓ}{A : Set ℓ} → Vec (Vec A m) n → A → Vec A m
    unders [] fallback = vreplicate fallback
    unders (us :: _) _ = us

matsToMat : {w h : ℕ} → Vec (Vec Material w) h → Matrix Position w h
matsToMat matss = Mat (reverse (matterToPosVecs matss))

□_ : ∀{n} → Vec Material n → Vec Material (suc n)
□ xs = gas :: xs
■_ : ∀{n} → Vec Material n → Vec Material (suc n)
■ xs = solid :: xs
infixr 5 □_
infixr 5 ■_
example_level : Matrix Position 10 7
example_level = matsToMat (
  □ □ □ □ □ □ □ □ □ □ [] ::
  □ □ □ □ □ □ ■ ■ ■ □ [] ::
  □ □ □ □ □ □ □ □ □ □ [] ::
  □ □ □ ■ ■ ■ □ □ □ □ [] ::
  □ □ □ □ □ □ □ □ □ □ [] ::
  ■ □ □ □ □ □ □ □ □ ■ [] ::
  ■ ■ ■ ■ ■ □ □ ■ ■ ■ [] :: [])

_<'_ : ℕ → ℕ → Set
m <' zero = ⊥
zero <' suc n = ⊤
suc m <' suc n = m <' n

fromNat : ∀ {n}(k : ℕ){_ : k <' n} → Fin n
fromNat {zero} k {}
fromNat {suc n} zero = fzero
fromNat {suc n} (suc k) {p} = fsuc (fromNat k {p})

f : ∀ {n}(k : ℕ){_ : k <' n} → Fin n
f = fromNat

p : (x : Fin 10) → (y : Fin 7) → Position
p x y = lookup y x example_level

red_path_one : Path (Red KT) (p (f 7) (f 6)) (p (f 8) (f 6))
```

```
red_path_one = p (f 7) (f 6) ↠⟨ back ⟩
               p (f 6) (f 6) ↠⟨ next ⟩
               p (f 7) (f 6) ↠⟨ next ⟩
               p (f 8) (f 6) ↠⟨ stay ⟩ []

red_path_two : Path (Red KT) (p (f 2) (f 1)) (p (f 3) (f 1))
red_path_two = p (f 2) (f 1) ↠⟨ back ⟩
               p (f 1) (f 1) ↠⟨ next ⟩
               p (f 2) (f 1) ↠⟨ next ⟩
               p (f 3) (f 1) ↠⟨ next ⟩
               p (f 4) (f 1) ↠⟨ back ⟩
               p (f 3) (f 1) ↠⟨ stay ⟩ []

-- -- Type error shows up 'late' because 'cons' is right associative
-- red_nopath_one : Path (Red KT) (p (f 1) (f 1)) (p (f 0) (f 1))
-- red_nopath_one = p (f 1) (f 1) ↠⟨ back ⟩
--                  p (f 0) (f 1) ↠⟨ stay ⟩ []

-- -- Red KoopaTroopa can't step into a wall
-- red_nopath_two : Path (Red KT) (p (f 1) (f 1)) (p (f 0) (f 1))
-- red_nopath_two = p (f 1) (f 1) ↠⟨ back ⟩ []

-- -- Red Koopa Troopa can't step into air
-- red_nopath_three : Path (Red KT) (p (f 4) (f 1)) (p (f 5) (f 1))
-- red_nopath_three = p (f 4) (f 1) ↠⟨ next ⟩ []

-- Any path that is valid for red Koopa Troopas, is also valid for green
-- Koopa Troopas because we did not constrain Koopa Troopas to only turn
-- when there is an obstacle
green_path_one : Path (Green KT) (p (f 7) (f 6)) (p (f 8) (f 6))
green_path_one = p (f 7) (f 6) ↠⟨ back ⟩
                 p (f 6) (f 6) ↠⟨ next ⟩
                 p (f 7) (f 6) ↠⟨ next ⟩
                 p (f 8) (f 6) ↠⟨ stay ⟩ []

green_path_two : Path (Green KT) (p (f 7) (f 6)) (p (f 5) (f 0))
green_path_two = p (f 7) (f 6) ↠⟨ back ⟩
                 p (f 6) (f 6) ↠⟨ back ⟩
                 p (f 5) (f 6) ↠⟨ fall ⟩
                 p (f 5) (f 5) ↠⟨ fall ⟩
                 p (f 5) (f 4) ↠⟨ back ⟩
                 p (f 4) (f 4) ↠⟨ back ⟩
                 p (f 3) (f 4) ↠⟨ back ⟩
                 p (f 2) (f 4) ↠⟨ fall ⟩
                 p (f 2) (f 3) ↠⟨ fall ⟩
                 p (f 2) (f 2) ↠⟨ fall ⟩
                 p (f 2) (f 1) ↠⟨ back ⟩
                 p (f 1) (f 1) ↠⟨ next ⟩
                 p (f 2) (f 1) ↠⟨ next ⟩
                 p (f 3) (f 1) ↠⟨ next ⟩
```

```
                p (f 4) (f 1) ↠⟨ next ⟩
                p (f 5) (f 1) ↠⟨ fall ⟩ []

-- -- Green Koopa Troopa can't step into a wall
-- green_nopath_one : Path (Green KT) (p (f 1) (f 1)) (p (f 0) (f 1))
-- green_nopath_one = p (f 1) (f 1) ↠⟨ back ⟩ []
```

Bijlage B

KoopaTroopas in Haskell

```
{-

    Verified Koopa Troopa Movement
             Toon Nolten

-}

{-# LANGUAGE GADTs, DataKinds, KindSignatures #-}
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}

module Koopa where

data Nat = Z | S Nat
data Natty :: Nat -> * where
  Zy :: Natty Z
  Sy :: Natty n -> Natty (S n)
natter :: Natty n -> Nat
natter Zy = Z
natter (Sy n) = S (natter n)
data NATTY :: * where
  Nat :: Natty n -> NATTY
nattyer :: Nat -> NATTY
nattyer Z = Nat Zy
nattyer (S n) = case nattyer n of Nat m -> Nat (Sy m)


data Fin :: Nat -> * where
  Zf :: Fin (S n)
  Sf :: Fin n -> Fin (S n)
intToFin :: Natty n -> Integer -> Fin n
intToFin Zy _ = error "Fin Z is an empty type"
intToFin (Sy n) i
  | i <  0 = error "Negative Integers cannot be represented by a finite
   ↪  natural"
  | i == 0 = Zf
```

```
  | i >  0 = Sf (intToFin n (i-1))


data Vec :: * -> Nat -> * where
  V0   :: Vec a Z
  (:>) :: a -> Vec a n -> Vec a (S n)
infixr 5 :>

vlookup :: Fin n -> Vec a n -> a
vlookup (Zf)   (a :> _)  = a
vlookup (Sf n) (_ :> as) = vlookup n as

vreplicate :: Natty n -> a -> Vec a n
vreplicate Zy     _ = V0
vreplicate (Sy n) a = a :> vreplicate n a

vreverse :: Vec a n -> Vec a n
vreverse V0 = V0
vreverse (a :> V0) = a :> V0
vreverse (a :> as) = vreverse' as a
  where
    vreverse' :: Vec a n -> a -> Vec a (S n)
    vreverse' V0 x' = x' :> V0
    vreverse' (x :> xs) x' = x :> vreverse' xs x'


data Matrix :: * -> Nat -> Nat -> * where
  Mat :: Vec (Vec a w) h -> Matrix a w h

mlookup :: Fin h -> Fin w -> Matrix a w h -> a
mlookup row column (Mat rows) = vlookup column (vlookup row rows)

data Color = Green | Red
data Colorry :: Color -> * where
  Greeny :: Colorry Green
  Redy   :: Colorry Red

data KoopaTroopa :: Color -> * where
  KT :: Colorry c -> KoopaTroopa c

data Material = Gas | Solid
data Matty :: Material -> * where
  Gasy   :: Matty Gas
  Solidy :: Matty Solid
data MATTY :: * where
  Mater :: Matty m -> MATTY
mattyer :: Material -> MATTY
mattyer Gas = Mater Gasy
mattyer Solid = Mater Solidy
```

```
data Clearance = Low | High | Ultimate
data Clearry :: Clearance -> * where
  Lowy  :: Clearry Low
  Highy :: Clearry High
  Ultimatey  :: Clearry Ultimate
data CLEARRY :: * where
  Clear :: Clearry cl -> CLEARRY
clearryer :: Clearance -> CLEARRY
clearryer Low = Clear Lowy
clearryer High = Clear Highy
clearryer Ultimate = Clear Ultimatey


data Position = Pos { getX      :: Nat
                    , getY      :: Nat
                    , matter :: Material
                    , clr    :: Clearance
                    }
data Positionny :: Position -> * where
  Posy :: Natty x -> Natty y -> Matty m -> Clearry cl
          -> Positionny (Pos x y m cl)
data POSITIONNY :: * where
  Posit :: Positionny p -> POSITIONNY
positionnyer :: Position -> POSITIONNY
positionnyer (Pos x y m cl)
  | Nat xY <- nattyer x
  , Nat yY <- nattyer y
  , Mater mY <- mattyer m
  , Clear clY <- clearryer cl
  = Posit (Posy xY yY mY clY)

class CoClr (c :: Color) (cl :: Clearance)
instance CoClr c Low
instance CoClr Green High

data Follows :: Position -> Position -> Color -> * where
  Stay :: Follows (Pos x y Gas Low) (Pos x y Gas Low) c
  Next :: CoClr c cl => Follows (Pos (S x) y Gas cl) (Pos x y Gas Low) c
  Back :: CoClr c cl => Follows (Pos x y Gas cl) (Pos (S x) y Gas Low) c
  Fall :: Follows (Pos x y Gas cl) (Pos x (S y) Gas High) c

data Path :: Color -> Position -> Position -> * where
  P0 :: Path c p p
  Pcons :: Positionny p -> Follows q p c -> Path c q r -> Path c p r


-- Examples

exPath :: Path Red (Pos Z Z Gas Low) (Pos Z Z Gas Low)
exPath = Pcons (Posy Zy Zy Gasy Lowy) Next
```

```
        (Pcons (Posy (Sy Zy) Zy Gasy Lowy) Back
        (Pcons (Posy Zy Zy Gasy Lowy) Stay P0))

matterToPosVec :: Vec Material n -> Vec Material n -> Nat -> Nat
                  -> Vec Position n
matterToPosVec V0 V0 _ _ = V0
matterToPosVec (mat :> mats) (under :> unders) x y =
  Pos x y mat cl :> matterToPosVec mats unders (S x) y
    where
      clearance :: Material -> Material -> Clearance
      clearance Gas   Gas   = High
      clearance Gas   Solid = Low
      clearance Solid _     = Ultimate
      cl = clearance mat under

matterToPosVecs :: Natty w -> Natty h -> Vec (Vec Material w) h
                   -> Vec (Vec Position w) h
matterToPosVecs _ _ V0 = V0
matterToPosVecs w (Sy z) (mats :> matss) =
  matterToPosVec mats (unders w matss Gas) Z y  :> matterToPosVecs w z
   ↪  matss
    where
      y = natter (Sy z)
      unders :: Natty m -> Vec (Vec a m) n -> a -> Vec a m
      unders m V0 fallback = vreplicate m fallback
      unders _ (us :> _) _ = us

mattersToMatrix :: Natty w -> Natty h -> Vec (Vec Material w) h
                   -> Matrix Position w h
mattersToMatrix w h matss = Mat (vreverse (matterToPosVecs w h matss))

o :: Material
o = Gas
c :: Material
c = Solid

exampleLevel :: Matrix Position
                      (S(S(S(S(S(S(S(S(S Z)))))))))) -- 10
                      (S(S(S(S(S(S(S Z))))))) -- 7
exampleLevel = mattersToMatrix
                 (Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy)))))))))) -- 10
                 (Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy)))))))) -- 7
                 (
  (o :> o :> o :> o :> o :> o :> o :> o :> o :> V0) :>
  (o :> o :> o :> o :> o :> c :> c :> c :> o :> V0) :>
  (o :> o :> o :> o :> o :> o :> o :> o :> o :> V0) :>
  (o :> o :> o :> c :> c :> c :> o :> o :> o :> V0) :>
  (o :> o :> o :> o :> o :> o :> o :> o :> o :> V0) :>
  (c :> o :> o :> o :> o :> o :> o :> o :> c :> V0) :>
  (c :> c :> c :> c :> c :> o :> o :> c :> c :> V0) :> V0)
```

```
ix :: Integer -> Fin (S(S(S(S(S(S(S(S(S(S Z))))))))) -- 10
ix = intToFin (Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy))))))))) -- 10

iy :: Integer -> Fin (S(S(S(S(S(S(S Z)))))) -- 7
iy = intToFin (Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy)))))) -- 7

p :: Fin (S(S(S(S(S(S(S(S(S(S Z))))))))))
     -> Fin (S(S(S(S(S(S(S Z))))))
     -> POSITIONNY
p x y | Pos x' y' m' cl' <- mlookup y x exampleLevel
      , Nat xy <- nattyer x'
      , Nat yy <- nattyer y'
      , Mater my <- mattyer m'
      , Clear cly <- clearryer cl'
      = Posit (Posy xy yy my cly)


p01 = Posy Zy (Sy Zy) Solidy Ultimatey
p11 = Posy (Sy Zy) (Sy Zy) Gasy Lowy
p21 = Posy (Sy(Sy Zy)) (Sy Zy) Gasy Lowy
p22 = Posy (Sy(Sy Zy)) (Sy(Sy Zy)) Gasy Highy
p23 = Posy (Sy(Sy Zy)) (Sy(Sy(Sy Zy))) Gasy Highy
p24 = Posy (Sy(Sy Zy)) (Sy(Sy(Sy(Sy Zy)))) Gasy Highy
p31 = Posy (Sy(Sy(Sy Zy))) (Sy Zy) Gasy Lowy
p34 = Posy (Sy(Sy(Sy Zy))) (Sy(Sy(Sy(Sy Zy)))) Gasy Lowy
p41 = Posy (Sy(Sy(Sy(Sy Zy)))) (Sy Zy) Gasy Lowy
p44 = Posy (Sy(Sy(Sy(Sy Zy)))) (Sy(Sy(Sy(Sy Zy)))) Gasy Lowy
p51 = Posy (Sy(Sy(Sy(Sy(Sy Zy))))) (Sy Zy) Gasy Highy
p54 = Posy (Sy(Sy(Sy(Sy(Sy Zy))))) (Sy(Sy(Sy(Sy Zy)))) Gasy Lowy
p55 = Posy (Sy(Sy(Sy(Sy(Sy Zy))))) (Sy(Sy(Sy(Sy(Sy Zy))))) Gasy Highy
p56 = Posy (Sy(Sy(Sy(Sy(Sy Zy))))) (Sy(Sy(Sy(Sy(Sy(Sy Zy)))))) Gasy Highy
p66 = Posy (Sy(Sy(Sy(Sy(Sy(Sy Zy))))))
           (Sy(Sy(Sy(Sy(Sy(Sy Zy)))))) Gasy Lowy
p76 = Posy (Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy)))))))
           (Sy(Sy(Sy(Sy(Sy(Sy Zy)))))) Gasy Lowy
p86 = Posy (Sy(Sy(Sy(Sy(Sy(Sy(Sy(Sy Zy))))))))
           (Sy(Sy(Sy(Sy(Sy(Sy Zy)))))) Gasy Lowy

redPathOne :: Path Red (Pos (S(S(S(S(S(S Z)))))))
                            (S(S(S(S(S(S Z)))))) Gas Low)
                       (Pos (S(S(S(S(S(S(S Z))))))))
                            (S(S(S(S(S(S Z)))))) Gas Low)
redPathOne = Pcons p76 Back
           $ Pcons p66 Next
           $ Pcons p76 Next
           $ Pcons p86 Stay P0

redPathTwo :: Path Red (Pos (S(S Z))    (S Z) Gas Low)
```

```
                      (Pos (S(S(S Z))) (S Z) Gas Low)
redPathTwo = Pcons p21 Back
           $ Pcons p11 Next
           $ Pcons p21 Next
           $ Pcons p31 Next
           $ Pcons p41 Back
           $ Pcons p31 Stay P0

-- Because Stay only allows a position with material Gas to follow from
-- a position with material Gas, this is a type error
-- redNoPathOne :: Path Red (Pos (S Z) (S Z) Gas Low)
--                          (Pos Z (S Z) Solid Ultimate)
-- redNoPathOne = Pcons p11 Back
--              £ Pcons p01 Stay P0

-- Red Koopa Troopa can't step into a wall
-- This is for the same reason as in redNoPathOne but with the Back
-- constructor
-- redNoPathTwo :: Path Red (Pos (S Z) (S Z) Gas Low)
--                          (Pos Z (S Z) Solid Ultimate)
-- redNoPathTwo = Pcons p11 Back P0

-- Red Koopa Troopa can't step into air
-- Here the problem is that there is no instance for CoClr Red High which
-- is necessary for Next, however the solution GHC suggests is to add it
-- while it was actually not defined on purpose
-- redNoPathThree :: Path Red (Pos (S(S(S(S Z)))) (S Z) Gas Low)
--                            (Pos (S(S(S(S(S Z))))) (S Z) Gas High)
-- redNoPathThree = Pcons p41 Next P0

-- Any path that is valid for red Koopa Troopas, is also valid for green
-- Koopa Troopas because we did not constrain Koopa Troopas to only turn
-- when there is an obstacle
greenPathOne :: Path Green (Pos (S(S(S(S(S(S(S Z)))))))
                                (S(S(S(S(S(S Z)))))) Gas Low)
                           (Pos (S(S(S(S(S(S(S Z)))))))
                                (S(S(S(S(S(S Z)))))) Gas Low)
greenPathOne = Pcons p76 Back
             $ Pcons p66 Next
             $ Pcons p76 Next
             $ Pcons p86 Stay P0

greenPathTwo :: Path Green (Pos (S(S(S(S(S(S(S Z)))))))
                                (S(S(S(S(S(S Z)))))) Gas Low)
                           (Pos (S(S(S(S(S Z))))) Z Gas Low)
greenPathTwo = Pcons p76 Back
             $ Pcons p66 Back
             $ Pcons p56 Fall
             $ Pcons p55 Fall
             $ Pcons p54 Back
```

```
              $ Pcons p44 Back
              $ Pcons p34 Back
              $ Pcons p24 Fall
              $ Pcons p23 Fall
              $ Pcons p22 Fall
              $ Pcons p21 Back
              $ Pcons p11 Next
              $ Pcons p21 Next
              $ Pcons p31 Next
              $ Pcons p41 Next
              $ Pcons p51 Fall P0

-- Green Koopa Troopa can't step into a wall
-- Exactly the same as for redNoPathTwo
-- greenNoPathOne :: Path Green (Pos (S Z) (S Z) Gas Low)
--                              (Pos Z (S Z) Solid Ultimate)
-- greenNoPathOne = Pcons p11 Back P0
```

Bijlage C

Red-Black Trees in Agda

Eerst volgt een redelijk letterlijke vertaling van de code van Okasaki naar Agda om te tonen dat dependent types absoluut niet verplicht zijn. Daarna de code van de tweede case in Agda.

C.1 Okasaki

```
module Okasaki where

open import Data.Bool using (Bool; true; false) renaming (T to So; not to
 ↪  ¬)

open import Data.Nat hiding (_<_; _≤_; _≟_; compare)
  renaming (decTotalOrder to ℕ-DTO)

open import Relation.Binary hiding (_⇒_)

module RBTree {a ℓ}(order : StrictTotalOrder a ℓ ℓ) where

  open module sto = StrictTotalOrder order
  A = Carrier

  pattern LT = tri< _ _ _
  pattern EQ = tri≈ _ _ _
  pattern GT = tri> _ _ _
  _≤_ = compare


  data Color : Set where
    R B : Color

  _=ᶜ_ : Color → Color → Bool
  R =ᶜ R = true
  B =ᶜ B = true
  _ =ᶜ _ = false
```

```
Height = ℕ

data Tree : Set a where
  E : Tree
  T : Color → Tree → A → Tree → Tree


set = Tree

empty : set
empty = E

member : A → set → Bool
member x E = false
member x (T _ a y b) with x ≤ y
... | LT = member x a
... | EQ = true
... | GT = member x b

insert : A → set → set
insert x s = makeBlack (ins s)
  where
    balance : Color → set → A → set → set
    balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
    balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
    balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
    balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
    balance color a x b = T color a x b

    ins : set → set
    ins E = T R E x E
    ins (T color a y b) with x ≤ y
    ... | LT = balance color (ins a) y b
    ... | EQ = T color a y b
    ... | GT = balance color a y (ins b)

    makeBlack : set → set
    makeBlack E = E
    makeBlack (T _ a y b) = T B a y b
```

C.2 Red-Black Trees

```
{-
  Verified Red-Black Trees
        Toon Nolten

  Based on Chris Okasaki's "Red-Black Trees in a Functional Setting"
  where he uses red-black trees to implement sets.
```

```
  Invariants
  ----------

  1. No red node has a red parent
  2. Every Path from the root to an empty node contains the same number of
     black nodes.
     (Empty nodes are considered black)
-}
module RedBlackTree where

open import Data.Bool using (Bool; true; false) renaming (T to So; not to
 ↪  ¬)
_⇒_ : ∀{ℓ₁ ℓ₂} → Set ℓ₁ → Set ℓ₂ → Set _
P ⇒ T = {{p : P}} → T
infixr 3 _⇒_

if_then_else_ : ∀{ℓ}{A : Set ℓ} b → (So b ⇒ A) → (So (¬ b) ⇒ A) → A
if true  then t else f = t
if false then t else f = f

open import Data.Nat hiding (_<_; _≤_; _≟_; compare)
  renaming (decTotalOrder to ℕ-DTO)
open import Level hiding (suc)
open import Relation.Binary hiding (_⇒_)

open import Data.Sum using (_⊎_) renaming (inj₁ to h+0; inj₂ to h+1)
open import Data.Product using (Σ; Σ-syntax; _,_; proj₁; proj₂)

module RBTree {a ℓ}(order : StrictTotalOrder a ℓ ℓ) where

  open module sto = StrictTotalOrder order
  A = Carrier

  pattern LT = tri< _ _ _
  pattern EQ = tri≈ _ _ _
  pattern GT = tri> _ _ _
  _≤_ = compare


  data Color : Set where
    R B : Color

  _=ᶜ_ : Color → Color → Bool
  R =ᶜ R = true
  B =ᶜ B = true
  _ =ᶜ _ = false

  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)

  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

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

  -- Insertion

  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)

  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)

  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

  insert : ∀{c h} → (a : A) → (t : Tree c h)
           → Tree B h ⊎ Tree B (suc h)
  insert {R} a t with ins a t
  ... | R , t' = h+1 (blacken t')
  ... | B , t' = h+0 (blacken t')
  insert {B} a t with ins a t
  ... | R , t' = h+1 (blacken (RB t'))
  ... | B , t' = h+0 (blacken (RB t'))


  -- Simple Set Operations
  set : ∀{c h} → Set _
  set {c}{h} = Tree c h

  empty : set
  empty = E

  member : ∀{c h} → (a : A) → set {c}{h} → Bool
  member a E = false
  member a (R l b r) with a ≤ b
  ... | LT = member a l
  ... | EQ = true
  ... | GT = member a r
  member a (B l b r) with a ≤ b
  ... | LT = member a l
  ... | EQ = true
  ... | GT = member a r



-- Usage example

open import Relation.Binary.Properties.DecTotalOrder ℕ-DTO
ℕ-STO : StrictTotalOrder _ _ _
ℕ-STO = strictTotalOrder

open module rbtree = RBTree ℕ-STO

t0 : Tree B 2
t0 = B (R (B E 1 E) 2 (B (R E 3 E) 5 (R E 7 E)))
       8
       (B E 9 (R E 10 E))
```

```
t1 : Tree B 3
t1 = B (B (B E 1 E) 2 (B E 3 E))
       4
       (B (B E 5 (R E 7 E)) 8 (B E 9 (R E 10 E)))

t2 : Tree B 3
t2 = B (B (B E 1 E) 2 (B E 3 E))
       4
       (B (R (B E 5 E) 6 (B E 7 E)) 8 (B E 9 (R E 10 E)))

open import Relation.Binary.PropositionalEquality
t1≡t0+4 : h+1 t1 ≡ insert 4 t0
t1≡t0+4 = refl

t2≡t1+6 : h+0 t2 ≡ insert 6 t1
t2≡t1+6 = refl
```

Bijlage D

Red-Black Trees in Haskell

Eerst volgt de code van Okasaki dan de code voor de case en daarna nog een korte vertaling van de `if_the_else_` functie in Agda naar een functie op typeniveau in Haskell.

D.1 Okasaki

```
module RedBlackTree where

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

type Set a = Tree a

empty :: Set elt
empty = E

member :: Ord elt => elt -> Set elt -> Bool
member x E = False
member x (T _ a y b) | x <  y = member x a
                     | x == y = True
                     | x >  y = member x b

-- originally without type signature
balance :: Color -> Tree elt -> elt
           -> Tree elt -> Tree elt
balance B (T R (T R a x b) y c) z d =
  T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d =
  T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) =
  T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) =
  T R (T B a x b) y (T B c z d)
balance color a x b = T color a x b
```

```
insert :: Ord elt => elt -> Set elt
          -> Set elt
insert x s = makeBlack (ins s)
  where
    ins E = T R E x E
    ins (T color a y b)
      | x <  y = balance color (ins a) y b
      | x == y = T color a y b
      | x >  y = balance color a y (ins b)

    makeBlack (T _ a y b) = T B a y b
```

D.2 Red-Black Trees

```
{-
  Verified Red-Black Trees
        Toon Nolten

  Based on Chris Okasaki's "Red-Black Trees in a Functional Setting"
  where he uses red-black trees to implement sets.

  Invariants
  ----------

  1. No red node has a red parent
  2. Every Path from the root to an empty node contains the same number of
     black nodes.
    (Empty nodes are considered black)
-}
{-# LANGUAGE GADTs, DataKinds, KindSignatures #-}
module RedBlackTree where

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

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

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

instance Eq a => Eq (Tree c h a) where
  ET == ET = True
  RT l a r == RT m b s = a == b && l == m && r == s
  -- Black trees need further pattern matching because of cl and cr
  BT ET a ET == BT ET b ET =
    a == b
  BT ET a r@(RT {}) == BT ET b s@(RT {}) =
    a == b && r == s
```

```
  --
  BT l@(RT {}) a ET == BT m@(RT {}) b ET =
    a == b && l == m
  BT l@(RT {}) a r@(RT {}) == BT m@(RT {}) b s@(RT {}) =
    a == b && l == m && r == s
  BT l@(RT {}) a r@(BT {}) == BT m@(RT {}) b s@(BT {}) =
    a == b && l == m && r == s
  --
  BT l@(BT {}) a r@(RT {}) == BT m@(BT {}) b s@(RT {}) =
    a == b && l == m && r == s
  BT l@(BT {}) a r@(BT {}) == BT m@(BT {}) b s@(BT {}) =
    a == b && l == m && r == s
  _ == _ = False


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
  (:>:) :: 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

--Insertion

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 :: 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)

-- Surprisingly difficult to find the right formulation
-- (ins in a pattern guard)
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)

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'


-- Simple Set operations
-- Partial Type Signatures might allow 'hiding' the color and height
type Set c h a = Tree c h a

empty :: Set B Z a
empty = ET

member :: Ord a => a -> Set c h a -> Bool
member _ ET = False
member a (RT l b r)
  | a < b = member a l
  | a == b = True
  | a > b = member a r
member a (BT l b r)
  | a < b = member a l
  | a == b = True
  | a > b = member a r



-- Usage example
t0 :: Tree B (S (S Z)) Integer
t0 = BT (RT (BT ET 1 ET) 2 (BT (RT ET 3 ET) 5 (RT ET 7 ET)))
        8
        (BT ET 9 (RT ET 10 ET))

t1 :: Tree B (S (S (S Z))) Integer
t1 = BT (BT (BT ET 1 ET) 2 (BT ET 3 ET))
        4
        (BT (BT ET 5 (RT ET 7 ET)) 8 (BT ET 9 (RT ET 10 ET)))
```

```
t2 :: Tree B (S (S (S Z))) Integer
t2 = BT (BT (BT ET 1 ET) 2 (BT ET 3 ET))
        4
        (BT (RT (BT ET 5 ET) 6 (BT ET 7 ET)) 8 (BT ET 9 (RT ET 10 ET)))

-- Would a proof with refl and equality require the entire tree at type
-- level?
t1_is_t0_plus_4 :: Bool
t1_is_t0_plus_4 = t1 == t0_plus_4
  where Right t0_plus_4 = insert 4 t0

t2_is_t1_plus_6 :: Bool
t2_is_t1_plus_6 = t2 == t1_plus_6
  where Left t1_plus_6 = insert 6 t1
```

D.3 Type Level If

```
{-# LANGUAGE TypeFamilies, DataKinds, GADTs, KindSignatures #-}

data Tree :: Bool -> * -> * where
  E :: Tree False a
  T :: Tree False a -> a -> Tree False a -> Tree True a

type family If (cond :: Bool) thenn elsse :: *

type instance If True a b = a
type instance If False a b = b

type family And (a :: Bool) (b :: Bool) :: Bool
type instance And True True = True
type instance And False b = False
type instance And a False = False

f :: Tree h a -> If (And h h) Bool Integer
f E = 5
f (T _ _ _) = True

main :: IO ()
main = do print $ "f E:"
          print $ f E
          print $ "f (T E 3 E)"
          print $ f (T E 3 E)
```

Bibliografie

[1] U. Norell, "Dependently Typed Programming in Agda," in *Proceedings of the 6th International Conference on Advanced Functional Programming*, ser. AFP'08. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 230–266, ISBN-10 3-642-04651-7, ISBN-13 978-3-642-04651-3.

[2] (2015, Jul.) Haskell Language. [Online]. Available: https://www.haskell.org/

[3] R. Hindley, "The Principal Type-Scheme of an Object in Combinatory Logic," *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, Dec. 1969.

[4] B. Kernighan and D. Ritchie, *The C Programming Language*, 2nd ed.

[5] (2015, Aug.) The Java Language Specification, Java SE 8 Edition. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/html/index.html

[6] H. Curry, "Functionality in Combinatory Logic," in *Proceedings of the National Academy of Sciences*, ser. 20, 1934, pp. 584–590.

[7] P. Martin-Löf, *Intuitionistic Type Theory.* Bibliopolis, 1984, ISBN 88-7088-105-9.

[8] (2015, Jul.) The Glorious Glasgow Haskell Compiler. [Online]. Available: https://www.haskell.org/ghc/

[9] B. Pierce, *Types and Programming Languages.* MIT Press, 2002, ch. 29: Type Operators and Kinding, ISBN 0-262-16209-1.

[10] (2015, Aug.) The Agda Standard Library. [Online]. Available: http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.StandardLibrary

[11] P. Dybjer, "Inductive Families," *Formal Aspects of Computing*, vol. 6, pp. 440–465, 1997.

[12] N. Oury and W. Swierstra, "The Power of Pi," 2008.

[13] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised).* MIT Press, 1997, ISBN 0-262-63181-4.

[14] X. Leroy. (2015, Aug.) The OCaml system. [Online]. Available: http://caml.inria.fr/pub/docs/manual-ocaml/

[15] (2015, Jul.) Datatype Promotion. [Online]. Available: https://downloads.haskell.org/~ghc/7.6.3/docs/html/users_guide/promotion.html

[16] H. Xi, C. Chen, and G. Chen. (2003, Jan.) Guarded Recursive Datatype Constructors. New York, NY, USA.

[17] (2015, Aug.) Mario. [Online]. Available: https://en.wikipedia.org/wiki/Mario

[18] (2015, Aug.) Nintendo. [Online]. Available: https://www.nintendo.com/

[19] (2015, Aug.) Super Mario Bros. [Online]. Available: https://en.wikipedia.org/wiki/Super_Mario_Bros.

[20] (2015, Aug.) How to Jump Over the Flagpole on stage 1-1 in Super Mario Bros. [Online]. Available: https://youtu.be/dzlmNdP-ApU

[21] S. Lindley and C. McBride, "Hasochism: The Pleasure and Pain of Dependently Typed haskell Programming," 2008.

[22] (2015, Jul.) Type Operators. [Online]. Available: https://downloads.haskell.org/~ghc/7.6.3/docs/html/users_guide/data-type-extensions.html

[23] (2015, Aug.) The singletons library. [Online]. Available: http://www.cis.upenn.edu/~eir/packages/singletons/

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw-Hill, 2001, ch. 13: Red-Black Trees, pp. 273–301, ISBN 0-262-03293-7.

[25] C. Okasaki, "Functional Pearls: Red-Black Trees in a Functional Setting," *Journal of Functional Programming*, vol. 9 (04), pp. 471–477, Jan. 1999.

[26] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," *SIGPLAN Not.*, vol. 35, no. 9, pp. 268–279, Sep. 2000.

[27] (2015, Jul.) Type families. [Online]. Available: https://downloads.haskell.org/~ghc/7.6.3/docs/html/users_guide/type-families.html

[28] (2015, Jul.) Pattern guard. [Online]. Available: https://wiki.haskell.org/Pattern_guard

[29] C. McBride, "How to Keep Your Neighbours in Order," 2008.

Fiche masterproef

Student: Toon Nolten

Titel: Programmeren en Bewijzen met Dependently Typed Talen

Engelse titel: Programming and Proofs in Dependently Typed Languages

UDC: 681.3

Korte inhoud:
In this thesis we present two case studies, both in Agda, a fully dependently typed programming language, and in Haskell, a functional programming language with a Hindley-Milner style type system. After formulating a model of a problem in Agda, we mimic this model in Haskell, relying on Glasgow Haskell Compiler extensions to provide type system features we lack in standard Haskell. We try to illustrate how much, or little effort goes into static verification. And hint at reducing reliance on test suites by encoding simple properties in our types.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen

Promotoren: Dr. Dominique Devriese
Prof. dr. ir. Frank Piessens

Assessoren: Prof. dr. Marc Denecker
Dr. Thomas Heyman

Begeleider: Jesper Cockx