Bachelorscriptie
Informatica / Informatiekunde

Radboud Universiteit

Toepassingen in COcaml

Auteur:
Martin Huyben
S4205332

Inhoudelijk begeleider:
Dr. A. Martins Da Silva
alexandra.cs.ru.nl

16 juni 2015

Samenvatting

COcaml is een uitbreiding van de programmeertaal Ocaml dat om kan gaan met vormen van oneindigheid. COcaml is nog onbekend, wat jammer is, want op deze manier is het moeilijk om er toepassingen voor te vinden. In deze scriptie leg ik uit wat COcaml is en wat het onderscheidt van Ocaml.

Inhoudsopgave

1 Inleiding **2**

2 Solvers **9**
2.1 De Iterator Solver 9
2.2 De Constructor Solver 19
2.3 Wat COcaml kan samengevat 28
2.4 De appears solver en zelfgemaakte solvers 29

3 Voorbeelden **31**
3.1 List module functies voor oneindige regulaire datatypen: list.ml 31
3.2 Abstract interpretation: ai.ml 38
3.3 Free variables: fv.ml 40
3.4 kansberekening met de gaussian solver: probability.ml 41
3.5 Eigen uitwerking: Graphs 43
3.6 Eigen uitwerking: 2-dimensionale oneindige matrices 51

4 Suggesties **53**
4.1 Aanpassingen 53
4.1.1 Console 53
4.1.2 Bugs 55
4.1.3 Gebruiksvriendelijkheid 58
4.2 Mogelijkheden 59
4.2.1 oneindige logische datastructuren 59
4.2.2 Corecursieve functies als first-class citizens 60
4.2.3 Ingewikkelde oneindige datastructuren 63

5 Gerelateerd onderzoek **67**

6 Verder Onderzoek **68**

A Appendix **69**

Hoofdstuk 1

Inleiding

COcaml[1] is een uitbreiding op de strikte functionele programmeertaal Ocaml[2] dat om kan gaan met regulaire oneindige datastructuren. In deze scriptie leg ik uit hoe je COcaml kunt gebruiken.

Voordat ik dit kan doen leg ik eerst de volgende begrippen uit:

- Functionele programmeertalen
- Het verschil tussen strikte en luie functionele programmeertalen
- Ocaml, een strikte functionele taal
- Oneindige regulaire datastructuren
- COcaml, een uitbreiding op Ocaml dat om kan gaan met oneindige regulaire datastructuren
- Het doel van deze scriptie

Functionele programmeertalen
De meest bekende programmeertalen zijn imperatieve programmeertalen: talen waarin je stap voor stap beschrijft wat de computer moet berekenen. Voorbeelden van zulke talen zijn Java en C. Elke programmeertaal heeft voordelen en nadelen; een nadeel van imperatieve talen is dat er na elke stap neveneffecten mogelijk zijn. Als je een functie hebt en argumenten voor die functie, kan de uitkomst niet alleen afhangen van de argumenten, maar ook van aangeroepen variabelen. In functionele talen is de uitkomst bij een functie alleen afhankelijk van de gegeven argumenten. In een functionele taal beschrijf je niet de stappen die de computer moet nemen, maar je beschrijft hoe de uitkomst gemaakt kan worden uit de argumenten. De compiler haalt daar zelf uit welke stappen in welke volgorde nodig zijn. Dit maakt functionele talen heel geschikt voor computers met meerdere processoren, aangezien het verdelen van de benodigde stappen toch al door de compiler gebeurt. Verder is in een standaard functionele taal alles een functie; een int (een getal) is een functie zonder argumenten. Dit geeft mogelijkheden zoals:

- een functie als argument geven. Zo is er een functie `map` die, gegeven een functie en een lijst, de functie toepast op elk element in de lijst.

- uitstellen van argumenten geven. Stel je hebt een functie die twee argumenten nodig heeft. Als je die het eerste argument geeft, levert dat een functie op die nog een argument nodig heeft. Deze functie kun je gewoon meegeven aan andere functies etc.

Strikt en lui

Functionele programmeertalen zijn onderverdeelbaar in strikt en lui. In strikte talen worden argumenten van een functie geëvalueerd voordat ze worden gebruikt. In luie programmeertalen worden argumenten pas gebruikt als ze nodig zijn. Dit geeft luie programmeertalen twee bijzondere eigenschappen:

- Ten eerste zorgt dit er voor dat als iets niet nodig is, het ook niet wordt geëvalueerd. Dit kan voor snellere performance zorgen.
- Ten tweede geeft het de programmeur de mogelijkheid om compleet onmogelijke dingen te definiëren, zoals bijvoorbeeld een functie die deelt door nul, zonder dat het programma vastloopt, mits hij de functie niet ergens anders aanroept. Een ander voorbeeld is een oneindige lijst, iets dat gewoon gedefiniëerd kan worden, zolang nooit wordt geprobeerd het laatste element op te vragen.

Maar dit brengt ook meteen het grootste nadeel mee van luie talen: je kunt er als programmeur *per ongeluk* een stuk code schrijven waar niets van klopt, en je komt er pas achter als je het foute gedeelte aanroept in een andere functie. Dit maakt het debuggen van luie talen moeilijk.

Ocaml

Ocaml is een strikte functionele programmeertaal met een focus op veiligheid[2]. Omdat het strikt is, is het namelijk makkelijk om tests op functies uit te voeren, aangezien de uitkomst alleen afhangt van de argumenten en onmogelijke uitkomsten niet worden goedgerekend.En omdat het static-typed is (dus een int (afgerond getal) wordt niet zomaar een float (kommagetal) als de situatie daarvoor vraagt, wat in Java bijvoorbeeld wel gebeurt), kan onopgemerkte typeswitching geen problemen veroorzaken.

De syntax van Ocaml lijkt veel op de syntax van andere functionele talen. Functies zien er zo uit:

```
let functienaam argumenten =
        functionbody
```

let is hier het keyword waaraan te herkennen is dat het om een functie gaat. Dit wordt gevolgd door de naam van de functie en de argumenten. Je kunt een functie maken zonder argumenten. Dan maak je een declaratie. In de functionbody wordt beschreven hoe uit de argumenten het antwoord kan worden gemaakt. Een functie kan gebruik maken van de keywords **match** .. **with** of **function**, maar dit hoeft niet. Hier is een functie die daar geen gebruik van maakt. Gegeven een getal geeft het het kwadraat:

```
let kwadraat x = x*x
```

De keywords **match** .. **with** en **function** geven aan dat in de functionbody een speciale syntax voorkomt van de vorm:

```
| case -> antwoord
```

Zo'n regel heet een stelling. Als het gegeven argument voldoet aan de case, dan wordt het antwoord gegeven. Hieronder volgen twee voorbeelden waar beide keywords worden gebruikt. Beide functies doen hetzelfde: gegeven twee getallen en een boolean, geven ze het eerste getal als de boolean true is en het tweede getal als de boolean false is.

```
let choose_nr a b c = match c with
| true -> a
| x -> b

let choose_nr a b = function
| true -> a
| x -> b

val choose_nr : 'a -> 'a -> bool -> 'a = <fun>
```

Dit geeft de volgende uitkomst:

```
choose_nr 1 2 true
 - : int = 1
choose_nr 1 2 false
 - : int = 2
```

Een paar dingen zouden hier op moeten vallen:

- Een antwoord en een uitkomst zijn niet hetzelfde. Een antwoord is de stap die gedaan moet worden als een argument in een case past. Een uitkomst is het resultaat van wanneer een functie zijn argumenten krijgt.
- Uitkomsten worden gegeven met de syntax - : type = waarde, wat zowel het type als de waarde van de uitkomst geeft. Ocaml kan zelf uitvogelen welke types bij de waardes horen. Dit hoeft de programmeur niet te specificeren. Als er een functie wordt gedefinieerd of een variabele wordt gedeclareerd staat er val naam : type = waarde.
- De functie choose_nr heeft als type 'a -> 'a -> bool -> 'a, wat betekent dat het een functie is met drie argumenten, twee van het type 'a en een van het type bool. De uitkomst is ook van het type 'a. Het type 'a is het meest generieke type. Deze functie is dus geschikt voor elk type. Hierbij geldt wel dat het eerste argument hetzelfde type moet hebben als het tweede argument. Had dit niet gehoeven, dan was het tweede argument van het type 'b geweest, dus een andere letter om aan te geven dat het een ander type mag zijn.
- Stellingen worden in volgorde geëvalueerd. Stel we hebben true als argument. Dan voldoet dat aan de eerste case, en wordt het eerste argument gegeven. Hebben we niet true als argument, dan voldoet het

niet aan de eerste case en komen we bij de tweede case. De tweede case is een ongespecificeerde variabele. Hier voldoet dus alles aan, want een ongespecificeerde variabele kan alles zijn. Dus dan voldoet ons argument aan de tweede case en krijgen we het tweede argument als antwoord. We hadden als tweede case ook gewoon `false` kunnen gebruiken en dan deed de functie exact hetzelfde.

- Het verschil tussen **match** .. **with** en **function** is dat **function** standaard het laatste argument gebruikt in de cases. Dit argument hoeft dan ook niet meer weergegeven te worden. Bij **match** .. **with** kun je zelf kiezen welk argument je gebruikt. Je kunt hier op de plek van de puntjes ook al argumenten bewerken, zoals bijvoorbeeld **match not** `c` **with**.

Het laatste belangrijke keyword dat ik ga uitleggen is **rec**. Dit keyword betekent dat de functie zichzelf aan mag roepen. Hiermee kun je dus recursieve functies schrijven. Een voorbeeld is deze functie, die gegeven een lijst de som van de elementen teruggeeft:

```
let rec opgeteld arg = match arg with
| [ ] -> 0
| h :: t -> h + opgeteld t

opgeteld [1;2;3]
 - : int = 6
```

Hier wordt gelijk wat nieuwe syntax geïntroduceerd: `[ ]` betekent een lege lijst. `h :: t` betekent een lijst met minstens een element. Het betekent gelijk dat het eerste element van die lijst `h` wordt genoemd en de rest van de lijst `t` wordt genoemd. Dus wat doet deze code? Als het een lege lijst krijgt geeft het 0. Als het een niet-lege lijst krijgt, geeft het het eerste getal plus de opgetelde rest van de lijst. De functie roept zichzelf dus telkens aan met een nieuw argument totdat het nieuwe argument de lege lijst is. In stappen ziet dit er zo uit:

```
opgeteld [1;2;3]                    (*past in de tweede case*)
1 + opgeteld [2;3]                  (*past in de tweede case*)
1 + 2 + opgeteld [3]                (*past in de tweede case*)
1 + 2 + 3 + opgeteld [ ]            (*past in de eerste case*)
1 + 2 + 3 + 0
6
```

De tweede stelling zorgt er dus voor dat de functie opnieuw wordt aangeroepen en de eerste stelling dat het aanroepen stopt. Daarom wordt de tweede stelling de recursive stelling genoemd en de eerste stelling de base stelling. Corresponderend hiermee zijn de base case, base antwoord, recursive case en recursive antwoord. Belangrijk bij recursief aanroepen is dat het nieuwe argument altijd meer op de base case lijkt dan het oude argument. Het is namelijk de bedoeling dat het nieuwe argument uiteindelijk de base case wordt, anders blijft de functie eeuwig zichzelf aanroepen. In het geval van de functie `opgeteld` wordt het nieuwe argument een steeds kleinere lijst, totdat het leeg is.

Regulaire oneindige datastructuren
Een datastructuur is een vorm waarin data kan worden opgeslagen. Voorbeelden zijn lijsten, tupels en zelfgedefinieerde types.

Een regulaire oneindige datastructuur is een oneindige datastructuur die zichzelf na een tijdje herhaalt. Een voorbeeld van een regulaire oneindige datastructuur is de oneindige lijst [1; 2; 1; 2; 1; 2; ...] . Een voorbeeld van een irregulaire oneindige datastructuur is de oneindige lijst [1; 2; 3; 4; 5; ...] waar het volgende element constant iets groter is. Deze lijst is irregulair omdat de lijst nooit ergens in de lijst weer opnieuw begint te tellen vanaf 1. In theorie zijn oneindige datastructuren niet mogelijk in Ocaml. In praktijk zijn er wat uitzonderingen. Zo mogen regulaire oneindige lijsten gedefinieerd worden. Een voorbeeld is de lijst **let rec** alt = 1::2:: alt. alt is een recursieve functie zonder argumenten en zonder base case. Dit betekent dat deze functie nooit stopt. Het is een lijst die constant de getallen 1 en 2 toevoegt aan zichzelf. Normaal zou deze functie voor een stackoverflow zorgen, aangezien het nooit stopt met zichzelf aanroepen, maar in dit specifieke geval (het geval waarin een lijst recursief wordt gedeclareerd) werkt Ocaml daar omheen. Voor de programmeur werkt deze lijst precies zoals je verwacht van een oneindige lijst. Voor de compiler is het een lijst die alleen groter wordt als een functie een grotere versie van de lijst nodig heeft. Het evalueert deze lijst dus lui. Regulaire oneindige lijsten zijn dus mogelijk in OCaml, irregulaire oneindige lijsten niet.

Wat kun je met deze lijsten? Je kunt wel:
- elementen op een bepaalde plek in de lijst opvragen
- elementen op een bepaalde plek in de lijst wijzigen

Je kunt niet:
- opvragen of een kenmerk geldt voor alle elementen in de lijst
- alle elementen in de lijst wijzigen

Het is logisch dat dit niet kan, want hiervoor zou je zou je elk element moeten bekijken en in een oneindige lijst zijn dit er oneindig veel. In Ocaml zijn ook Streams te definiëren, die wel irregulair en oneindig kunnen zijn. Daar gelden dezelfde limitaties voor.

In de uitbreiding COcaml is elke regulaire oneindige datastructuur definieerbaar die te maken is met lijsten tupels en zelfgemaakte types en hebben ze de bovengenoemde limitaties niet. Voorbeelden van zulke datastructuren zijn:

```
let rec zeros = 0::zeros
let rec a = (0,a)
type tree = Branch of tree*tree | Leaf of int
let rec b = Branch (Leaf 0, b)
let rec c = 1::1::zeros
let rec d = (1,(1,zeros))
let rec e = (zeros, 1)
let rec f = (zeros, f)
```

De bijbehorende uitkomsten in COcaml zijn:

```
val zeros : int list = [0; 0; 0; 0; 0; ...]
val a : (int * (int * 'a)) = 0, 0, 0, 0, 0, ...
val b : tree = Branch (Leaf 0, Branch (Leaf 0, Branch (Leaf 0,
    ...)))
val c : int list = [1; 1; 0; 0; 0; 0; ...]
val d : (int * (int * int list)) = 1, (1, [0; 0; 0; 0; 0; ...])
val e : (int list * int) = [0; 0; 0; 0; 0; ...], ...
val f : (int list * (int list * 'a)) = [0; 0; 0; 0; 0; ...],
    ...
```

Een flink aantal dingen kunnen hier worden opgemerkt. Van boven naar beneden is dat:

- zeros is een lijst, a is een tupel en b is een datastructuur van het zelfgemaakte type tree.
- Een tupel is een datastructuur waar de lengte van vaststaat en niet alle elementen tot hetzelfde type hoeven te behoren. Vergelijk dit met een lijst waar de lengte niet vaststaat en wel alle elementen tot hetzelfde type moeten behoren.
- zeros zou ook te maken zijn in Ocaml. Alle andere functies zouden voor een stackoverflow hebben gezorgd.
- Oneindigheid wordt weergegeven met drie puntjes ... . Dit betekent niet alleen dat de datastructuur oneindig is, maar ook dat de compiler de lijst nog niet verder heeft bekeken dan dat.
- a wordt bij de uitkomsten wat vreemd weergegeven. Het lijkt namelijk alsof a een oneindig grote tupel is. Dit is niet het geval. a ziet er eigenlijk zo uit: (0, (0, (0, (0, ...)))). Deze versie van COcaml is een testversie en er zijn dus zulke bugs.
- Je kunt in Ocaml (en dus ook in COcaml) zelf types maken. Dit doe je met het keyword **type**. Hier is het type tree gedefinieerd als ofwel een binaire tupel van trees ofwel een getal.
- c is de samenvoeging van een regulaire oneindige lijst en een eindige lijst. Regulaire oneindige datastructuren kunnen gewoon samengevoegd worden met eindige datastructuren en zijn dan nog steeds regulaire oneindige datastructuren. Zo is d de combinatie van een eindige tupel en een oneindige lijst.
- Bij de uitkomst van e zie je dat het tweede element van de tupel wordt weergegeven als ... , terwijl het 1 hoort te zijn. Dit is omdat de compiler zijn best doet om de oneindige datastructuren niet te ver te bekijken. Als het ze namelijk helemaal uit wil schrijven, loopt het programma alsnog vast, aangezien het dan oneindig lang schrijft. Soms is de compiler hier te rigoreus in. De 1 bestaat wel in het computergeheugen.
- Combinaties van regulaire oneindige datastructuren zijn ook nog steeds regulaire oneindige datastructuren. f is de combinatie van oneindige tupelrijen en oneindige lijsten. Zoals je ziet wordt na de oneindige lijst

de rest weergegeven als ... . De rest van de structuur is er wel in het computergeheugen. Het wordt gewoon niet weergegeven.

COcaml en deze scriptie

De versie van COcaml waar ik het in deze scriptie over heb is een testversie. Haar doel is niet om even perfect te zijn als een eindproduct, maar om de mogelijkheden van de taal te laten zien. Hierdoor heeft COcaml wat bugs/features die geen afbreuk doen aan de krachtigheid van de taal, maar hier en daar wel lelijk kunnen zijn. Ik behandel in hoofdstuk 4 , Suggesties, wat er mogelijk allemaal veranderd mag worden. Verder beschrijf ik in de scriptie bepaalde bugs/features wanneer nodig.

Het doel van COcaml is om te kunnen omgaan met regulaire oneindige datastructuren. Dit doet het door middel van solvers, waarmee zogenaamde corecursieve functies kunnen worden gemaakt. Deze solvers zijn niet moeilijk te begrijpen. Om dit te bewijzen leg ik in deze scriptie uit hoe je corecursieve functies maakt in hoofdstuk 2 , Solvers.

Deze versie van COcaml komt samen met wat voorbeeldfiles. In deze voorbeeldfiles zijn mogelijke toepassingen te zien. Verder heb ik zelf ook geëxperimenteerd met mogelijke toepassingen. Ik behandel al dit in hoofdstuk 3 , Voorbeelden.

Hoofdstuk 2

Solvers

COcaml geeft twee dingen: oneindige regulaire datastructuren en functies die er mee om kunnen gaan, genaamd corecursieve functies. In COcaml kun je elke oneindige regulaire datastructuur maken die te maken is met (combinaties van) lijsten, tupels en zelfgemaakte types. De corecursieve functies kun je maken met behulp van zogenaamde solvers. Een corecursieve functie ziet er uit als volgt.

```
1  let corec[solvernaam] functienaam argument =
2        functionbody
```

Op de plek 'solvernaam' kies je welke solver je wilt gebruiken. Er zijn drie standaard solvers en de mogelijkheid om je eigen solvers te definiëren. Maar voordat ik deze uitleg, vertel ik eerst de op dit moment grootste bugs/features:

- Corecursieve functies kunnen maar 1 argument hebben. Wil je een corecursieve functie die meerdere argumenten heeft, dan kun je de argumenten geven in een tupel. Een corecursieve functie zonder argumenten is mogelijk, maar heeft nooit praktisch nut ten opzichte van een gewone functie.

- De =**function** keyword werkt niet goed voor corecursieve functies. Het beste is om standaard het keyword **match** arg **with** te gebruiken.

Deze bugs zijn gemakkelijk rondom heen te werken. En dan nu de solvers.

2.1 De Iterator Solver

De iterator solver is bedoeld om eindige data uit een oneindige regulaire datastructuur te halen. De bijbehorende corecursieve functie ziet er zo uit:

```
1  let corec[iterator(solverantwoord)] functienaam argument =
2      functionbody
```

Voordat ik uitleg hoe de iterator werkt laat ik eerst een voorbeeld zien. Hieronder staat de corecursieve functie toset, die een set maakt van alle voorkomende variabelen van de gegeven lijst.

```
let aux arg = match arg with
| (x,y) -> if mem x y then y else x::y

let corec[iterator([ ])] toset arg = match arg with
| [ ] -> [ ]
| x::y -> aux (x, (toset y))
```

Waarbij

```
let rec alt = 1::2::alt
let rec zeros = 0::zeros
let t1 = toset [1;1;2]
let t2 = toset alt
let t3 = toset zeros
```

het volgende als uitkomst geeft:

```
val alt : int list = [1; 2; 1; 2; 1; 2; 1; 2; ...]
val zeros : int list = [0; 0; 0; 0; ...]
val t1 : int list = [1; 2]
val t2 : int list = [1; 2]
val t3 : int list = [0]
```

De functie aux voegt een gegeven element toe aan een gegeven lijst als het element nog niet voorkomt in de lijst. Het maakt gebruik van de functie mem. Dit is een functie uit de Ocaml list module. In deze module zitten veel voorkomende functies op lijsten. aux had hier genesteld kunnen worden in toset, maar ik vond dit overzichtelijker.

Er moet opgemerkt worden dat de functie toset erg lijkt op de functie die je zou schrijven voor eindige datastructuren. Stel we zouden zo'n functie schrijven, dan zou die er zo uit zien:

```
let rec toset arg = match arg with
| [ ] -> [ ]
| x::y -> aux (x, (toset y))
```

Het enige verschil is corec[iterator([ ])] in plaats van **rec**. Deze recursieve functie zou gegeven [1;1;2] de volgende stappen doen:

```
toset [1;1;2]
aux (1,(toset [1;2]))
aux (1,aux (1, (toset [2])))
aux (1, aux (1, aux (2, (toset [ ]))))
aux (1, aux (1, aux (2, [ ])))
aux (1, aux (1, [2]))
aux (1, [1;2])
[1;2]
```

Deze recursieve functie zou gegeven alt de volgende stappen doen:

```
toset alt
aux (1, (toset 2::alt))
```

```
3   aux (1, aux (2, (toset alt)))
4   aux (1, aux (2, aux (1 (toset 2::alt))))
5   aux (1, aux (2, aux (1, aux (2, (toset alt)))))
6   etc.
```

De recursieve functie zou nooit stoppen.

Dus wat doet de iterator solver in dit geval? Het berekent toset alt totdat het op een zeker moment de nieuwe aanroep vervangt door het solverantwoord. Dat ziet er zo uit:

```
1   toset alt
2   aux (1, (toset 2::alt))
3   aux (1, aux (2, (toset alt)))
4   aux (1, aux (2, aux (1 (toset 2::alt))))
5   aux (1, aux (2, aux (1, aux (2, (toset alt)))))
6   (*de aanroep wordt vervangen*)
7   aux (1, aux (2, aux (1, aux (2, [ ]))))
8   aux (1, aux (2, aux (1,[2])))
9   aux (1, aux (2, [1;2]))
10  aux (1, [1;2])
11  [1;2]
```

Bij regel 5 t/m 7 vervangt de iterator solver de aanroep door het solverantwoord. Hoe kiest de iterator solver wanneer het de aanroep vervangt? Het vervangt zodra:

- het argument al gebruikt is, en
- de uitkomst zouden we de aanroep vervangen gelijk is aan de uitkomst zouden we de aanroep vervangen toen het argument de vorige keer werd gebruikt.

Beide voorwaarden zijn vervuld bij regel 5. De eerste voorwaarde is voor ons makkelijk te controleren: alt is al gebruikt in regel 3. De tweede voorwaarde is ingewikkelder. Hieronder is een tabel waarin per stap de aanroep vervangen wordt door het solverantwoord. De bijbehorende uitkomst noemen we de solveruitkomst.

Aanroep is vervangen bij stap	Dit geeft	Solveruitkomst
1 toset alt	[ ]	[ ]
2 aux (1, (toset 2:: alt))	aux (1, [ ])	[1]
3 aux (1, aux (2, (toset alt)))	aux (1, aux (2, [ ]))	[1;2]
4 aux (1, aux (2, aux (1 (toset 2:: alt))))	aux (1, aux (2, aux (1 [ ])))	[1;2]
5 aux (1, aux (2, aux (1, aux (2, (toset alt)))))	aux (1, aux (2, aux (1, aux (2, [ ]))))	[1;2]

Zoals je ziet is de solveruitkomst gelijk aan de solveruitkomst van regel 3. Waarom deze voorwaarden? Omdat op dat moment, zou je verder blijven rekenen, er geen uitkomst meer komt dat je nog niet hebt gehad.

toset heeft elk mogelijk argument al gehad (alt en 2:: alt) en aux heeft ook elk mogelijk argument al gehad ((1,[ ]), (2,[ ]), (1,[2]), (2,[1]), (1,[1;2]) en (2,[1;2])). Deze argumenten komen respectievelijk voor in stap 2 als (1,[ ]),

stap 3 als (2,[ ]), stap 3 als (1, aux(2, [ ])), stap 4 als (2, aux(1, [ ])), stap 4 als (1, aux(2, aux(1, [ ]))) en stap 5 als (2, aux(1, aux(2, [ ]))). Daardoor heb je ook elk mogelijk antwoord van aux al gehad, dus je hebt elke uitkomst gehad.

Dus de iterator solver kan, zou een functie oneindig blijven loopen, de loop afbreken en de aanroep vervangen door het solverantwoord, mits duidelijk is dat de solveruitkomst toch niet meer verandert, zou de loop later zijn afgebroken.
Hieronder volgen verschillende paragrafen over wat de iterator solver kan.

De iterator kan een eindig antwoord uit een oneindige datastructuur halen

Dit is het voornaamste doel van de iterator. Hieronder volgen een paar voorbeelden. De functie contains2 geeft, gegeven een oneindige int list, true als er een 2 in de lijst voorkomt en false als dat niet zo is.

```
let corec[iterator(false)] contains2 arg = match arg with
| 2::y -> true
| x::y -> contains2 y
```

Het solverantwoord wordt gegeven als de functie in een oneindige loop zit. De functie zit alleen in een oneindige loop als er geen 2 in de lijst zit.

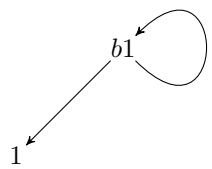
```
contains2 ones;;
 - : bool = false
contains2 alt;;
 - : bool = true
```

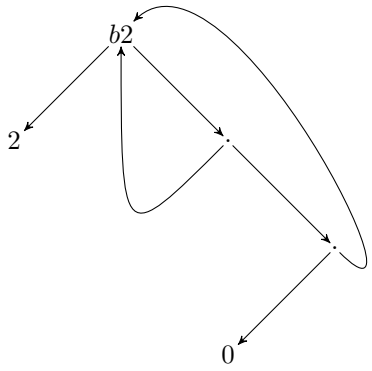
Deze functie geeft, gegeven een boomstructuur het hoogst voorkomende getal:

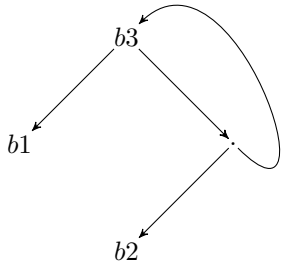
```
type tree = Branch of tree*tree | Leaf of int

let get_highest a b = if a>b then a else b

let corec[iterator(0)] highest arg = match arg with
| Leaf x -> x
| Branch (x,y) -> get_highest (highest x) (highest y)

let rec b1 = Branch (Leaf 1, b1)
let rec b2 = Branch (Leaf 2, Branch(b2, Branch(Leaf 0, b2)))
let rec b3 = Branch (b1, Branch(b2,b3))

let h1 = highest b1
let h2 = highest b2
let h3 = highest b3

val b1 : tree = Branch (Leaf 1, Branch (Leaf 1, Branch (Leaf
    1, ... )))
val b2 : tree = Branch (Leaf 2, Branch (Branch (Leaf 2,
    Branch (Branch ( ... , ... )), ... ))
```

```
    val b3 : tree = Branch (Branch (Leaf 1, Branch (Leaf 1,
        Branch (Leaf 1,  ... ))),  ... )
    val h1 : int = 1
    val h2 : int = 2
    val h3 : int = 2
```

Meer uitleg over bomen in COcaml komt in sectie 2.2, De Constructor Solver, pagina 22 en in sectie 3.5 , Graphs. De bomen b1, b2 en b3 kunnen weergegeven worden als:







De iterator stopt hier de recursie als elk getal minstens een keer is geweest. Zoals je ziet is er de functie get_highest. Deze functie bestaat zodat er geen if-statement in de functionbody van highest staat. Er is namelijk een bug waar als er in de functionbody van een iterator functie in het if-gedeelte, zowel als het then- of else-gedeelte een recursieve aanroep staat, de iterator

de loop niet kan breken en er een stackoverflow komt. De oorspronkelijke versie van de tweede stelling van highest was:

```
| Branch (x,y) -> if (highest x)>(highest y) then (highest x)
    else (highest y)
```

Hier komt een aanroep van highest voor in zowel het if-gedeelte als in het then-gedeelte (en het else-gedeelte), dus dit zorgt voor een stackoverflow.

De iterator kan een eindig antwoord halen uit een structuur waarvan niet bekend is of die oneindig is

De versie van de corecursieve functie contains2 werkt alleen voor oneindige datastructuren. Hieronder is de recursieve versie, die alleen werkt voor eindige datastructuren:

```
let rec contains2 arg = match arg with
| [ ] -> false
| 2::y -> true
| x::y -> contains2 y
```

Deze functie heeft twee base stellingen. Een voor als het antwoord true is en een voor als het antwoord false is. De vorige versie gaf false als bleek dat de functie aan het loopen was. Deze functie geeft false als de lijst eindigt. Deze twee functies kunnen makkelijk gecombineerd worden tot een corecursieve functie die met zowel eindige als oneindige lijsten om kan gaan.

```
let corec[iterator(false)] contains2 arg = match arg with
| [ ] -> false
| 2::y -> true
| x::y -> contains2 y
```

De functie highest is al geschikt voor eindige datastructuren.

```
let b4= Branch (Branch (Leaf 1, Leaf 2), Leaf 3)
let h4 = highest b4
val h4 : int = 3
```

Een iterator kan een loop niet stoppen als de loop constant een nieuw argument krijgt

Zie de volgende functie.

```
let corec[iterator([ ])] a x= a (1::x)
```

Waarbij

```
let b = a [ ]
```

een stackoverflow geeft. Hier krijgt elke nieuwe aanroep van a een steeds grotere lijst gevuld met enen als argument. Dit terwijl de solveruitkomst hetzelfde blijft. a gebruikt het argument namelijk niet in de solveruitkomst. De solveruitkomst is altijd [ ]. Waarom is dit? Omdat de solver niet met zekerheid kan zeggen dat de aanroep met volgende argument niet een andere

solveruitkomst geeft. Bij deze functie lijkt het overduidelijk dat er nooit een andere solveruitkomst komt, maar als er meerdere stellingen zijn of als er andere functies voorkomen in het antwoord, wordt het bepalen al moeilijker. Er is zelfs met zekerheid te zeggen dat dit onmogelijk is, aangezien uit het halting probleem (het is onmogelijk voor elke functie te bepalen of die ooit stopt) volgt dat het onmogelijk is de uitkomst van elke functie te bepalen. Hetzelfde geldt voor alle stellingen.

De iterator kan dus alleen loops herkennen en afbreken waarbij het aantal verschillende argumenten in de loop eindig is. Hier is een ander voorbeeld van een functie die een stackoverflow geeft.

```
1    let corec[iterator(0)] a x -> a (x+1)
```

Het nieuwe argument is hier constant een groter dan het vorige argument. Bij standaard instellingen geeft dit een stackoverflow. Maar als het programma genoeg ruimte krijgt toegewezen, kan het argument groot genoeg worden om de opslagruimte voor een int te overschrijden en negatief te worden, waarna het verder gaat met optellen. Zo kan het uiteindelijk toch een argument gebruiken dat het al eerder heeft gebruikt. Bij genoeg ruimte en tijd kan dit dus een antwoord geven, hoewel dat niet zou moeten. Dit is geen bug, maar de manier waarop ints werken.

De iterator stopt zodra bij een argument dat al eerder is gebruikt de solveruitkomst hetzelfde is als de solveruitkomst bij de vorige keer dat dit argument werdt gebruikt

Zie de volgende functie:

```
1    let corec[iterator(true)] choice t = match t with
2    | [] -> true
3    | x::y -> not choice y
```

Gegeven een lijst geeft deze functie true als de lengte van de lijst even is en false als het oneven is.

```
1    let one = [ 1 ]
2    let two = [ 1; 2 ]
3    let rec ones = 1::ones
4    let rec alt = 1::2::alt
5    let c2 = choice one
6    let c3 = choice two
7    let c4 = choice ones
8    let c5 = choice alt
```

geeft:

```
1   val one : int list = [1]
2   val two : int list = [1; 2]
3   val ones : int list = [1; 1; 1; 1; ...]
4   val alt : int list = [1; 2; 1; 2; 1; 2; 1; 2; ...]
5   val c2 : bool = false
6   val c3 : bool = true
```

```
stackoverflow
val c5 : bool = true
```

Dus waarom geeft choice ones stackoverflow terwijl choice alt dat niet doet? Hieronder is voor beide functies een stappentabel. De stappentabel van choice ones is bij stap 5 afgebroken, omdat de berekening oneindig doorgaat. De stappentabel van choice alt stopt bij stap 3, net als de solver.

Aanroep is vervangen bij stap	Dit geeft	Solveruitkomst
1 choice ones	true	true
2 **not** choice ones	**not** true	false
3 **not not** choice ones	**not not** true	true
4 **not not not** choice ones	**not not not** true	false
5 ...		

Aanroep is vervangen bij stap	Dit geeft	Solveruitkomst
1 choice alt	true	true
2 **not** choice 2:: alt	**not** true	false
3 **not not** choice alt	**not not** true	true

De iterator stopt pas als de solveruitkomst gelijk is aan de solveruitkomst toen het argument de vorige keer werdt gebruikt. Als we kijken naar stap 2 van choice ones, zien we dat de solveruitkomst false is en de solveruitkomst toen het argument de vorige keer werdt gebruikt true is in stap 1. In stap 3 is het weer true en in stap 4 weer false, etc. Er is geen een stap waar de solveruitkomst gelijk is aan de vorige solveruitkomst.

Als we kijken naar choice alt, zien we dat bij stap 3 dat de solveruitkomst gelijk is aan de vorige keer dat het argument alt werd gebruikt, namelijk in stap 1. De iterator stopt hier en geeft true als uitkomst.

We hebben hier dus een functie die true geeft als het recursieve deel van een oneindige lijst even is en een stackoverflow als het oneven is. Dit is niet bepaald een nuttige functie, maar de uitkomsten zijn wel logisch, als we een oneindige lijst zien als een steeds groeiende lijst. choice ones geeft namelijk zowel true als false als uitkomst naarmate het groeit. Bij choice alt is de enige uitkomst altijd false na elke groei. ones is gedefinieerd als **let** ones = 1::ones, dus deze groeit elke keer met één element. alt is gedefinieerd als **let** alt = 1::2:: alt dus deze groeit elke keer met twee elementen.

Er moet opgemerkt worden dat **let** ones = 1::1:: ones automatisch door de solver wordt vereenvoudigd tot **let** ones = 1::ones, dus dit geeft nog steeds een stackoverflow.

Je kunt met de iterator checken of een datastructuur oneindig is
Zie de functie is_finite hieronder, die true geeft als de lijst eindig is en false

als die oneindig is.

```
let corec[iterator(false)] is_finite l = match l with
| [ ] -> true
| h :: t -> is_finite t
```

De iterator kan het element geven waar een loop in een datastructuur begint

Zie de functie hieronder. De input is 0 en een oneindige int list. De uitkomst is het element waar de loop begint in de int list.

```
let corec[iterator(0)] get_loop_element arg = match arg with
| last, el::rest -> -last + el + (get_loop_element (el,rest))

let rec ones = 1::ones
let a1 = get_loop_element (0,ones)
let rec alt3 = 1::2::3::alt3
let a2 = get_loop_element(0,alt3)
let rec alt3 = 1::2::3::alt3
let fivesalt3 = 5::5::alt3
let a3 = get_loop_element(0,aa)

val ones = [1; 1; 1; 1; ... ]
let a1 = 1
val alt3 = [1; 2; 3; 1; 2; 3; ... ]
let a2 = 1
val fivesalt3 = [5; 5; 1; 2; 3; 1; 2; 3; ... ]
let a3 = 1
```

Aangezien we het laatste element willen voordat de iterator de functie afbreekt, bestaat de uitkomst uit het laatste element. Om te zorgen dat we alleen maar het laatste element als antwoord krijgen, is het solverantwoord 0 en trekken we het vorige element van de output af.

Merk op dat het onmogelijk is om de index van het laatste element te geven. Dan zou de uitkomst namelijk per element verschillen en kunnen we geen loop vinden. Hier is de iterator simpelweg niet voor bedoeld.

De iterator kan loops afbreken

We hebben hier vooral voorbeelden bekeken die om kunnen gaan met oneindige datastructuren en dit is ook waar de iterator het intuïtiefst is. Maar de toepassingen van de iterator zijn breder. Het kan namelijk loops afbreken, ook als daar geen oneindige datastructuren voor aan de pas komen. Zie de functie testje. Deze functie doet niet veel. Het verhoogt het gegeven argument met een. Maar het gebruikt daarbij wel een berekening die niet zou kunnen in een recursieve functie.

```
let rec aux l1 l2 = match (l1,l2) with
| 0, l2 -> (l2+1)
| l1, l2 -> if (l1-1)=l2 then l1 else 5

let corec[iterator(0)] testje t = aux (testje t) t
```

aux is hierbij een functie die, als het 0 en een ander getal krijgt, het andere getal met een verhoogt, en als het twee getallen krijgten het eerste getal is een hoger dan het tweede, het dat eerste getal teruggeeft. In enige andere situatie geeft het 5. testje roept zichzelf recursief aan, zodat, gegeven het getal 2, de volgende stappen gebeuren:

```
testje 2
aux (testje 2) 2
aux (aux (testje 2) 2) 2
aux (aux (aux (testje 2) 2) 2) 2
aux (aux (aux (aux (testje 2) 2) 2) 2) 2
...
```

De recursieve versie van deze functie zou een stackoverflow geven. Maar de iterator herkent de loop, vervangt de aanroep door het solverantwoord en geeft 3 als uitkomst. In sectie 3.2 , Ai.ml, zie je een voorbeeld van hoe dit nuttig gebruikt kan worden.

2.2 De Constructor Solver

Met de constructor solver kun je corecursieve functies maken die regulaire oneindige datastructuren kunnen maken. Zulke corecursieve functies zien er zo uit:

```
let corec[constructor] functienaam argument =
      functionbody
```

Ik laat eerst twee voorbeelden zien en leg daarna pas uit hoe de constructor werkt.

De simpelste constructor functie is de volgende functie. Gegeven een argument maakt het een oneindige lijst gevuld met dat argument.

```
let corec[constructor] make_list arg = arg :: make_list arg
```

Als je een constructor functie een lijst meegeeft, kan het die veranderen. Hier is bijvoorbeeld de map functie gemaakt met de constructor. De map functie past een gegeven functie toe op elk element in de gegeven lijst.

```
  let corec[constructor] map arg = match arg with
  | f, [ ] -> [ ]
  | f, h :: t -> (f h) :: map(f,t)
```

Waarbij

```
  let test1 = function
  | 1 -> 5
  | x -> x

  let mt1 = map (test1, zeros)
  let mt2 = map (test1, alt)
```

het volgende als antwoord geeft:

```
  val mt1 : int list = [0; 0; 0; 0; ...]
  val mt2 : int list = [5; 2; 5; 2; 5; 2; 5; 2; ...]
```

De constructor heeft geen solverantwoord
Eindige lijsten hebben een begin nodig om op te bouwen. Oneindige lijsten hebben dit niet nodig. Die bouwen namelijk op zichzelf. Een voorbeeld:

> De eindige lijst [0; 0; 0] is gebouwd op [ ]. [0; 0; 0] staat namelijk voor 0:: 0:: 0:: [ ], oftewel "een 0 voor een 0 voor een 0 voor een lege lijst".
>
> De oneindige lijst zeros is gebouwd op zichzelf. zeros staat namelijk voor 0::zeros, oftewel "een 0 voor zeros".

Als je een functie hebt die, gegeven een eindige datastructuur, een andere eindige datastructuur teruggeeft, heeft die andere eindige datastructuur een

begin nodig om op te bouwen. De functie heeft dus een base stelling nodig. Dit geldt voor elke niet-corecursieve functie.

Als je een functie hebt die, gegeven een oneindige datastructuur, een eindige datastructuur teruggeeft, heeft ook die eindige datastructuur een begin nodig. Maar een base stelling helpt hier vaak niet, aangezien de oneindige datastructuur geen begin heeft. Hier heb je dus een extra begin nodig. Dit is de reden waarom de iterator een solverantwoord nodig heeft.

Als je een functie hebt die een oneindige datastructuur teruggeeft, heb je geen begin nodig. Daarom heeft de constructor geen solverantwoord nodig en, als je een functie maakt die alleen oneindige structuren teruggeeft, ook geen base stelling. Zo bestaat de functionbody van make_list alleen uit een recursive stelling.

Hoe werkt de constructor?

Zie de functie replace, die gegeven een oneindige lijst en twee elementen, elk voorkomen van het eerste element in de lijst vervangt door het tweede element.

```
1  let corec[constructor] replace arg = match arg with
2  | x,y,h::t -> (if h=x then y else h)::(replace (x,y,t))
3
4  let a1= replace (1,5,alt)
5  val a1 : int list = [5; 2; 5; 2; 5; 2; 5; 2; ...]
```

De voorwaarden voor het breken van loops voor de iterator zijn:

- het argument is al gebruikt, en
- de solveruitkomst is gelijk aan de solveruitkomst toen het argument de vorige keer werd gebruikt.

De voorwaarden voor de constructor zijn simpeler:

- het argument is al gebruikt.

Bij a1 ziet de stappentabel er zo uit:

Stap	Argument
1 replace (1,5, alt)	(1,5, alt)
2 5::replace (1,5,2:: alt)	(1,5,2:: alt)
3 5::2:: replace (1,5, alt)	(1,5, alt)
4 a1 = 5::2::a1	

In stap 3 wordt het argument (1,5, alt) voor de tweede keer gebruikt en wordt de loop afgebroken. De constructor bekijkt dan hoe de datastructuur gegroeid is en maakt hiervan een oneindige variant. Bij a1 is er een 5 en een 2 voor de aanroep gegroeid. De oneindige variant is dan ook een oneindige lijst van vijven en tweeën.

Hieronder volgen verschillende paragrafen over wat de constructor solver kan.

De constructor kan elke regulaire oneindige datastructuur maken
Hier is de functie a die de datastructuur structure = [structure;arg] maakt, gegeven een argument arg:

```
let corec[constructor] a arg= [a arg;arg]

let a1 = a [1];;
val a1 : int list list = [[[[[ ... ; ( ... )]; ( ... )]; ( ... )
    ]; ( ... )]; ( ... )]
```

Zoals wel vaker wordt de datastructuur wat vreemd weergegeven. Dit heeft geen invloed op de datastructuur zelf. Een logischere weergave zou zijn:

```
val a1 : int list list = [[[[[ ... ; [1]]; [1]]; [1]]; [1]];
    [1]]
```

Hier is de functie a die gegeven een een eindige lijst een oneindige lijst maakt met het argument als recursieve deel.

```
let corec[constructor] aux arg= match arg with
| x::y, z -> x::(aux (y,z))
| [ ], z -> aux (z,z)

let a arg = aux (arg,arg)
```

De constructor heeft een oneindige loop nodig om een oneindige lijst te kunnen maken. Daarom krijgt de functie aux zijn argumentlijst dubbel. Zo kan het, wanneer de argumentlijst op is, de lijst verdubbelen en verder gaan.

```
let a1 = a [1;2;3]
val a1 : int list = [1; 2; 3; 1; 2; 3; 1; 2; 3; ... ]
```

De constructor is niet geschikt om 1 waarde in een datastructuur te veranderen
Stel je hebt de oneindige lijst zeros en je wilt het 100ste element veranderen in een 1. Dit kan met de constructor. De bijbehorende functie ziet er zo uit.

```
let corec[constructor] replace_xth_element arg = match arg with
| 0, replacement, h::t -> replacement::t
| x, replacement, [] -> []
| x, replacement, h::t -> h::(replace_xth_element ((x-1),
    replacement,t))

let rxe=replace_xth_element (100,1,zeros)
val rxe : int list = [0; 0; 0; 0; ...]
```

Het eerste gegeven getal is op welke plek in de lijst je het tweede gegeven getal wil. Dit eerst gegeven getal telt af tot 0 naarmate we verder in de lijst komen. Op plek 100 is het 0 en wordt het 100ste element vervangen.

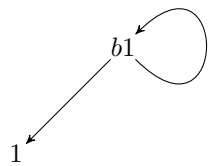
De benodigde rekentijd voor rxe is zeer lang. Dit is omdat de constructor telkens probeert een loop te vinden en hiervoor steeds de nieuwe argumenten vergelijkt met eerdere argumenten. De recursieve versie is hier stukken sneller:

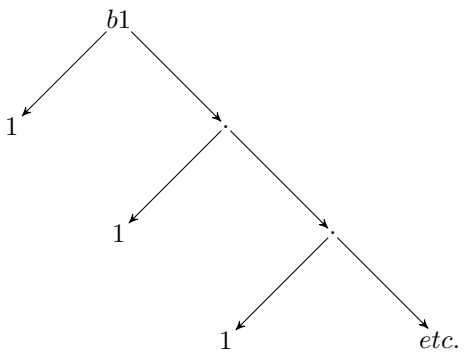
```
let rec rep_xth_element arg = match arg with
| 0, replacement, h::t -> replacement::t
| x, replacement, [] -> []
| x, replacement, h::t -> h::(rep_xth_element ((x-1),replacement
    ,t))
```

Het enige verschil tussen beide functies is dat corec[constructor] is veranderd in **rec** en dat de rekentijd korter is. De uitkomst is bij beide functies exact hetzelfde: een lijst van 99 nullen met daarna een 1 met daarna zeros. Dus het is aan te raden om altijd een recursieve functie te gebruiken als je een eindige verandering wil toepassen.

Met de constructor kun je oneindige regulaire bomen maken

Zulke bomen worden ook wel graphs genoemd. Stel we hebben de boom **let rec** b1 = Branch (Leaf 1, b1). Dan kun je deze boom op twee manieren tekenen: Met weergegeven loop en uitgeschreven zonder loop.





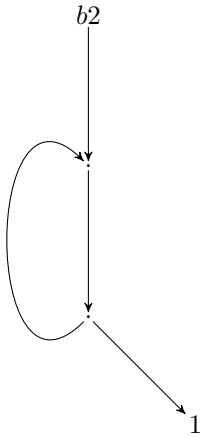
De eerste weergave is (naar mijn mening) verreweg het overzichtelijkst. Stel je wilt een boom met een loop maken, maar weet pas at run-time hoe die boom er uit moet zien. Dan zul je deze boom met de constructor moeten maken. Ik zocht naar een manier om dit makkelijk te kunnen en het volgende was mijn eerste opzet. Om duidelijk te maken hoe je je boom wilt, maak je een eindige versie waarin je aangeeft waar het begin en het einde van je loop wil. Deze begin en eind noem ik respectievelijk de arrowtail en de arrowhead van de loop. Dit kun je specificeren met het volgende datatype.

```
type ltree = Leaf of int | ArrowTail | Branch of bool*ltree |
    Branch2 of bool*ltree*ltree
```

We maken hier onderscheid tussen twee soorten branches: branches met een tak en branches met twee takken. Elke branch heeft een bool. Staat deze bool op true, dan betekent dit dat er naar deze branch een arrowhead wijst. De boom b1 zou er in dit type zo beschreven worden:

```
let b1 = Branch2(true, Leaf(1),ArrowTail)
```

Hieronder een tekening van een andere boom en de bijbehorende beschrijving:




```
let b2 = Branch(false, Branch(true, Branch2(false, ArrowTail,
    Leaf(1))))
```

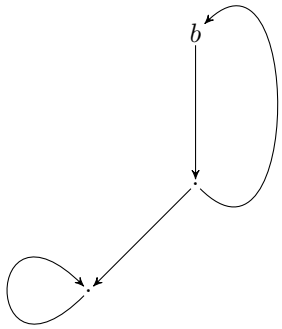
Met deze beschrijving hebben we genoeg informatie om de echte boom te maken. Dat gebeurt met de volgende constructor functie:

```
type ctree = Cbranch of ctree | Cbranch2 of ctree*ctree | Cleaf
    of int

let corec[constructor] totree arg = match arg with
| Leaf(x), h -> Cleaf(x)
| ArrowTail, h -> totree (h,h)
| Branch (b,t), h -> Cbranch(totree (t, if b then Branch(b,t)
    else h))
| Branch2 (b,t1,t2), h -> Cbranch2(totree (t1, if b then Branch2
    (b,t1,t2) else h), totree(t1, if b then Branch2(b,t1,t2) else
     h))

let cb = totree (cb, Leaf(0));;
val cb : ctree = Cbranch (Cbranch (Cbranch2 (Cbranch (Cbranch2 (
    Cbranch (Cbranch2 (Cbranch ... ), ... )), ... )), ... )))
```

Deze functie krijgt twee argumenten in een tupel. Het eerste argument is de boombeschrijving, het tweede is een leaf. Zou er een arrowhead komen zonder arrowtail, dan komt er in plaats van dat er een loop gemaakt wordt, een leaf. Komt er wel een arrowtail, dan wordt het tweede argument vervangen

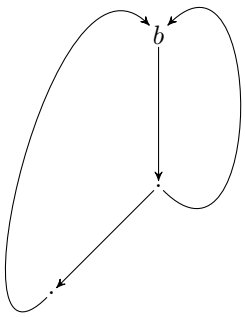
door die branch in de boom, zodat als de functie bij de arrowhead komt, we de plek hebben waar de loop naar wijst. Deze functie heeft voordelen en nadelen. Zo zijn de volgende bomen wel mogelijk (met voorbeeld):

- bomen met meerdere loops.




```
let b = totree(Branch(true,Branch2(false, Branch(true,
    ArrowTail),ArrowTail)), Leaf(0))
```

- bomen waarvan meerdere loops naar dezelfde branch wijzen.

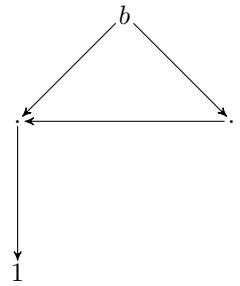



```
let b = totree(Branch(true, Branch2(false,Branch(false,
    ArrowTail),ArrowTail)),Leaf(0))
```

De volgende dingen zijn niet mogelijk:

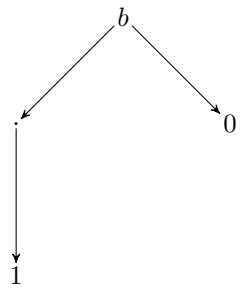
- bomen met loops die omlaag of naar een andere tak wijzen. Dit zijn eigenlijk geen loops, maar ze lijken wel definieerbaar met het type ltree. Als je de boom hieronder probeert te definieren krijg je op de plek van de Branch(false,ArrowTail) een Leaf(0).

Een ArrowTail kan alleen naar een ArrowHead wijzen als die zich boven de ArrowTail bevindt. Anders komt er standaard een Leaf(0).

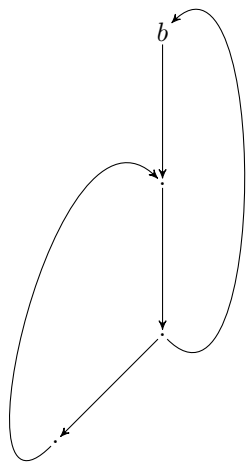



```
let b = totree(Branch2(false,Branch(true,Leaf(1)),Branch(
    false,ArrowTail)),Leaf(0))
```

De bijbehorende Ctree ziet er dus zo uit:



– bomen waarbij er een arrowhead van een loop tussen de arrowhead en arrowtail van een andere loop ligt (onafhankelijk van waar de arrowtail van de eerste loop ligt)



In dit voorbeeld zal de loop die wijst naar b overschreven worden door de loop daaronder, aangezien onze totree functie maar 1 loop kan onthouden. De rechtse loop zal dus wijzen naar de node onder b.

Meer hierover in sectie 3.5 , Graphs.

Manieren waarop de constructor niet gebruikt moet worden

We bekijken wat mogelijke recursive antwoorden voor een functie a die een lijst krijgt als argument:

```
| h::t -> h::(a t)
| h::t -> (aux h)::(a t)
| h::t -> h::(aux (a t))
```

De corecursieve functie probeert altijd een loop te vinden. Een constructor functie, als het er een gevonden heeft, probeert de groei van de datastructuur vast te stellen en maakt daaruit de gewilde oneindige datastructuur. Om deze groei te kunnen vaststellen is het nodig dat alles in een antwoord dat niet de aanroep is, compleet is uitgerekend. Dit is de reden dat van de drie hierboven beschreven antwoorden de laatste voor een foutmelding zal zorgen, als die aangeroepen wordt in een constructor functie. aux ligt namelijk buiten de aanroep, maar heeft de aanroep wel nodig. Dit zorgt ervoor dat de groei niet berekenbaar is totdat de aanroep is berekend, en de aanroep niet berekenbaar is totdat de groei is berekend.

Functies waarbij de groei niet berekenbaar is, geven een foutmelding nadat het argument gegeven is. Deze foutmelding ziet er zo uit:

```
Fatal error: exception Failure("unknown2var")
```

unknown2var is de nog ongenaamde variabele waar de structuur recursief op wordt gebouwd. Een voorbeeld: stel we willen een functie plusone die gegeven een oneindige int list elke int verhoogt met 1. De goede manier om dit te maken is:

```
let corec[constructor] plusone arg = match arg with
| h::t -> (h+1)::plusone t
```

Hier zijn groei en aanroep gescheiden. Een manier waarop ze niet gescheiden zijn is deze:

```
let aux arg = match arg with
| (h,t) -> (h+1)::t

let corec[constructor] plusone arg = match arg with
| h::t -> aux(h,plusone t)
```

Hierbij is het eerst nodig dat we plusone t uitrekenen, voordat we aux aan kunnen roepen. Maar plusone t is nog niks totdat de constructor de groei heeft gevonden, en die kan hij niet vinden totdat het aux heeft aangeroepen. Deze functie geeft dus de foutmelding. Constructor functies waarbij de recursieve aanroep in het if-gedeelte van een if-statement staat kunnen

dus ook niet. De if-statement kan niet uitgerekend worden totdat de groei gevonden is, en die wordt niet gevonden totdat de if-statement wordt uitgerekend. Dit geldt ook als de then-else gedeeltes beide dezelfde groei zouden geven.

Een constructor functie zonder argumenten is geen datastructuur
Dit is een wat vreemde bug. Zie de volgende functie.

```
let corec[constructor] a = 2::a
```

Dit zou de lijst [2; 2; 2; 2; ...] moeten geven. In plaats daarvan rekent de functie niets uit en is het van het type **fun**. Hier is makkelijk rond om heen te werken. Als je een functie zonder argumenten wil maken, maak je de recursieve variant. Deze werkt in alle gevallen.

Je kunt geen datastructuren van functies maken met de constructor
Wil je een oneindige lijst van functies maken, genaamd lijs , dan kun je dat zo doen:

```
let test1 arg = match arg with
| 0 -> 1
| x -> x

let rec lijs= test1::lijs
val lijs : int -> int list = [(fun arg -> (match arg with 0 -> 1
    | x -> x)); lijs]
```

Dit geeft een oneindige lijst van de functie test1. Maar probeer je dit met de constructor (bijvoorbeeld met make_list test1), dan krijg je de unknow2var-foutmelding. De reden hiervoor is dat de constructor in een recursive stelling pas de recursieve aanroep uitrekent als al het andere al uitgerekend is. Dus pas als er geen functies meer in het antwoord voorkomen en alleen maar declaraties. In dit specifieke geval is de constructor hier te ijverig in, aangezien het hier ons doel is dat er een functie overblijft. Dit is een bug.

Als er geen datastructuur groeit, geeft de constructor dots terug
Zie de volgende functies:

```
      let corec[constructor] make_dots arg = match arg with
      | x -> dots x

      let dots1= make_dots zeros

      let corec[constructor] make_dots arg = match arg with
      | x::y -> dots y

      let dots2= make_dots zeros
```

Deze geven de volgende uitkomst:

```
val dots1 : 'a =  ...
val dots2 : 'a =  ...
```

Deze uitkomst noem ik dots. Ze hebben geen eigenschappen en het meest generieke type. Dit kun je zien aan het feit dat het met het type 'a worden weergegeven. Waarom wordt deze uitkomst gegeven? Er groeit hier geen datastructuur. Dus er is niets voor de constructor om op te bouwen. Dus het geeft de meest generieke uitkomst.

Dots heeft een paar vreemde eigenschappen. Zo kan er elk element aan dots worden toegevoegd alsof het een lijst is:

```
let a = "test"::dots1
val a : string list = ["henk"; ...]
```

Dit verandert verder niets aan dots. Ook geeft dots bij de vraag dots1=object voor elk object van elk type als antwoord true. Dit terwijl het dat object niet is.

Dots moet niet verward worden met de puntjes die standaard worden weergegeven aan het eind van de weergave van een oneindige datastructuur. Dots is een object. De puntjes zijn om aan te geven dat de lijst langer is dan weergegeven.

2.3 Wat COcaml kan samengevat

Hier is een overzicht van wat er mogelijk is in COcaml en welke soort functie er geschikt voor is.

- oneindige regulaire datastructuren maken.

  Dit kan hard coded met recursieve declaraties en gedurende het programma met de constructor.

- elementen op een bepaalde plek in een datastructuur opvragen.

  Recursieve functies zijn hiervoor het meest geschikt (lees: snelst), maar iterator en constructor functies kunnen ook.

- elementen op een bepaalde plek in een datastructuur wijzigen.

  Recursieve functies zijn hiervoor het meest geschikt (lees: snelst), maar iterator en constructor functies kunnen ook.

- opvragen of een kenmerk geldt voor een mogelijk oneindig aantal elementen in een datastructuur

  Hiervoor is de iterator.

- een mogelijk oneindig aantal elementen in een datastructuur wijzigen

  Hiervoor is de constructor.

2.4 De appears solver en zelfgemaakte solvers

De appears solver

De appears solver is een simpele solver. Appears functies kunnen alleen een boolean als uitkomst hebben. Zo'n functie ziet er als volgt uit:

```
let corec[appears(schatting)] functienaam argument =
        functionbody
```

Als de functionbody als uitkomst de schatting geeft, dan geeft de corecursieve functie true. Geeft de functionbody een andere uitkomst of komt het in een oneindige loop, dan geeft de corecursieve functie false.

Een simpel voorbeeld is de corecursieve functie is_finite , die kijkt of een lijst eindig is.

```
let corec[appears(2)] is_finite l = match l with
| [ ] -> 2
| h :: t -> is_finite t

let a1= is_finite [1;2]
let zeros = 0:: zeros
let a2 = is_finite zeros

val a1 : bool = true
val a2 : bool = false
```

Is de gegeven lijst eindig, dan komt de functionbody uiteindelijk bij de eerste stelling. Vervolgens geeft het 2 als uitkomst aan de solver, waarop de solver als uitkomst true geeft. Is de gegeven lijst niet eindig, dan komt de functionbody in een oneindige loop, de solver herkent dit en geeft false.

Merk hier op dat het niet uitmaakt wat de schatting is, zolang de uitkomst bij een lege lijst gelijk is aan het schatting. De schatting is hier 2, maar de corecursieve functie doet exact hetzelfde als de schatting "foo" is, zolang de 2 in de functionbody dan ook "foo" is.

Voor elke functie die te maken is met de appears solver is er een iterator functie te maken die hetzelfde doet.

Zelfgemaakte solvers

COcaml geeft de mogelijkheid om je eigen solver te schrijven. Dit is praktisch als je een zeer specifieke solver nodig hebt. Hoe zo'n solver gemaakt wordt ligt buiten de scope van deze scriptie. Op het moment zijn er in de voorbeeldfiles drie voorbeelden:

- De unit solver, die op alles de lege unit teruggeeft.
- De separated solver, waarmee de functie dseparate te maken is die, gegeven een regulaire lijst, het eindige deel van het recursieve deel scheidt. Dit is het best uitgelegd met een aantal voorbeelden:

```
let rec alt3 = 1 :: 2 :: 3 :: alt3
let r1 = dseparate [1;2]
let r2 = dseparate alt3
```

```
let r3 = dseparate (1 :: 2 :: alt3)
```

geeft:

```
val r1 : ('a list * 'a list) = [1; 2], [ ]
val r2 : ('a list * 'a list) = [ ], [1; 2; 3]
val r3 : ('a list * 'a list) = [1; 2], [1; 2; 3]
```

- De gaussian solver, die bepaalde limieten kan berekenen. Deze solver heeft zeer specifieke toepassingen die ik verder uitleg in sectie 3.4 , Probability.ml.

Hoofdstuk 3

Voorbeelden

Deze versie van COcaml komt samen met een aantal voorbeeldfiles. Sommigen hiervan geven mogelijke toepassingen en implementaties. Andere zijn gewoon testfiles. Ik beschrijf in dit hoofdstuk alle nuttige files. Verder beschrijf ik twee eigen uitwerkingen van mogelijke toepassingen. Deze staan in de Appendix.

3.1 List module functies voor oneindige regulaire datatypen: list.ml

Deze file bevat verschillende functies die in de welbekende List module voor Ocaml voorkomen, maar dan aangepast voor oneindige regulaire datatypen.

Ik geef eerst de volgende definities:

- **let** one = [1]
- **let** two = [1; 2]
- **let rec** zeros = 0 :: zeros
- **let rec** alt = 1 :: 2 :: alt
- *f* is een functie
- *l*, *l1* en *l2* zijn lijsten
- *el* is een element van een lijst
- *a* is een object

De voorkomende List functies zijn:

Map (*f*, *l*), vervangt elk element *el* in de lijst *l* met *f el*, gemaakt met de constructor.

```
1   let corec[constructor] map arg = match arg with
2   | f, [ ] -> [ ]
3   | f, h :: t -> (f h) :: map(f,t)
4
5   let r4 = map ((fun i -> i+1), one)
6   let r5 = map ((fun i -> i+1), two)
7   let r6 = map ((fun i -> i+1), zeros)
```

```
val r4 : int list = [2]
val r5 : int list = [2; 3]
val r6 : int list = [1; 1; 1; 1; ... ]
```

Je kunt met deze functie natuurlijk geen lijst van functies maken, wat met de Ocaml map functie wel kan.

Exists (*f*, *l*), geeft true als er een *el* in de lijst *l* is waarvoor *f el* is true. Deze functie is zowel met de appears solver als met de iterator solver te maken.

```
let corec[appears(true)] exists_appears arg = match arg
    with
| f, [ ] -> false
| f, h :: t -> f h || exists_appears (f, t)

let corec[iterator(false)] exists arg = match arg with
| f, [ ] -> false
| f, h :: t -> f h || exists (f, t)

let r10 = exists ((fun i -> i = 1), one)
let r11 = exists ((fun i -> i = 0), two)
let r12 = exists ((fun i -> i = 2), alt)
let r13 = exists ((fun i -> i = 1), zeros)

val r10 : bool = true
val r11 : bool = false
val r12 : bool = true
val r13 : bool = false
```

De appears solver geeft true als de functionbody true geeft en false in elke andere situatie. De functionbody geeft alleen true als er een element is dat voldoet aan *f*.

De iterator geeft false als solverantwoord bij een oneindige loop. Het geeft dus alleen true als uitkomst als er al een element was dat true heeft gegeven.

Beide functies stoppen zodra ze een element hebben gevonden dat voldoet aan *f*.

For_all (*f*, *l*), geeft true als *f el* true geeft voor elk element *el* in de lijst *l*. Deze functie is zowel met de appears solver als met de iterator solver te maken.

```
let corec[appears(false)] for_all_aux arg = match arg
    with
| f, [ ] -> true
| f, h :: t -> f h && for_all_aux (f, t)
let for_all arg = not(for_all_aux arg)

let corec[iterator(true)] for_all arg = match arg with
```

```
| f, [ ] -> true
| f, h :: t -> f h && for_all (f, t)

let r7 = for_all ((fun i -> i = 1), one)
let r8 = for_all ((fun i -> i = 1), two)
let r9 = for_all ((fun i -> i = 1), alt)

val r7 : bool = true
val r8 : bool = false
val r9 : bool = false
```

De appears solver geeft true als de functionbody de solveruitkomst geeft. In dit geval moet de solveruitkomst false zijn. Dat is alleen het geval als er een element is dat niet aan de test *f* voldoet. Dus dan geeft for_all_aux true en for_all false.

De iterator solver herkent een oneindige loop als die er is. In dit geval geeft het false als er al een element false heeft gegeven, en anders geeft het true.

Beide functies stoppen zodra een element niet voldoet aan *f*.

Mem (*a*, *l*), geeft true als *a* voorkomt in *l*, gemaakt met de appears zowel als de iterator.

```
let corec[appears(true)] mem arg = match arg with
| e, [ ] -> false
| e, h :: t -> e = h || mem (e, t)

let corec[iterator(false)] mem arg = match arg with
| e, [ ] -> false
| e, h :: t -> e = h || mem (e, t)

let r15 = mem (1, one)
let r16 = mem (0, two)
let r17 = mem (1, alt)
let r18 = mem (1, zeros)
let r19 = mem (2, alt)

val r15 : bool = true
val r16 : bool = false
val r17 : bool = true
val r18 : bool = false
val r19 : bool = true
```

Filter (*f*, *l*), geeft een lijst van elementen *el* in lijst *l* waarvoor *f el* is true, gemaakt met de constructor.

```
let corec[constructor] filter arg = match arg with
| f, [ ] -> [ ]
| f, h :: t ->
  if f h then h :: filter(f, t)
  else if exists(f, t) then filter(f, t) else [ ]
```

```
6
7     let r36 = filter ((fun i -> i = 1), alt)
8     let r37 = filter ((fun i -> i < 1), alt)
9
10    val r36 : int list = [1; 1; 1; ...]
11    val r37 : int list = [ ]
```

Deze functie maakt gebruik van de eerder beschreven exists functie in de tweede stelling. Zou de tweede stelling namelijk als volgt gedefinieerd zijn, dan zou het dots als uitkomst geven als er geen enkel element aan f voldoet, in plaats van de gewilde lege lijst:

```
1   | f, h :: t ->
2    if f h then h :: filter_naive(f, t)
3    else filter_naive(f, t)
```

De constructor, zodra het een loop ontdekt, probeert de groei te vinden en als die er niet is geeft het dots. Om dit tegen te gaan wordt er zodra een element niet voldoet aan f gekeken of er een element in de lijst bestaat die wel voldoet aan f.

Het lijkt alsof de volgende oplossing ook zou kunnen werken. Dat is niet het geval.

```
1       let corec[constructor] filter_aux arg = match arg
            with
2       | f, [ ] -> [ ]
3       | f, h :: t ->
4         if f h then h :: filter_aux(f, t)
5         else filter_aux(f, t)
6
7       let filter f l = if exists (f,l) then filter_aux (f,l
            ) else []
8
9       let r36 = filter ((fun i -> i = 1), alt)
10      let r37 = filter ((fun i -> i < 1), alt)
11      let r38 = filter ((fun i -> i < 1), 0:: alt)
12
13      val r36 : int list = [1; 1; 1; 1; ...]
14      val r37 : int list = [ ]
15      val r38 : int list = [1; ...]
```

Het grootste verschil tussen deze twee functies is dat de eerste de functie exists aanroept elke keer als er een element niet voldoet aan f en er in de tail van de lijst er nog wel een element voldoet, en de tweede functie maar een keer exists aanroept. De puntjes in r38 zijn niet de puntjes die aangeven dat een lijst oneindig is, maar dots. Optimaal zou zijn als het mogelijk was om **if not** exists(f, t) **then** [ ] aan te roepen bij de grens van het eindige deel met het recursieve deel van de lijst en voor de rest exists nooit aan te roepen. Maar aangezien de constructor deze grens niet weet, moet exists vaker aangeroepen worden.

Remove (*a*, *l*), verwijdert elk voorkomen van *a* in *l*, gemaakt met de constructor.

```
let corec[constructor] remove arg = match arg with
| e, [ ] -> [ ]
| e, h :: t ->
    if e = h then
      if exists((fun e1 -> e <> e1), t) then remove(e,
          t) else [ ]
    else h :: remove(e, t)

let r40 = remove(1, alt)
let r41 = remove(1, ones)

val r40 : int list = [2; 2; 2; 2; ...]
val r41 : int list = []
```

Ook deze functie roept veelvoudig exists aan om dots tegen te gaan.

Append (*l1*, *l2*), voegt de twee lijsten samen, gemaakt met de constructor.

```
let corec[constructor] append arg = match arg with
| [ ], l2 -> l2
| h :: t, l2 -> h :: append(t, l2)

let r42 = append (one, two)
let r43 = append (two, ones)
let r44 = append (zeros, ones)
let h44 = remove (0,r44)
let h44 = remove (1,r44)

val r42 : int list = [1; 1; 2]
val r43 : int list = [1; 2; 1; 1; 1; 1; ...]
val r44 : int list = [0; 0; 0; 0;  ...]
val h44 : int list = [ ]
val h44 : int list = [0; 0; 0; 0; ...]
```

Hier moet opgemerkt worden dat wiskundig gezien iets toevoegen aan het eind van een oneindige lijst niet mogelijk is. Hetgene dat toegevoegd wordt verdwijnt dan bij deze functie. Oftewel: r44 is gelijk aan ones.

combine (*l1*, *l2*), maakt van een paar van lijsten een lijst van paren, gemaakt met de constructor. Net zoals List.combine geeft deze functie een foutmelding als de lijsten niet even lang zijn.

```
let corec[constructor] map2 arg = match arg with
| f, [ ], [ ] -> [ ]
| f, h1 :: t1, h2 :: t2 -> f h1 h2 :: map2 (f, t1, t2)

let combine l1 l2 = map2 ((fun i j -> i, j), l1, l2)

```

```
let r45 = combine zeros ones
let r46 = combine zeros alt
let r47 = combine alt alt3

val r45 : (int * int) list = [(0, 1); (0, 1); (0, 1); (0,
    1); ...]
val r46 : (int * int) list = [(0, 1); (0, 2); (0, 1); (0,
    2); ...]
val r47 : (int * int) list = [(1, 1); (2, 2); (1, 3); (2,
    1); ...]
```

map2 is hier een functie die, voor elke elementen *el1* en *el2* in de lijsten *l1* en *l2*, *f el1 el2* teruggeeft.

Set *l*, maakt een set van de voorkomende elementen. In tegenstelling tot de eerder beschreven toset op pagina 10 zijn de elementen hier wel in volgorde van grootte. Gemaakt met de iterator.

```
let rec insert i l = match l with
| [] -> [i]
| h :: t ->
  if i < h then i :: h :: t
  else if i > h then h :: insert i t
  else h :: t

let corec[iterator([])] set l = match l with
  [] -> []
| h :: t -> insert h (set t)

let r55 = set one
let r56 = set two
let r57 = set zeros
let r58 = set ones
let r59 = set alt
let r60 = set alt3

val r55 : int list = [1]
val r56 : int list = [1; 2]
val r57 : int list = [0]
val r58 : int list = [1]
val r59 : int list = [1; 2]
val r60 : int list = [1; 2; 3]
```

insert is hier een functie die een element op de juiste plek in een gesorteerde lijst stopt.

Verder zijn er nog de volgende functies die niet voorkomen in de oorspronkelijke List module:

Is_finite *l*, geeft true als de lijst eindig is, gemaakt met de appears zowel als de iterator.

```
1    let corec[appears(true)] is_finite_appears l = match l
         with
2    | [ ] -> true
3    | h :: t -> is_finite_appears t
4
5    let corec[iterator(false)] is_finite l = match l with
6    | [ ] -> true
7    | h :: t -> is_finite t
8
9    let r1 = is_finite one
10   let r2 = is_finite two
11   let r3 = is_finite zeros
12
13 val r1 : bool = true
14 val r2 : bool = true
15 val r3 : bool = false
```

Separate *l*, splitst een lijst op in het eindige en het recursieve deel, gemaakt met de separate solver.

```
1    type 'a sep = I1 | I2 of 'a * sep | I3 of 'a list * 'a
         list
2    let corec[separate] separate i = match i with
3    | [ ] -> I1
4    | i :: t -> I2(i, separate t)
5
6    let r25 = separate one
7    let r27 = separate ones
8    let r29 = separate alt
9    let r31 = separate (1 :: 2 :: alt3)
10
11   val r25 : int sep = I3 ([1], [])
12   val r27 : int sep = I3 ([], [1])
13   val r29 : int sep = I3 ([], [1; 2])
14   val r31 : int sep = I3 ([1; 2], [1; 2; 3])
```

Hoe dit precies gebeurt is mij onduidelijk en ligt buiten de scope van deze scriptie. Ik weet niet wat de complexiteit van deze functie is. Ik kan dus helaas geen uitspraken doen over in welke situaties deze functie gebruiken in andere functies optimaler is. De functie filter bijvoorbeeld zou exists maar een keer hoeven aan te roepen als het wist waar de grens tussen het eindige en het recursieve deel in de gegeven lijst ligt. Het zou dus sneller kunnen zijn als het gebruik maakte van separate, maar dat hangt af van de complexiteit.

Zip (*l1*, *l2*), maakt een lijst met om de beurt een element uit *l1*, dan uit *l2*, etc. Gemaakt met de constructor.

```
1    let corec[constructor] zip arg = match arg with
2    | [], [] -> []
3    | h1 :: t1, h2 :: t2 -> h1 :: h2 :: zip (t1, t2)
4
```

```
let r48 = zip (zeros, ones)
let r49 = zip (ones, zeros)
let r50 = zip (alt, alt3)

val r48 : int list = [0; 1; 0; 1; 0; 1; ...]
val r49 : int list = [1; 0; 1; 0; 1; 0; 1; ...]
val r50 : int list = [1; 1; 2; 2; 1; 3; ...]
```

3.2 Abstract interpretation: ai.ml

Abstract interpretation is een manier om informatie te verkrijgen over een programma zonder het uit te voeren. Deze manier werkt als volgt. In plaats van de waarden van ints bij te houden, houden we alleen bij in welke van de volgende vier states het zich bevindt: positief, negatief, nul of onbekend. Als een int gedurende het programma altijd positief is, heeft het de state positief. Zo ook voor negatief en nul. Verandert voor de int gedurende het programma of het positief, negatief of nul is, dan komt het in de state onbekend. Dit kan handig zijn bij debuggen.

Hiervan is een simpele versie gemaakt in de file ai.ml. In deze versie zijn er programma's te maken met de volgende commando's:

Skip doe niks

Assign string number geef de variabele met naam string de waarde number

Cons command command voer de commands in volgorde uit

If number command command als number niet nul is, doe het eerste command, anders het tweede command

While number command zolang number nul is, doe command

Een number is een constante, een variabele of de optelling van twee numbers. Met deze commando's heb je de basis van wat een computer kan. Verder zijn de states gedefinieerd als volgt:

```
type anumber = APos | ANeg | AZero | ATop
```

Waarbij ATop is de waarde onbekend.

Stel we hebben de volgende twee programma's.

```
let prog1 = While(Var "b", Assign("b", Int 1))
let prog2 = While(Var "b", Assign("b", Int 0))
```

Dan zijn de waardes voor b per beginwaarde:

Beginwaarde	Prog1	Prog2
Positief	Positief	Positief
Negatief	Negatief	Negatief
Nul	Onbekend	Nul

Deze waardes worden gegeven door

de functie interpret. Deze functie maakt gebruik van de iterator solver in het geval dat **while** in het programma staat. Dit omdat de iterator solver een fundamenteel probleem oplost: wat als het programma in een while loop komt. Dit is het betreffende stuk code:

```
1  let rec interpret (c:command) s = match c with
2    | Skip -> s
3    | Cons(c1, c2) -> interpret c2 (interpret c1 s)
4    | Assign(x, n) -> insert (x, interpret_number n s) s
5    | If(n, c1, c2) -> (
6      match interpret_number n s with
7        | APos -> interpret c1 s
8        | ANeg -> interpret c1 s
9        | AZero -> interpret c2 s
10       | ATop -> join_env (interpret c1 s) (interpret c2 s) )
11   | While(b, c) ->
12     let corec[iterator([])] interpret_while s =
13       match interpret_number b s with
14       | APos -> s
15       | ANeg -> s
16       | an -> join_env s (interpret_while (interpret c s))
17     in interpret_while s
```

De stelling waarin de iterator wordt aangeroepen begint op regel 11. s is hier de states van alle voorkomende variabelen. interpret_number zoekt in s welke state hoort bij de variabele b. Als dit nul of onbekend is, dan wordt het antwoord behorend bij de case an aangeroepen. interpret_while voert de code in de while loop van het programma uit en join_env zet alle ints die verschillen naar ATop. De stelling voert de while-loop dus gewoon uit en vertrouwt op de iterator om op een zeker moment de loop te herkennen en onderbreken.

Dit is een voorbeeld van hoe de eigenschappen van de iterator ook kunnen helpen bij programma's zonder oneindige datastructuren. Het is beargumenteerbaar of dit een goed voorbeeld is. Aan de ene kant doet de iterator precies wat je wilt. Het onderbreekt de loop zodra er niets meer verandert. Aan de andere kant had een simpele functie die kijkt of de twee states die aan join_env worden gegeven gelijk zijn en zoja de loop breekt ook kunnen werken. Wat de iterator doet is meer werk dan nodig is.

Naar mijn mening ligt de kracht van de iterator bij oneindige datastructuren.

3.3 Free variables: fv.ml

We hebben al laten zien dat we een set kunnen maken van een oneindige regulaire lijst (zie de functie toset op pagina 10). In fv.ml hebben we een functie die dat kan voor oneindige regulaire binaire bomen. Deze bomen worden gemaakt met de volgende datastructuur:

```
type tree = Var of string | App of tree * tree
```

De desbetreffende iterator functie heet fv en geeft een lijst van strings van alle voorkomende Vars.

```
let corec[iterator([])] fv t = match t with
| Var s -> [s]
| App(t1, t2) -> merge (fv t1) (fv t2)

let r5 = fv(Var "x")
let r6 = fv(Var "y")
let r7 = fv(App(Var "x", Var "y"))
let r8 = fv(App(Var "x", App(Var "z", Var "y")))
let rec t0 = App(Var "x", App(Var "y", t0))
let r9 = fv(t0)

val r5 : string list = ["x"]
val r6 : string list = ["y"]
val r7 : string list = ["x"; "y"]
val r8 : string list = ["x"; "y"; "z"]
val t0 : tree = App (Var "x", App (Var "y", App (Var "x", App (
    Var "y", App (Var "x", App (Var "y", ... ))))))
val r9 : string list = ["x"; "y"]
```

merge is hier een functie die gegeven twee lijsten de lijsten samenvoegt, ordent en dubbele elementen verwijdert.

Verder is er een constructor functie subst die elk voorkomen van een bepaalde Var in een gegeven tree vervangt door een andere gegeven tree. De Var wordt hier x genoemd en de vervangende tree t.

```
let corec[constructor] subst arg = match arg with
| x, t, Var v -> if v = x then t else Var v
| x, t, App(t1, t2) -> App(subst (x, t, t1), subst (x, t, t2))

let r11 = subst ("x", Var "y", Var "y")
let r12 = subst ("x", Var "y", Var "x")
let r13 = subst ("x", App(Var "x", Var "x"), App(Var "x", Var "x
    "))
let rec t0 = App(Var "x", App(Var "y", t0))
let r14 = subst("x", App(Var "z", Var "x"), t0)

val r11 : tree = Var "y"
val r12 : tree = Var "y"
val r13 : tree = App (App (Var "x", Var "x"), App (Var "x", Var
    "x"))
val t0 : tree = App (Var "x", App (Var "y", App (Var "x", App (
    Var "y", App (Var "x", App (Var "y", ... ))))))
```

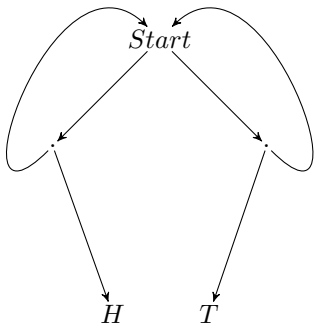
```
val r14 : tree = App (App (Var "z", Var "x"), App (Var "y", App
    (App (Var "z", Var "x"), App (Var "y", ... ))))
```

3.4 kansberekening met de gaussian solver: probability.ml

Stel je hebt een muntje die, als je hem gooit, een kans van 1 op 3 heeft om op de kopse kant te vallen. We noemen zo'n munt een eenderde munt. En je wilt een munt simuleren die een kans heeft van 1 op 2 om op de kopse kant te vallen, genaamd een halve munt. Dan kun je die halve munt simuleren met de eenderde munt op de volgende manier:
Je gooit de eenderde munt 2 keer.

- Als de munt eerst op kop en daarna op munt valt, beschouw je dit als kop.
- Als de munt eerst op munt valt en daarna op kop, beschouw je dit als munt.
- Bij elke andere combinatie gooi je opnieuw totdat je een van de bovenstaande combinaties hebt.

Zou je dit als een boom tekenen dan ziet dat er uit als de boom hieronder. Hierbij volg je de linkerpijl vanaf een punt als je kop gooit met de eenderde munt en de rechterpijl als je munt gooit.



H staat hier voor heads, oftewel kop. T staat hier voor tails, oftewel munt.

Deze boom simuleert een halve munt. Namelijk, de kans dat je bij H uitkomt is de kans dat je kop en daarna munt gooit. De kans dat je bij T uitkomt is de kans dat je munt en daarna kop gooit. Maar die kansen zijn natuurlijk even groot. Beide kansen zijn $kop \times munt$ oftewel $\frac{1}{3} \times \frac{2}{3}$.

De file probability.ml is gemaakt om zulk soort bomen te kunnen maken en vervolgens uit te rekenen wat de kans is om bij H uit te komen. Dit doet het met de hiervoor gemaakte gaussian solver. Hieronder volgt de functie probability waarmee de kans om bij H uit te komen wordt uitgerekend:

```
type tree = Heads | Tails | Flip of float * tree * tree

let corec[gaussian] probability t = match t with
  (* computes the final probability of Heads *)
| Heads -> 1.
| Tails -> 0.
| Flip(p, t1, t2) ->
    p *. probability t1 +. (1. -. p) *. probability t2
```

De bomen worden hier gemaakt met het type Flip, dat bestaat uit een drietupel. Het eerste element in de tupel is de kans om kop te krijgen als je gooit, het tweede element is de boom waar je uitkomt als je kop gooit en het derde element is de boom waar je uitkomt als je munt gooit. Onze boom zou er in code zo uitzien:

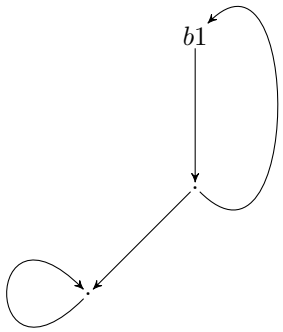
```
let eenderde= 1./.3.
let rec halvemunt = Flip(eenderde, Flip(eenderde,halvemunt,Heads
    ), Flip(eenderde,Tails,halvemunt))
let half = probability(halvemunt)

val eenderde : float = 0.333333333333
val halvemunt : tree = Flip (0.333333333333, Flip
    (0.333333333333, Flip (0.333333333333, Flip (0.333333333333,
     ... ), ... ), ... ), ... )
val half : float = 0.5
```

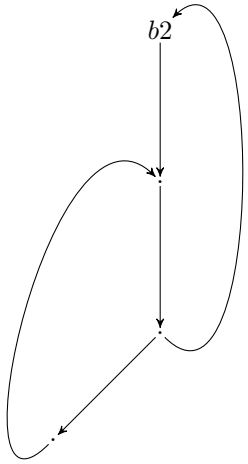
Hieronder volgen wat andere voorbeelden uit de file.

```
let r1 = probability Heads
let r2 = probability Tails
let r3 = probability (Flip(0.3, Heads, Tails))
let r4 = probability (Flip(0.3, Tails, Heads))

let rec coin0 = Flip(0.4, Heads, Flip(0.3, Tails, coin0))
let r5 = probability coin0

let rec coin1 = Flip(0.5, Heads, Flip(0.5, Tails, coin1))
let r6 = probability coin1

val r1 : float = 1.
val r2 : float = 0.
val r3 : float = 0.3
val r4 : float = 0.7
val coin0 : tree = Flip (0.4, Heads, Flip (0.3, Tails, Flip
    (0.4, Heads, Flip (0.3, Tails, Flip (0.4, Heads, Flip (0.3,
    Tails, ... ))))))
val r5 : float = 0.689655172414
val coin1 : tree = Flip (0.5, Heads, Flip (0.5, Tails, Flip
    (0.5, Heads, Flip (0.5, Tails, Flip (0.5, Heads, Flip (0.5,
    Tails,    ... ))))))
val r6 : float = 0.666666666667
```

3.5 Eigen uitwerking: Graphs

Stel je wilt de volgende twee bomen hard coded definiëren.





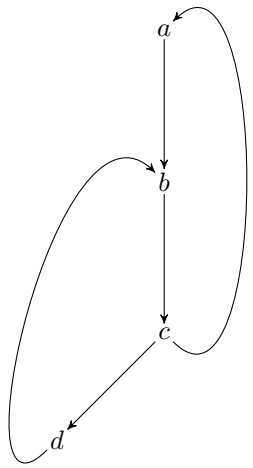
Dan kun je b1 maken als volgt:

```
type ctree = Cbranch of ctree | Cbranch2 of ctree*ctree |
    Cleaf of int

let aux = Cbranch(aux)
let b1 = Cbranch(Cbranch2(aux,b))
```

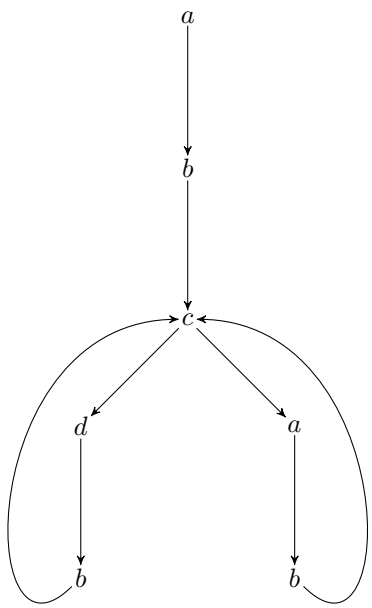
We hebben hierbij een hulpboom nodig, genaamd aux, zodat er een loop naar die boom kan wijzen. De boom b2 is hard coded niet te maken. Als je aux wil

definiëren om de linkse loop te maken, moet je naar b2 kunnen wijzen, maar om b2 te kunnen definiëren heb je aux nodig. Je hebt een kringdefinitie.

De constructor kan deze kringdefinities wel aan. De boom b2 is namelijk ook anders te tekenen. Stel we geven elke branch een letter, zodat ze makkelijker te herkennen zijn.



Dan is deze boom te transformeren naar:



Deze boom, hoewel hij er anders uit ziet, geeft exact dezelfde nodes bij dezelfde padkeuzes als b2 als je er doorheen loopt. Met andere woorden, dit is b2, maar dan anders getekend. Deze omgevormde boom is wel hard coded te definiëren, namelijk met:

```
let aux = Cbranch2(Cbranch(Cbranch(aux)), Cbranch(Cbranch(aux)));
let b2 = Cbranch(Cbranch(aux));
```

Elke boom is om te vormen tot een boom die wel hard coded te definiëren is, oftewel tot een boom waar bij elk paar loops geldt dat:

- ze naar dezelfde node wijzen, of
- de arrowhead en -tail van de ene loop allebei onder de arrowtail van de andere loop staan.

Ik heb helaas geen bewijs voor deze stelling.

De constructor bouwt bomen zonder naar de vorm te kijken; het kijkt louter naar welke paden genomen worden bij welke nodes. De constructor gebruikt dus automatisch de vorm van de boom die wel hard coded te definiëren is.

Op pagina 22 liet ik de functie totree zien, waarmee alleen bomen te maken zijn die ook hard coded te maken zijn:

```
type ltree = Leaf of int | ArrowTail | Branch of bool*ltree | Branch2 of bool*ltree*ltree

type ctree = Cbranch of ctree | Cbranch2 of ctree*ctree | Cleaf of int
```

```
4
5   let corec[constructor] totree arg = match arg with
6   | Leaf(x), h -> Cleaf(x)
7   | ArrowTail, h -> totree (h,h)
8   | Branch (b,t), h -> Cbranch(totree (t, if b then Branch(b,t)
      else h))
9   | Branch2 (b,t1,t2), h -> Cbranch2(totree (t1, if b then
      Branch2(b,t1,t2) else h), totree(t1, if b then Branch2(b,t1
      ,t2) else h))
10
11  let cb = totree (cb, Leaf(0));;
12  val cb : ctree = Cbranch (Cbranch (Cbranch2 (Cbranch (Cbranch2
      (Cbranch (Cbranch2 (Cbranch ... ), ... )), ... )), ...
      ))
```

De functie totree is met een paar simpele stappen te transformeren naar een functie die elke boom kan maken:

```
1  type ltree = Leaf of int | ArrowTail of int | Branch of int*
     ltree | Branch2 of int*ltree*ltree
2
3  type ctree = Cbranch of ctree | Cbranch2 of ctree*ctree | Cleaf
     of int
4
5  let rec get_nth = function
6  | x::y, 0 -> x
7  | x::y, z -> get_nth(y,(z-1))
8
9  let rec put_at = function
10 | t, 0, x::y -> t::y
11 | t, i, x::y -> x::(put_at(t,(i-1),y))
12
13 let corec[constructor] totree arg = match arg with
14 | Leaf(x), h -> Cleaf(x)
15 | ArrowTail (i), h -> totree (get_nth(h,i),h)
16 | Branch (i,t), h -> Cbranch(totree (t, if i>=0 then put_at (
     Branch(i,t),i,h) else h))
17 | Branch2 (i,t1,t2), h -> Cbranch2(totree (t1, if i>=0 then
     put_at (Branch2(i,t1,t2),i,h) else h), totree(t1, if i>=0
     then put_at (Branch2(i,t1,t2),i,h) else h))
```

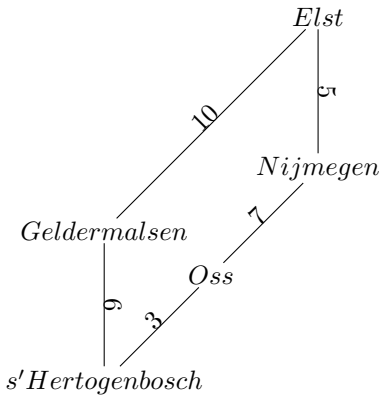
Hier krijgt elke ArrowHead een nummer en wijst elke ArrowTail naar het desbetreffende nummer. Het argument van de totree functie is nu een ltree en een lijst van leafs even groot als het aantal loops. Heb je een branch waar geen ArrowTail naar wijst, dan krijgt deze het cijfer -1. Hieronder volgen b1 en b2 als tree en als ctree.

```
1  let b1 = Branch(0,Branch2(-1,Branch(1,ArrowTail(1)),ArrowTail(0)
     ))
2  let c1= totree(b1,[Leaf(0);Leaf(0)])
3
4  let b2 = Branch(0,Branch(1,Branch2(-1,Branch(-1,ArrowTail(1)),
     ArrowTail(0))))
5  let c2= totree(b2,[Leaf(0);Leaf(0)])
```

```
val c1 : ctree = Cbranch Cbranch2 (Cbranch Cbranch Cbranch
    Cbranch ... , ... )
val c2 : ctree = Cbranch Cbranch Cbranch2 (Cbranch Cbranch
    Cbranch2 (Cbranch ... , ... ), ... )
```

Wat gebeurt er achter de schermen? De leafs worden gedurende de functie vervangen door waar de loop die hetzelfde nummer heeft als hun plaats naar wijst. Deze (oneindige) subbomen worden dan gebruikt om de totaalboom te maken.

Graphs zijn oneindige regulaire bomen. Dit maakt de constructor uitermate geschikt om te werken met graphs. Vaak heeft bij graphs elke node een naam en elke tak een waarde. Een voorbeeld van een graph is:



Met de constructor kun je de graph maken als een oneindige datastructuur waar je door heen kunt lopen. Dit zorgt dat constructor graphs geschikt zijn om de volgende soort vragen op te lossen:

- Bij welke node kom ik als ik x keer bij elke splitsing links ga
- Bij welke node kom ik als ik x keer de kortste weg kies
- Kom ik in een loop terecht als ik altijd rechtdoor ga?

Als je de graph hierboven als constructor graph wil maken, kan dat als volgt. Zie de functie tograph:

```
type data = Data of string
type cost = Cost of int
type id = Id of int
type node = Node of (id*data)
type cnode = Cnode of node* (cost*cnode) list
type path = Path of (pathtype*id*id*cost)
type pathtype = Directed | Undirected

let rec findpaths arg = match arg with
| id, [(Path(Directed,x,y,cost))] -> if x=id then [(y,cost)]
    else []
| id, (Path(Directed,x,y,cost))::restpaths -> if x=id then (y,
    cost)::findpaths(id,restpaths) else findpaths(id,restpaths)
```

```
| id, [(Path(Undirected,x,y,cost))] -> if x=id then [(y,cost)] else (if y=id then [(x,cost)] else [])
| id, (Path(Undirected,x,y,cost))::restpaths -> if x=id then (y,cost)::findpaths(id,restpaths) else (if y=id then (x,cost)::findpaths(id,restpaths) else findpaths(id,restpaths))

let rec idtonode arg = match arg with
| id, (Node(nid,ndata))::restnodes -> if id=nid then (Node(nid,ndata)) else idtonode(id,restnodes)

let corec[constructor] tonode arg = match arg with
| [],nodes,paths -> []
| (id, cost)::rest, nodes, paths -> (cost,(Cnode(idtonode(id,nodes),(tonode(findpaths(id,paths),nodes,paths)))))::(tonode(rest, nodes, paths))

let tograph arg = match arg with
| startid, nodes, paths -> (Cnode(idtonode(startid,nodes),(tonode(findpaths(startid,paths),nodes,paths))))
```

Let hier ook op de zelfgedefinieerde types:

- Een node is van de vorm Node(id,data), waarbij id een zelfgedefinieerde int is en het voor de functie niet uit maakt hoe het type data gedefinieerd is.
- Een path is van de vorm (pathtype,source id,destination id,cost), waarbij cost een zelfgedefinieerde int is. Een pathtype is Directed (je kan via dit pad alleen van de source id naar de destination id) of Undirected (je kan beide kanten op).
- Een Cnode is van de vorm (startnode, lijstje van Cnodes die bereikbaar zijn vanuit de startnode en de bijbehorende cost). Dit is de output van de functie.

De functie tograph heeft als input nodig:

- het id van de startnode,
- een lijstje van nodes, en
- een lijstje van paden.

tograph gebruikt drie hulpfuncties:

- findpaths, die uit de lijst van paden de paden vindt die vanuit de node met het gegeven id bereikbaar zijn.
- idtonode, die uit de lijst met nodes de node vindt behorend bij een id, en
- tonode. Dit is de constructor functie die de Cnode maakt.

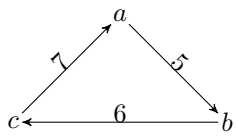
In de volgende code passen we de functie toe:

```
let nodes1=[Node(Id(1),Data("a"));Node(Id(2),Data("b"));Node(Id(3),Data("c"))]
let paths1=[Path(Directed,Id(1),Id(2),Cost(5));Path(Directed,Id(2),Id(3),Cost(6));Path(Directed,Id(3),Id(1),Cost(7))]
let paths2=[Path(Undirected,Id(1),Id(2),Cost(5))]
let paths3=[Path(Directed,Id(1),Id(2),Cost(5));Path(Directed,Id(1),Id(3),Cost(6))]
```

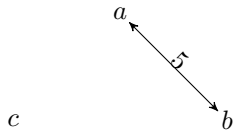
```
let graph1 = (tograph(Id(2),nodes1,paths1))
let graph2 = (tograph(Id(1),nodes1,paths2))
let graph3 = (tograph(Id(1),nodes1,paths3))

val graph1 : cnode = Cnode (Node (Id 2, Data "b"), [(Cost 6,
    Cnode (Node (Id 3, Data "c"), [(Cost 7, Cnode (Node (Id 1,
    Data "a"), [(Cost 5, Cnode (Node (Id 2, Data "b"), [(Cost 6,
    Cnode (Node (Id 3, Data "c"), [(Cost 7, Cnode (Node (Id 1,
    Data "a"),  ... )); ...])); ...])); ...])); ...])); ...])

val graph2 : cnode = Cnode (Node (Id 1, Data "a"), [(Cost 5,
    Cnode (Node (Id 2, Data "b"), [(Cost 5, Cnode (Node (Id 1,
    Data "a"), [(Cost 5, Cnode (Node (Id 2, Data "b"), [(Cost 5,
    Cnode (Node (Id 1, Data "a"),  ... )); ...])); ...])); ...]))
    ; ...])
val graph3 : cnode = Cnode (Node (Id 1, Data "a"), [(Cost 5,
    Cnode (Node (Id 2, Data "b"), [])); (Cost 6, Cnode (Node (Id
    3, Data "c"), []))])
```

Hieronder de grafische weergave van deze graphs:

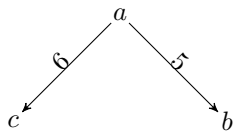
- graph1



- graph2



- graph3



Hieronder volgen een paar mogelijke vragen die over graph1 gevraagd kunnen worden en de code om ze op te lossen:

- Bij welke node kom ik als ik zes keer achter elkaar bij elke padkeuze het pad naar de node met de laagste naam kies?

```
let rec get_name arg = match arg with
| (cost, (Cnode( (Node(id,(Data(naam)))), others))) -> naam

let rec get_lowest arg = match arg with
| func, [x] -> [x]
| func, [x;y] -> if (func x) <= (func y) then [x] else [y]
| func, x::y::z -> if (func x) <= (func y) then x::z else
y::z
```

```
let get_cnode arg = match arg with
| (cost, cnode) -> cnode

let rec walk_on_name arg= match arg with
| steps, (Cnode(node,[])) -> node
| 0, (Cnode(node,others)) -> node
| steps, (Cnode(node,others)) -> walk_on_name(steps-1,
    get_cnode((hd(get_lowest(get_name,others)))))

let answer = walk_on_name (6,graph1)

val answer : node = Node (Id 2, Data "b")
```

Hierbij is de opbouw van het type cnode (= Cnode **of** node* (cost*cnode) list) heel belangrijk. De functie get_lowest geeft, gegeven een functie en een lijst, het element van de lijst dat volgens de functie het laagst is. Merk hier ook op dat de functies niet corecursief zijn.

- Bij welke node kom ik als ik acht keer achter elkaar de kortste weg kies?

```
let rec get_cost arg = match arg with
| (cost, cnode) -> cost

let rec walk_on_cost arg= match arg with
| steps, (Cnode(node,[])) -> node
| 0, (Cnode(node,others)) -> node
| steps, (Cnode(node,others)) -> walk_on_name(steps-1,
    get_cnode((hd(get_lowest(get_cost,others)))))

let answer = walk_on_cost (8,graph1)

val answer : node = Node (Id 1, Data "a")
```

Deze functie lijkt heel veel op de vorige. get_name is vervangen door get_cost.

- Kom ik in een loop als ik altijd de eerste padkeuze kies?

```
let corec[iterator(true)] walk arg = match arg with
| Cnode(node,[]) -> false
| Cnode(node,(cost,newcnode)::others) -> walk newcnode

let w1 = walk graph1
let w2 = walk graph3

val w1 : bool = true
val w2 : bool = false
```

Deze functie maakt gebruik van de iterator, aangezien het loops probeert te ontdekken.

- Zit er een loop in de graph?

```
let corec[iterator(true)] has_loop arg = match arg with
| Cnode(node,[]) -> false
| Cnode(node,(cost,newcnode)::others) -> has_loop (Cnode(
    node,others)) || has_loop newcnode

let h1 = has_loop graph1
let h2 = has_loop graph3

val h1 : bool = true
val h2 : bool = false
```

  Deze functie werkt door alle mogelijke paden te nemen totdat elk pad of eindigt of loopt.

Voordelen van de Cnode als graphrepresentatie:
- Je kunt door je graph heen lopen.
- De graphs zijn simpel te definiëren. Je hebt alleen een rijtje nodes, een rijtje paden en een startnode nodig.
- Door de graph heen lopen gaat zeer snel.

Nadelen van de Cnode als graphrepresentatie:
- Zeer ongeschikt als je je graph regelmatig wil aanpassen. De enige manier is namelijk om met een constructor een nieuwe graph te bouwen. Dit geldt ook als je de data van een node wil veranderen.

3.6 Eigen uitwerking: 2-dimensionale oneindige matrices

In de code hieronder is zeros een 1-dimensionale matrix en matrix een 2-dimensionale matrix.

```
let rec zeros = 0::zeros
let rec matrix = zeros::matrix

val zeros : int list = [0; 0; 0; 0; ...]
val matrix : int list list = [([0; 0; 0; 0; ...]); ...]
```

Een oneindige matrix geeft je een ruimte waar je geen rekening hoeft te houden met de grenzen, aangezien de ruimte oneindig is. Dit geeft wat handige eigenschappen. Zo hoeft de functie put (die gegeven een object, een locatie en een oneindige lijst het object op de locatie in de lijst zet) geen rekening te houden met de situatie waarin de lijst eindigt. De case | el,x,[] is niet nodig.

```
let rec put arg = match arg with
| el,0, loc::list -> el::list
| el,x, loc::list -> loc::(put (el,x-1,list))
```

Hieronder wat functies die handig zijn bij het datatype int int list. Aanname is hier dat er nooit negatieve coördinaten worden meegegeven.

get2 (*x,y,matrix*), geeft het element op de gegeven coördinaten in de matrix. Aangezien deze functie nooit eeuwig loopt, hoeft de functie niet corecursief te zijn.

```
let rec get2 arg = match arg with
| 0,y, el::column::row -> get (y,column)
| x,y, column::row -> get2 (x-1,y,row)
```

put2 (*el,x,y,matrix*), zet het gegeven element op de gegeven coördinaten in de matrix. Ook deze functie kan niet eeuwig loopen en kan dus recursief zijn.

```
let rec put2 arg = match arg with
| el,0,y, loc::column::row -> (put (el,y,column))::row
| el,x,y, column::row -> column::(put2 (el,x-1,y,row))
```

changecolumn (*a,x,matrix*), verandert alle elementen die de gegeven x-coördinaat hebben naar de gegeven waarde. Aangezien deze functie een oneindig aantal elementen verandert, is hier een constructor functie nodig.

```
let corec[constructor] make_list arg = arg::(make_list arg)

let corec[constructor] changecolumn arg = match arg with
| changeto, 0, column::row -> (make_list changeto)::row
| changeto, at, column::row -> column::(changecolumn(
    changeto,at-1,row))
```

changerow (*a,y,matrix*), verandert alle ellementen die de gegeven y-coördinaat hebben naar de gegeven waarde.

```
let corec[constructor] changerow arg = match arg with
| changeto, at, column::row -> (put (changeto,at,column))
    ::(changerow (changeto,at,row))
```

map2 (*f,matrix*), mapt een functie over een matrix.

```
let corec[constructor] map2 arg = match arg with
| func, column::row -> (map (func, column))::(map2 (func,
    row))
```

Hoofdstuk 4

Suggesties

In dit hoofdstuk suggesties van wat in mijn ogen goede veranderingen aan COcaml zouden zijn. Ik heb ze onderverdeeld in de volgende secties en subsecties:

Aanpassingen, alles wat COcaml beter in het gebruik zou maken zonder dat de taal substantieel verandert.

- Console
- Bugs
- Gebruiksvriendelijkheid

Mogelijkheden, de verschillende richtingen die COcaml op zou kunnen gaan.

- Oneindige logische datastructuren
- Corecursieve functies als first-class citizens
- Ingewikkelde oneindige datastructuren

4.1 Aanpassingen

4.1.1 Console

In deze sectie suggesties van wat mooier kan worden weergegeven in de console. Door deze scriptie heen heb ik verschillende keren verbeterpunten opgemerkt, maar ik heb ook weergaves verzwegen of aangepast, zodat de scriptie leesbaar bleef. Hier een overzicht van alle punten.

functies
Stel ik heb de volgende functie:

```
1  let corec[iterator(false)] mem arg = match arg with
2  | e, [ ] -> false
3  | e, h :: t -> e = h || mem (e, t)
```

Dan wordt deze functie zo weergegeven in de console:

```
val mem : 'a -> 'a list -> bool = fun e l -> exists ((fun i -> e = i), l)
```

Dit was misschien handig toen de corecursieve functies nog in hun testfase zaten, maar nu ze werken is dit een betere weergave:

```
val mem : 'a -> 'a list -> bool = <Iterator fun>
```

Oneindige tupels

Eindige tupels worden wel goed weergegeven (hoewel er nog wel twee haakjes rond om heen mogen):

```
let a = ((1,2),(3,4))
val a : ((int * int) * (int * int)) = (1, 2), (3, 4)
```

Oneindige tupels worden zo weergegeven

```
let rec a = (0,a)
val a : (int * (int * 'a)) = 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
let rec b = (0,0,b)
val b : (int * int * (int * int * 'a)) = 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
```

Oneindige structuren in eindige structuren

Stel ik heb een 2-tupel van een int list en een int. Dan laat de console de int niet zien als de int list oneindig is. In plaats daarvan laat het drie puntjes zien:

```
let rec zeros = 0::zeros
let a = (zeros,1)

val a : (int list * int) = [0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; ...], ...
```

Dit geldt niet alleen in dit geval, maar in elk geval waar een oneindige datastructuur in een eindige datastructuur wordt weergegeven en er nog data volgt na de oneindige datastructuur. Als dit gebeurt zou de compiler moeten herkennen dat de oneindige datastructuur zich in een eindige datastructuur bevindt en dat deze eindige datastructuur wel moet worden weergegeven.

Loops

Zie de volgende declaraties:

```
let zeros = 0::zeros
let a = 2::1::2::zeros
val zeros : int list = [0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; ...]
val a : int list = [2; 1; 2; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; ...]
```

Dan zijn de weergaves van deze twee lijsten onnodig lang. Het feit dat de console bij beide lijsten even veel nullen laat zien, wijst er op dat het wel iets van een check doet om te kijken of de lijst regulair is. Dus wat ik in dit geval mooier zou vinden is bijvoorbeeld dit:

```
let zeros = 0::zeros
let a = 2::1::2::zeros
val zeros : int list = [0; 0; 0; ...]
val a : int list = [2; 1; 2; 0; 0; 0; ...]
```

Wat helemaal mooi zou zijn, is het volgende:

```
  let zeros = 0::zeros
  let a = 2::1::2::zeros
  let alt = 1::2::alt
  val zeros : int list = [>0>]
  val a : int list = [2; 1; 2; >0>]
  val alt : int list = [>1; 2>]
```

Waarbij het deel tussen de twee >'s het loopende deel is. Hier is wel voor nodig dat de console gebruik kan maken van de iterator, wat mogelijk ingewikkeld is.

4.1.2 Bugs

In deze sectie een verzameling van alle bugs. Ik beschouw iets als een bug als het ervoor zorgt dat je niet op een bepaalde manier dingen op kan schrijven die in Ocaml wel mogelijk is of als een foutmelding onduidelijk is. Een aantal van deze bugs zijn pas echt goed op te lossen als COcaml een eigen taal wordt.

function keyword

Dit is hoe het function keyword normaal werkt:

```
let f = function
| b -> b
val f : 'a -> 'a = function b -> b

f 2
- : int = 2

let f arg = function
| b -> b
val f : 'a -> 'b -> 'b = fun arg -> (function b -> b)

f 2 3
- : int = 3
```

Met het function keyword kun je een argument weglaten in de eerste regel van je functie. Dit argument gebruik je in de stellingen.

Dit is hoe de iterator er mee omgaat:

```
let corec[iterator(0)] f = function
| b -> b
val f : 'a -> 'a = fun f corec[iterator 0] -> (function b ->
    b)

f 2
Runtime error: Corecursive function must have exactly one
    argument

let corec[iterator(0)] f a = function
| a-> a
val f : 'a -> 'b -> 'b = fun f corec[iterator 0] a -> (
    function a -> a)

> f 2
- : 'a -> 'a = function a -> a
> f 2 3
- : int = 3

let corec[iterator(0)] f a = function
| [] -> a
| x::y -> f a y
val f : 'a -> 'b list -> 'a = fun f corec[iterator 0] a -> (
    function [] -> a | x :: y -> f a y)

f 1 zeros
Fatal error: exception Stack_overflow
```

standaardfuncties doen het niet

Zoals string_to_int en sort.

-1 mag niet in de case voorkomen

```
let x arg = match arg with
| -1 -> 3
Parse error at line 2 char 3

let x arg = match arg with
| (-1) -> 3
Parse error at line 2 char 4

let x arg = match arg with
| -(1) -> 3
Parse error at line 2 char 3

let x arg = match arg with
| 2::-1::y -> 3
Parse error at line 2 char 6

let x arg = match arg with
| (2,-1) -> 3
```

```
Parse error at line 2 char 6
```

Een zelfgedefinieerde type mag niet in een datastructuur in de case voorkomen zonder haakjes

Dit kan wel:

```
type getalletje = Getal of int

let f arg = match arg with
| Getal(x) -> 5
val f : getalletje -> int = fun arg -> (match arg with Getal x
    -> 5)

let f arg = match arg with
| x::rest -> 5
val f : 'a list -> int = fun arg -> (match arg with x :: rest
   -> 5)
```

En dit niet:

```
let f arg = match arg with
| Getal(x)::rest -> 5
Type error: conflict involving 'a list and int
```

In Ocaml kan het wel:

```
# type getalletje = Getal of int;;
type getalletje = Getal of int

# let f arg = match arg with
  | Getal(x)::rest -> 5;;
Characters 15-53:
  ...............match arg with
    | Getal(x)::rest -> 5..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val f : getalletje list -> int = <fun>
# f([Getal(2)]);;
- : int = 5
# f([Getal(2);Getal(5)]);;
- : int = 5
```

Er valt rond om heen te werken met simpelweg meer haakjes, maar het is niet zoals het hoort.

```
let f arg = match arg with
| (Getal(x))::rest -> 5

val f : getalletje list -> int = fun arg -> (match arg with
   Getal x :: rest -> 5)
```

Als een foutmelding de regel wil geven waar de fout in een file zit, telt het comments niet mee als een regel
Stel ik heb de volgende file:

```
1  (*
2
3  *)
4
5  let a = 5
6
7  let a = (()
8
9  let a = 5
10
11 let a = 5
```

Dan is de foutmelding: Parse error at line 7 char 3, terwijl de foutmelding op line 9 is (de fout is **let** in een tupel).

Heb ik de volgende file dan geeft het wel de goede foutmelding:

```
1  (**)
2  (**)
3  (**)
4
5  let a = 5
6
7  let a = (()
8
9  let a = 5
10
11 let a = 5
```

4.1.3 Gebruiksvriendelijkheid

In deze sectie dingen in COcaml die logisch zijn, maar die de taal gebruiksvriendelijker zouden maken als ze anders waren.

De iterator en if-statements
Zie de functie toset2.

```
1  let corec[iterator([])] toset2 arg = match arg with
2  | [] -> []
3  | x::y -> if mem x (toset2 y) then (toset2 y) else x::(toset2
     y)
```

Als je deze functie een eindige lijst geeft, geeft het een set van elementen die voorkomen in de lijst terug.

Als je deze functie een oneindige lijst meegeeft, komt het programma in een eeuwige loop. Dit is logisch, want de iterator kan niet zeggen of mem x (toset2 y) waar is, totdat het d()toset2 y) bekeken heeft. En dat kan het niet totdat het een loop gevonden heeft, wat het pas kan als het het antwoord weet op de if-statement.

Toch is eeuwig blijven loopen niet de meest handige manier om dit aan te geven. Beter zou zijn als de compiler opmerkte dat deze functie geen antwoord kan geven als toset2 een oneindige input krijgt.

corecursieve functies zonder argumenten
Op dit moment zijn corecursieve functies zonder argumenten nutteloos:

```
> let corec[constructor] zeros = 0::zeros;;
val zeros : int list = fun zeros corec[constructor] -> 0 ::
    zeros
> hd zeros;;
Runtime error: Match failure on fun zeros corec[constructor]
    -> 0 :: zeros
```

Misschien wil je dit wel toestaan. Misschien wil je een waarschuwing geven als iemand dit probeert.

iterators die niet bouwen op hun solverargument kunnen type-inconsequent zijn
In dit voorbeeld wordt het solverargument niet gebruikt in de antwoorden, wat de functie de kans geeft type-inconsequent te zijn.

```
let corec[iterator(0)] f arg = match arg with
| [] -> true
| x::y -> f y
> f [0;0;0];;
- : bool = true
> f [];;
- : bool = true
> f zeros;;
- : bool = 0
```

Hoewel dit overduidelijk de schuld van de programmeur is als hij dit doet, mag hier toch een foutmelding voor komen.

4.2 Mogelijkheden

In deze sectie suggesties welke richting COcaml op kan groeien.

4.2.1 oneindige logische datastructuren

Corecursieve functies kunnen op dit moment alleen omgaan met regulaire oneindige datastructuren. Maar een zeer waarschijnlijke uitbreiding van COcaml, volgens de makers, is corecursieve functies die om kunnen gaan met logische oneindige datastructuren. Voordat ik uitleg wat ik daarmee bedoel, een voorbeeld:

```
let plusones=1::(1+plusones)

val plusones : int list [1; 2; 3; 4; 5; 6; 7; ...]
```

In dit voorbeeld is plusones de oneindige lijst gevuld met optellende nummers. Oneindige datastructuren hoeven dus niet per se te bestaan uit een eindig aantal verschillende elementen. Met deze uitbreiding valt elke datastructuur te maken die logisch te definiëren is. Een paar voorbeelden:

```
1  let plusas="a"::("a"+plusas)
2  val plusas : string list ["a"; "aa"; "aaa"; "aaaa"; ...]
3
4  let timestwo=1::(2*timestwo)
5  val timestwo : int list [1; 2; 4; 8; 16; 32; ...]
6
7  let x = plusones+timestwo
8  val x : int list [2; 4; 7; 12; 21; 38; ...]
```

De syntax kan verschillen, maar deze lijsten zijn dan te maken.

Een iterator functie over logische datastructuren is een stuk krachtiger dan de iterator die we nu hebben. Zo is de volgende functie larger_than_5, die kijkt of er ooit een getal groter dan 5 in de lijst voorkomt, een stuk interessanter bij logische datastructuren.

```
1  let corec[iterator(false)] larger_than_5 arg = match arg with
2  | [] -> false
3  | x::y -> if x>5 then true else larger_than_5 y
```

4.2.2 Corecursieve functies als first-class citizens

In functionele talen worden functies regelmatig beschreven als first-class citizens. Wat hiermee bedoelt wordt is dat de zo taal gemaakt is dat voor bijna probleem een functie de meest praktische en voor de hand liggendste oplossing is. Functies zijn de kracht van de taal en alles is aangepast zodat makkelijk met functies gewerkt kan worden. Zo kun je een 1-arguments functie maken door een 2-arguments functie een argument te geven.

In deze versie van COcaml zijn corecursieve functies tweederangs burgers. Ze worden opzijgeduwd door de vaak handigere en intuïtievere recursieve functies. Hieronder beschrijf ik wat goede stappen zouden zijn om corecursieve functies naar de eerste plaats te krijgen.

Solvers om kunnen laten gaan met datastructuren waar de structuur al van bekend is

Stel ik heb de lijst zeros en ik wil het 100ste element wijzigen naar een 1. Dan kan dit met de constructor functie, maar de recursieve variant is sneller. De reden daarvoor is dat de constructor de argumenten van elke aanroep bijhoudt en doorkijkt om een loop te ontdekken. Maar we weten al waar de loop is (op plek 100 in de lijst) en we weten al hoe de loop er uit ziet (zeros). De constructor zou dus veel sneller kunnen zijn als we deze informatie konden geven. Dan zou het namelijk simpelweg de lijst uitbreiden tot het 99ste element, dan er een 1 achter plaatsen en dan zeros er achter plaatsen.

Dit is precies wat de recursieve variant doet, wat de recursieve variant hier geschikter maakt.

Maar er zijn gevallen waar de constructor functie sneller kan, waar de recursieve variant niet werkt. Stel we hebben de lijst a die gemaakt is door in zeros het 100ste element op 1 te zetten. En we willen nu elk element in a met 1 verhogen. Dan moet dit met een constructor functie. Maar ook hier weten we waar de loop ligt en hoe de loop er uit ziet. We zouden de constructor dus veel werk kunnen besparen.

Een ander voorbeeld. Stel we hebben een graph gemaakt met de functie tograph in sectie 3.5 , Graphs, en we willen de data van een zekere node veranderen. Dan weten we dat de structuur van de graph niet verandert.

Er zijn twee vormen van informatie die handig kunnen zijn om aan een functie mee te geven:

- Is de functie structure safe? Oftewel heeft de output dezelfde structuur als een van de argumenten? Welk argument?
- Zoniet, hoe hangt de structuur van de output af van de input? Oftewel, wat is de structure dependence?

Een voorbeeld: de functie remove hieronder is niet structure safe, met uitzondering van het geval dat het eerste argument niet voorkomt in het tweede argument. De structure dependence is als volgt:

- De plek van de loop in de output lijst is de plek van de loop in het tweede argument minus het aantal keer dat het eerste argument voorkomt in het gedeelte voor de loop.
- De lengte van de loop in de output lijst is de lengte van de loop in het tweede argument minus het aantal keer dat het eerste argument voorkomt in de loop.

```
1   let corec[constructor] remove arg = match arg with
2   | a, [] -> []
3   | a, x::y -> if x=a then remove (a,y) else remove (a,x::y)
```

Stel we willen deze informatie aan een constructor functie geven, dan kan dat op twee manieren: we laten de programmeur dit regelen of we laten de compiler dit regelen.

Als we de compiler dit laten regelen, dan kunnen we alleen structure safety checken, want structure dependence is te ingewikkeld. Nadeel van deze tactiek is dat de programmeur weet waarvoor de functie bedoeld is en de compiler niet. Voordeel is dat het onzichtbaar is voor de programmeur.

Als we de programmeur de verantwoordelijkheid geven, zijn er twee mogelijkheden: of we laten hem alleen structure safety bepalen, of we geven hem ook de optie structure dependency te bepalen.

- Het voordeel van de eerste manier is dat het makkelijk is. Zo is de hierboven beschreven functie die elk element in een lijst met 1 verhoogt overduidelijk structure safe. Dit is een belangrijk voordeel, want als je een verkeerde functie structure safe verklaart krijg je vreemde uit-

komsten.

- Het voordeel van de tweede manier is dat het de programmeur veel controle geeft om te optimaliseren. Zo kan het zijn dat de programmeur, als hij de functie remove gebruikt, al weet hoeveel elementen de functie gaat verwijderen en deze informatie dus kan meegeven. Het nadeel is dat dit vrij ingewikkeld is en fouten vergaande gevolgen kunnen hebben.

Ook iterator functies kunnen sneller zijn als we mee kunnen geven hoe vaak ze worden aangeroepen, zodat we de loop op de juiste plek kunnen afbreken.

Alles is een corecursieve functie

Het idee waar functionele talen op werken is dat alles een functie is. Bij corecursieve functies ontbreekt dit principe op twee manieren:

- Je kunt op dit moment geen oneindige lijst van functies maken.
- De solvers kunnen op dit moment niet omgaan met functies die meer dan 1 argument krijgen.

Dit betekent dat het volgende niet kan met corecursieve functies:

```
let f a b = a+b
val f : int -> int -> int = fun a b -> a + b

let g = f 2
val g : int -> int = fun b -> %1 + b

map g [1;2;3]
- : int list = [3; 4; 5]

map f [1;2;3]
- : int -> int list = [(fun b -> %590 + b); (fun b -> %599 +
    b); (fun b -> %608 + b)]
```

Dotsvervanger

```
let corec[constructor] filter arg = match arg with
| f, [ ] -> [ ]
| f, h :: t ->
   if f h then h :: filter_naive(f, t)
   else filter_naive(f, t)
```

Zie de functie filter hierboven. Deze functie geeft dots terug als het een oneindige lijst krijgt waar elk element uit weg gefilterd wordt. De oplossing die in de list.ml file hiertegen wordt gebruikt is veelvoudig exists aanroepen. Maar dots is gewoon een object en dus te herkennen en te vervangen. Fijner zou hier dus zijn als we als optineel argument aan de constructor een dotsvervanger konden meegeven, in dit geval een lege lijst. Dat zou er zo uit kunnen zien:

```
1  let corec[constructor(dots=[])] filter arg = match arg with
2  | f, [ ] -> [ ]
3  | f, h :: t ->
4          if f h then h :: filter_naive(f, t)
5          else filter_naive(f, t)
```

Kortere namen

Als we de constructor en de iterator first-class citizens maken, is het handig als we niet elke keer als we een functie maken **let** corec[iterator ([])] f arg = of **let** corec[constructor] f arg = hoeven te typen. Fijner zou zijn: **let** itr ([]) f arg = en **let** con f arg =.

De iterator data laten negeren

Op dit moment is het niet mogelijk om een iterator functie te schrijven die gegeven een element en een lijst de index van dat element in de lijst geeft.

Wel is het mogelijk om een iterator functie te maken die kijkt of het element in de lijst voorkomt en vervolgens een recursieve functie de index te laten geven, maar het kan veel handiger.

Beter zou zijn als we bepaalde input als metadata konden specificeren en dat de iterator vervolgens deze metadata negeert bij het vinden van de loop. De functie get_index zou er dan als volgt uit kunnen zien:

```
1  let corec[iterator(-1)] get_index arg = match arg with
2  | el, meta(index), [] -> -1
3  | el, meta(index), x::y -> if x=el then index else get_index(
       el, index+1, y)
```

Dit is handig, want rekening moeten houden met wanneer de iterator gaat loopen is een van de moeilijkste dingen aan deze solver. Zo kan ik bijvoorbeeld geen functie schrijven die bij een graph kijkt wat de cost van een route is.

Als we metadata toestaan kan dit wel:

```
1  let corec[iterator(-1)] route_cost arg = match arg with
2  | (Cnode(node,[])), meta(cost), pick_node -> cost
3  | (Cnode(node,other)), meta(cost), pick_cnode -> route_cost (
       pick_cnode(other), cost+get_cost(pick_cnode(other)),
       pick_cnode)
```

4.2.3 Ingewikkelde oneindige datastructuren

Hetgene waar de iterator en de constructor op dit moment in uitblinken zijn ingewikkelde oneindige datastructuren. Hiermee bedoel ik oneindige trees, graphs en (met een paar aanpassingen) oneindige multi-dimensionale matrices. Als we dit als richting kiezen waarin we COcaml willen ontwikkelen, dan lijken mij de volgende twee toevoegingen zeer nuttig.

functies die matrices van verschillende dimensies kunnen geven
In sectie 3.6 laat ik functies voor oneindige 2-dimensionale matrices zien. Het oorspronkelijke idee was om deze functies te maken voor multi-dimensionale matrices, maar ik stuitte op serieuze problemen toen ik dit probeerde te maken. Zie de volgende functie:

```
1      let corec[constructor] ndimx arg = match arg with
2      | 0, x -> [x]
3      | 1, x -> x::(ndimx(1,x))
4      | n, x -> (ndimx ((n-1),x))::(ndimx(n,x))
5
6    let zero = ndimx(0,0)
7    let zeros = ndimx(1,0)
8    let matrix = ndimx(2,0)
9
10   val ndimx : (int * 'a) -> 'a list
11   val zero : int list = [0]
12   val zeros : int list = [0; 0; 0; 0; ...]
13   val square : int list = [[0; 0; 0; 0; ...]; ...]
```

Deze functie zou, gegeven een getal `n` en een object `x`, een n-dimensionale matrix gevuld met `x` moeten geven. Maar helaas doet deze functie niet wat je verwacht. Het geeft inderdaad n-dimensionale matrices, maar ze hebben allemaal het type `int list`. Daardoor is bijvoorbeeld `matrix=ndimx (2,0)` onbruikbaar voor functies die een `int list` verwachten, maar ook voor functies die een `int list list` verwachten. Het heeft namelijk het type `int list`, maar is een `int int list`. Hoe komt deze vreemde typering? Dat komt omdat `ndimx` type-consistent moet zijn. Je kunt geen functie hebben die zowel `'a list`'s als `'a list list`'s als `'a list list list`'s, etc. teruggeeft, terwijl de input niet van type verandert. Om dezelfde reden geeft bijvoorbeeld de volgende functie `a` een foutmelding omdat die type-inconsistent is. Bij `a` is dit bedoeld gedrag.

```
1    let a arg = match arg with
2    | true -> []
3    | false -> 3
```

Ook met zelfgedefinieerde types kun je geen types van willekeurige dimensie maken. Je kan namelijk geen type definieren als de lijst van datzelfde type.

Functies die werken over oneindige matrices van elke dimensie lijken mij een interessante toevoeging aan de taal. Dit geeft namelijk de kans om een matlab-achtige omgeving te maken in een functionele taal, waarbij je geen rekening hoeft te houden met de grenzen van je matrix, aangezien die oneindig is. De volgende functies zouden mogelijk zijn als functies matrices van verschillende dimensies als output konden geven. Aanname bij elke functie is dat de lengte van het lijstje van coördinaten overeenkomt met het aantal dimensies van de matrix en dat de coördinaten positief zijn. De functies kunnen uitgebreid worden met stellingen met foutmeldingen om dit af te dwingen, maar mijn mening is dat dit afbreuk doet aan de leesbaarheid.

getn (*coors*,*matrix*), die gegeven een lijstje van coördinaten en een matrix, het element op de coördinaten in de matrix geeft. Deze functie werkt ook in de huidige taal al, aangezien het geen aanname hoeft te doen over de dimensie van de matrix.

```
1   let rec getn arg = match arg with
2   | [coor], list -> get(coor,list)
3   | 0::coors, dim::mat -> getn (coors,dim)
4   | coor::coors, dim::mat -> getn ((coor-1)::coors, mat)
5
6   val getn : (int list * 'a list) -> 'a
```

putn (*a*,*coors*,*matrix*), die gegeven een object, een lijstje van coördinaten en een matrix, het object op de coördinaten in de matrix zet. Deze functie werkt in de huidige taal niet, aangezien de functie een impliciete aanname moet maken hoe de dimensie van de matrix afhangt ten opzichte van het object. Het maakt de aanname dat de matrix 1-dimensionaal is.

```
1 let rec putn arg = match arg with
2 | el, [coor], list -> put (el,coor,list)
3 | el, 0::coors, dim::mat -> (putn (el, coors,dim))::mat
4 | el, coor::coors, dim::mat -> dim::(putn (el, (coor-1)::
      coors, mat))
5
6 val putn : ('a * int list * 'a list) -> 'a list
```

changedims(*a*,*coors*,*matrix*). Deze functie mag behalve positieve coördinaten ook de coördinaat -1 meekrijgen. Het verandert alle elementen die voldoen aan de positieve coördinaten. Als er geen -1 wordt meegegeven doet changedims dus exact hetzelfde als putn. Als changedims de coördinaten [-1;y] en een 2-dimensionale matrix meekrijgt, waarbij y positief is, doet changedims hetzelfde als changerow in sectie 3.6 , 2-dimensionale oneindige matrices. Als het in plaats daarvan als coördinaten x;-1 meekrijgt, waarbij x positief is, doet het hetzelfde als changecolumn. Als de coördinaten alleen maar bestaan uit -1's, wordt heel de matrix gevuld met *a*.

```
1   let corec[constructor] changedims arg = match arg with
2   | changeto, [coor], list ->
3         if coor=-1
4   then ndimx (1,changeto)
5    else put (changeto,coor,list)
6   | changeto, 0::coors, dim::mat -> (changedims (changeto,
        coors,dim))::mat
7   | changeto, coor::coors, dim::mat ->
8         if coor=-1
9         then (changedims (changeto, coors, dim))::(
              changedims (changeto, (-1::coors), mat))
10        else dim::(changedims (changeto, (coor-1)::coors,
              mat))
```

Sketchers

Als we COcaml voornamelijk gebruiken voor trees, graphs en matrices, is het wel zo handig als we die goed kunnen weergeven. Goede weergaves van trees en graphs lijken mij weergaves die lijken op de weergaves die ik daarvoor in deze scriptie heb gebruikt. Goede weergaves van oneindige matrices boven de 2 dimensies lijken mij niet mogelijk, dus ik zou alleen weergaves van 2-dimensionale matrices toevoegen. Stel we definiëren een 2-dimensionale oneindige matrix als:

```
let matrix = zeros::matrix
let matrix = put2(4,0,4,matrix)
let matrix = changerow(1,1,matrix)
let matrix = changecolumn(1,1,matrix)
let matrix = put2(2,5,1,matrix)
```

Dan zou dit een mooie weergave zijn:

```
 0 1 0 0 4 0 >
 1 1 1 1 1 1 >
 0 1 0 0 0 0 >
 0 1 0 0 v v
 0 4 0 >
 0 1 0 >
 v v v
```

Waarbij > aangeeft dat de matrix links ervan regulair is en v dat de matrix eronder regulair is.

Hoofdstuk 5

Gerelateerd onderzoek

Bibliografie

[1] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva,
Language constructs for non-well-founded computation.,
Technical Report SEN-1202, CWI Amsterdam, July 2012. 22nd European Symposium on Programming (ESOP'13), to appear.

[2] The ocaml community,
What is Ocaml?,
`ocaml.org`

COcaml en de voorbeeldfiles zijn downloadbaar op `http://www.cs.cornell.edu/Projects/CoCaml/`. Ik maak gebruik van versie 4, laatst aangepast op 1 april 2013.

Hoofdstuk 6

Verder Onderzoek

Implementatiegericht, voor veranderingen en uitbreidingen aan COcaml, contacteer de makers op `ttp://www.cs.cornell.edu/Projects/CoCaml/`. Zij kunnen helpen door middel van kennis, maar ook implementatie. Mogelijke uitbreidingen heb ik beschreven in hoofdstuk 4 , Suggesties, maar andere ideeën zijn mogelijk.

Documentatiegericht, hetgene wat ik expliciet niet aan deze taal heb onderzocht is de werking van eigengemaakte solvers. Welke solvers zijn maakbaar? Hoe maakt men deze solvers? Wat zijn de mogelijkheden van deze solvers?

Bijlage A

Appendix

tree.ml

```
type ltree = Leaf of int | ArrowTail of int | Branch of int*
    ltree | Branch2 of int*ltree*ltree

type ctree = Cbranch of ctree | Cbranch2 of ctree*ctree | Cleaf
    of int


type path = Undir of int*int*int | Dir of int*int*int (*id1, id2
    , cost*)

let rec get_nth = function
    | x::y, 0 -> x
                | x::y, z -> get_nth(y,(z-1))

let rec put_at = function
        | t, 0, x::y -> t::y
        | t, i, x::y -> x::(put_at(t,(i-1),y))

(*arg = boom,h*)
(* h = [boom], de arrowheads info. begint als [Leaf(0)]*)
let corec[constructor] totree arg = match arg with
| Leaf(x), h -> Cleaf(x)
| ArrowTail (i), h -> totree (get_nth(h,i),h)
| Branch (i,t), h -> Cbranch(totree (t, if i>=0 then put_at (
    Branch(i,t),i,h) else h))
| Branch2 (i,t1,t2), h -> Cbranch2(totree (t1, if i>=0 then
    put_at (Branch2(i,t1,t2),i,h) else h), totree(t1, if i>=0
    then put_at (Branch2(i,t1,t2),i,h) else h))

let b1 = Branch(0,Branch2(-1,Branch(1,ArrowTail(1)),ArrowTail(0)
    ))
let c1= totree(b1,[Leaf(0);Leaf(0)])

let b2 = Branch(0,Branch(1,Branch2(-1,Branch(-1,ArrowTail(1)),
    ArrowTail(0))))
```

```
let c2= totree(b2,[Leaf(0);Leaf(0)])

let b3 = Branch(0,Branch(1,Branch2(-1,ArrowTail(1),Branch(-1,
    ArrowTail(0)))))
let c3= totree(b3,[Leaf(0);Leaf(0)])
```

graph.ml

```
type data = Data of string
type cost = Cost of int
type id = Id of int
type node = Node of (id*data)
type cnode = Cnode of node* (cost*cnode) list
type path = Path of (pathtype*id*id*cost)
type pathtype = Directed | Undirected

let rec findpaths arg = match arg with
| id, [(Path(Directed,x,y,cost))] -> if x=id then [(y,cost)] else []
| id, (Path(Directed,x,y,cost))::restpaths -> if x=id then (y,cost)::findpaths(id,restpaths) else findpaths(id,restpaths)
| id, [(Path(Undirected,x,y,cost))] -> if x=id then [(y,cost)] else (if y=id then [(x,cost)] else [])
| id, (Path(Undirected,x,y,cost))::restpaths -> if x=id then (y,cost)::findpaths(id,restpaths) else (if y=id then (x,cost)::findpaths(id,restpaths) else findpaths(id,restpaths))

let rec idtonode arg = match arg with
| id, (Node(nid,ndata))::restnodes -> if id=nid then (Node(nid,ndata)) else idtonode(id,restnodes)

let corec[constructor] tonode arg = match arg with
| [],nodes,paths -> []
| (id, cost)::rest, nodes, paths -> (cost,(Cnode(idtonode(id,nodes),(tonode(findpaths(id,paths),nodes,paths)))))::(tonode(rest, nodes, paths))

let tograph arg = match arg with
| startid, nodes, paths -> (Cnode(idtonode(startid,nodes),(tonode(findpaths(startid,paths),nodes,paths))))

let nodes1=[Node(Id(1),Data("a"));Node(Id(2),Data("b"));Node(Id(3),Data("c"))]
let paths1=[Path(Directed,Id(1),Id(2),Cost(5));Path(Directed,Id(2),Id(3),Cost(6));Path(Directed,Id(3),Id(1),Cost(7))]
let paths2=[Path(Undirected,Id(1),Id(2),Cost(5))]
let paths3=[Path(Directed,Id(1),Id(2),Cost(5));Path(Directed,Id(1),Id(3),Cost(6))]

let graph1 = (tograph(Id(2),nodes1,paths1))
let graph2 = (tograph(Id(1),nodes1,paths2))
let graph3 = (tograph(Id(1),nodes1,paths3))

let rec get_name arg = match arg with
| (cost, (Cnode( (Node(id,(Data(naam)))), others))) -> naam

let rec get_lowest arg = match arg with
| func, [x] -> [x]
| func, [x;y] -> if (func x) <= (func y) then [x] else [y]
```

```
| func, x::y::z -> if (func x) <= (func y) then x::z else y::z

let get_cnode arg = match arg with
| (cost, cnode) -> cnode

let rec walk_on_name arg= match arg with
| steps, (Cnode(node,[])) -> node
| 0, (Cnode(node,others)) -> node
| steps, (Cnode(node,others)) -> walk_on_name(steps-1, get_cnode
    ((hd(get_lowest(get_name,others)))))

let answer = walk_on_name (6,graph1)

let corec[iterator(true)] walk arg = match arg with
| Cnode(node,[]) -> false
| Cnode(node,(cost,newcnode)::others) -> walk newcnode

let w1 = walk graph1
let w2 = walk graph3

let corec[iterator(true)] has_loop arg = match arg with
| Cnode(node,[]) -> false
| Cnode(node,(cost,newcnode)::others) -> has_loop (Cnode(node,
    others)) || has_loop newcnode

let h1 = has_loop graph1
let h2 = has_loop graph3
```

ndim.ml

```
let corec[constructor] ndimx arg = match arg with
        | 0, x -> [x]
        | 1, x -> x::(ndimx(1,x))
        | n, x -> (ndimx ((n-1),x))::(ndimx(n,x))

let zero = ndimx(0,0)
let zeros = ndimx (1,0)
let square = ndimx (2,0)
let cube = ndimx (3,0)

let rec zeros = 0::zeros
let rec matrix = zeros::matrix
let rec cube = matrix::cube


let rec get arg = match arg with
| 0, el::list -> el
| x, el::list -> get (x-1,list)

let g1 = get (3,zeros)

let rec put arg = match arg with
| el,0, loc::list -> el::list
| el,x, loc::list -> loc::(put (el,x-1,list))

let p1 = put (1,3,zeros)

let rec index arg = match arg with
| tofind,loc, el::list -> if tofind=el then loc else index(
    tofind, loc+1, list)

let i1= index (1,0,p1)

let rec get2 arg = match arg with
| 0,y, el::column::row -> get (y,column)
| x,y, column::row -> get2 (x-1,y,row)

let g3 = get2 (2,3,matrix)

let corec[constructor] make_list arg = arg::(make_list arg)

let rec put2 arg = match arg with
| el,0,y, loc::column::row -> (put (el,y,column))::row
| el,x,y, column::row -> column::(put2 (el,x-1,y,row))

let p2 = put2 (1,2,3,matrix)
let pp1 = put2 (5,0,1,matrix)
let pp2 = put2(5,1,0,matrix)

let corec[constructor] changecolumn arg = match arg with
| changeto, 0, column::row -> (make_list changeto)::row
```

```
| changeto, at, column::row -> column::(changecolumn(changeto,at
    -1,row))

let cc0 = changecolumn (1,0,matrix)
let a = hd cc0
let b = hd (tl cc0)
let cc1 = changecolumn (1,2,matrix)

let corec[constructor] changerow arg = match arg with
| changeto, at, column::row -> (put (changeto,at,column))::(
    changerow (changeto,at,row))

let cr1 = changerow (1,2,matrix)

(*gekopieerd uit list.ml*)
let corec[constructor] map arg = match arg with
  f, [ ] -> [ ]
| f, h :: t -> (f h) :: map(f,t)

let corec[constructor] map2 arg = match arg with
| func, column::row -> (map (func, column))::(map2 (func,row))

let rec getn arg = match arg with
| [coor], list -> get(coor,list)
| 0::coors, dim::mat -> getn (coors,dim)
| coor::coors, dim::mat -> getn ((coor-1)::coors, mat)

let gn1 = getn ([3],p1)
let gn2 = getn ([2;3],p2)

let rec putn arg = match arg with
| el, [coor], list -> put (el,coor,list)
| el, 0::coors, dim::mat -> (putn (el, coors,dim))::mat
| el, coor::coors, dim::mat -> dim::(putn (el, (coor-1)::coors,
    mat))

let pn1 = putn (1,[3],zeros)
(*let pn2 = putn (1,[2;3],matrix)
let pn3 = putn (1, [2;3;4], cube)

let gn3 = getn ([2,3,4], cube)*)
(*
let corec[constructor] changedims arg = match arg with
| changeto, [coor],  [] -> []
| changeto, [coor], list ->
        if coor=-1
then ndimx (1,changeto)
 else put (changeto,coor,list)
| changeto, 0::coors, dim::mat -> (changedims (changeto, coors,
    dim))::mat
| changeto, coor::coors, dim::mat ->
        if coor=-1
        then (changedims (changeto, coors, dim))::(changedims (
            changeto, (-1::coors), mat))
```

```
            else dim::(changedims (changeto, (coor-1)::coors,mat))
*)
```