KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

De Tabulatie Monad in Haskell

Alexander Vandenbroucke

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

Promotor:
Prof. dr. ir. Tom Schrijvers

Assessor:
Prof. dr. ir. Maurice Bruynooghe,
Prof. dr. Bart Demoen

Begeleider:
Prof. dr. ir. Tom Schrijvers

Academiejaar 2014 – 2015

Voorwoord

Deze masterproef is het sluitstuk van mijn opleiding tot informaticus. Het is mijn hoop dat ik U, de lezer, op een helder manier kan bijbrengen wat deze masterproef inhoudt.

De inhoud of het onderwerp van deze masterproef betreft de "Tabulatie Monad in Haskell". In alle eerlijkheid moet ik bekennen dat mij aanvankelijk slechts twee van deze termen bekend waren, namelijk "Haskell" en in die context "monad" (als de lezer met geen enkele bekend is, treft hem dus geen enkele blaam). Laat "Tabulatie" nu net de belangrijkste van de drie zijn.

Gelukkig was mijn promotor prof. Schrijvers paraat om dit onevenwicht te herstellen. Zonder zijn geduldige begeleiding en sturing zou het eindresultaat ongetwijfeld veel armer ogen. Bovendien kon ik dankzij zijn thesisvoorstel ontdekken hoezeer functioneel programmeren me interesseert.

Daarnaast ben ik ook al alle andere professoren van de afgelopen 5 jaar dank verschuldigd, van Inleiding Programmeren in eerste Bachelor—in Oberon, vandaar misschien mijn interesse in enigszins alternatieve programeertalen—tot en met Computational Logic in het allerlaatste semester.

Tenslotte hoop ik dat mijn masterproef niet slechts het sluitstuk zal zijn, maar ook de eerste steen aan mijn doctoraat.

Alexander Vandenbroucke
17 Mei 2015

Inhoudsopgave

Voorwoord **i**

Samenvatting **iv**

1 Inleiding **1**
1.1 Over deze thesis . . . 1
1.2 Onderzoeksaanpak . . . 3
1.3 Literatuur . . . 5

2 Preliminaries **7**
2.1 Haskell . . . 7
2.2 Tabulatie in Prolog . . . 8
2.3 Open Recursie . . . 9
2.4 Effect Handlers . . . 11
2.5 Denotationele Semantiek . . . 14

3 Syntax voor Tabulatie in Haskell **19**
3.1 Concrete syntax: niet-determinisme en recursie . . . 20
3.2 Abstracte syntax . . . 20
3.3 Een getabled programma . . . 23
3.4 Polymorfisme afdwingen . . . 24
3.5 Samenvatting . . . 26

4 Denotationele Semantiek **27**
4.1 Opbouw van de Denotationele Semantiek . . . 28
4.2 Vergelijking met Tabled Prolog . . . 29
4.3 Uitvoerbaarheid . . . 31
4.4 Conclusie . . . 36

5 Aggregaten **37**
5.1 Tralies als aggregaten . . . 38
5.2 De semantische functie . . . 41
5.3 Vergelijking met Prolog . . . 43
5.4 Uitvoerbaarheid . . . 44
5.5 Voorbeelden . . . 45
5.6 Conclusie . . . 53

6 Benchmarks **55**
6.1 Knapzak . . . 55

6.2 Strongly Connected Components 57
6.3 Kortste pad 60
6.4 Conclusie 62

7 Verder Werk en Besluit **63**
7.1 Besluit 63
7.2 Verder werk 64

A Bewijzen van monotoniciteit voor denotationele semantiek **67**
A.1 Monotoniciteit 67
A.2 Continuiteit 69

B Bewijs Theorema 3 **73**

C Wetenschappelijke Paper **75**

D Poster **85**

Bibliografie **87**

Samenvatting

Niet-deterministische functies zijn functies die nul, één of meer resultaten terug kunnen geven. Een *recursieve* niet-deterministische functie berekent op basis van een eerste resultaat een ander resultaat. Hiermee kan opnieuw een resultaat worden berekend. Als dit laatste resultaat hezelfde is als het eerste resultaat dan stopt de recursie niet. De combinatie van niet-deterministische en recursie kan dus leiden tot een oneindige lus.

Tabulatie is een techniek die oorspronkelijk ontstaan is in het logisch programmeren. Logische programmeertalen zoals Prolog hebben inherent niet-determinisme. Een recursief predicaat in Prolog kan dus op hetzelfde probleem stuiten. Tabulatie onthoudt de reeds geziene oplossingen en stopt het programma wanneer er geen nieuwe resultaten meer bijkomen. Het voorkomt dus het voornoemde probleem. In deze thesis bestuderen we hoe de tabulatie-techniek in Haskell kan worden geïmplementeerd.

In Haskell modelleren we Tabulatie met een monad. Deze thesis construeert de Tabulatie Monad met behulp van de "Effect-Handlers"-aanpak. Het idee van deze aanpak is om de de definitie van een monad op te splitsen in twee delen. Eerst wordt een abstracte syntax geformuleert die alle nodige operaties (i.e. niet-determinisme en recursie) ondersteunt. In Haskell neemt dit de vorm aan van een algebraïsch datatype. Hierbovenop voorzien we een gebruiksvriendelijke interface voor de programmeur. De tweede stap bestaat uit het definiëren van effect handlers. Dit zijn functies die een abstracte syntaxboom afbeelden op een bepaald effect, in casu Tabulatie. De effect handler wordt eerst puur abstract wiskundig onderzocht. Vervolgens vertalen we de effect handler naar een Haskell functie.

Voor de effect handler bekijken we verschillende punten in de ontwerpruimte. In één richting breiden we de effect handler uit met een afhankelijkheidsanalyse. Dit verbetert de performantie aanzienlijk. In een tweede (orthogonale) richting ondersteunen we aggregaten. Met aggregaten kunnen gemakkelijker de optimale oplossing van een niet-deterministisch programma berekenen (bijvoorbeeld voor Dynamisch Programmeren). Het voordeel van aggregaten is tweevoudig. Enerzijds kunnen we bepaalde programma's eenvoudiger uitdrukken. Anderzijds wordt ook hier de performantie beter omdat alleen de optimale oplossing wordt berekend. Tenslotte evalueren we deze resultaten nog met benchmarks voor enkele klassieke problemen.

Hoofdstuk 1

Inleiding

1.1 Over deze thesis

Eindigt het volgende Haskell programma?

$$p :: [(Int, Int)]$$
$$p = (1,2) : [(y,x) \mid (x,y) \leftarrow p]$$

$$\ggg p$$
$$[(1,2),(2,1),(1,2),(2,1),...]$$

Eindigt het volgende equivalente Tabled Prolog programma?

```
:- table p/2.
p(1,2).
p(Y,X) :- p(X,Y).

?- p(A,B).
A = 2
B = 1;

A = 1
B = 2;

no.
```

Het eerste voorbeeld geeft een oneindige lijst van antwoorden, hoewel er maar twee verschillende antwoorden mogelijk zijn ($(1,2)$ en $(2,1)$). Het tweede voorbeeld geeft maar twee antwoorden (`A=1,B=2` en `A=2,B=1`) en stopt dan. Dit gedrag wordt veroorzaakt door de `:- table p/2` directive, die aanduidt dat *tabulatie* moet gebruikt worden om het predicaat `p/2` op te lossen.

Tabulatie [23] is een techniek die oorspronkelijk ontstaan is in het logisch programmeren. Logische programma's bestaan uit *clauses* die relaties tussen variabelen en constanten vastleggen in predicaten. Berekening vloeit voort uit het oplossen

van *goals*. Dit zijn uitdrukkingen die bestaan uit een predicaat, constanten en variabelen. Het oplossen van de goal houdt in dat Prolog (of een andere logische programmeertaal) op zoek gaat naar concrete waarden voor de variabelen in de goal die niet strijdig zijn met de clauses.

Door het tussentijds onthouden van oplossingen maakt tabulatie het mogelijk om Prolog programma's zoals hierboven sneller te berekenen of eindeloze herhaling van dezelfde oplossingen te voorkomen.

Haskell is een moderne, sterk getypeerde en pure functionele programmeertaal. Haskell wordt vaak gebruikt voor onderzoek naar programmeertalen, en kent een bescheiden, maar groeiende populariteit in de industrie.

Kunnen we op een informele manier verklaren waarom het Haskell programma oneindig veel oplossingen produceert? Intuïtief ontstaat dit gedrag omdat recursie en niet-determinisme samen voorkomen. Het spreekt voor zich dat een (functioneel) programma dat geen recursie bevat niet oneindig lang kan blijven lopen. In dit concrete geval is het de functie p die zichzelf oproept.

Met niet-determinisme bedoelen we hier dat één uitdrukking of functie nul, één of meerdere resultaten kan hebben, bijvoorbeeld door een lijst van oplossingen terug te geven. Het eerste antwoord van p is $(1, 2)$ (het eerste element in de lijst). Vervolgens kan p recursief $(2, 1)$ afleiden. Vanuit $(2, 1)$ wordt dan opnieuw $(1, 2)$ berekend, enzovoort. Als we onbegrensde recursie toelaten, samen met niet-determinisme, dan kan het programma dus ontsporen.

Prolog heeft inherent niet-determinisme: het oplossen van een Prolog goal kan meerdere oplossingen (geldige substituties van variabelen) hebben. Als in het voorbeeld de `table`-regel zou ontbreken, dan zou de oneindige reeks oplossingen `A=1,B=2; A=2,B=1; A=1,B=2; ...` berekend zijn. Met andere woorden, het Prolog programma zou aan hetzelfde kwaal hebben geleden als het Haskell programma.

> Als Tabulatie dit euvel in Prolog kan verhelpen, dan moet het mogelijk zijn om met Tabulatie hetzelfde te doen voor niet-determinisme in Haskell

Dit is het kernidee van deze thesis. Het vervolg van dit hoofdstuk bespreekt de aanpak van Tabulatie in Haskell in meer detail en overloopt de relevante literatuur en bestaand werk. Hoofdstuk 2 overloopt de voorkennis die voor deze thesis relevant is, waaronder een uitgebreide uiteenzetting over de "Effect Handlers" aanpak.

De belangrijkste bijdragen van deze thesis zijn:

- Een abstracte interface voor het schrijven van recursieve niet-deterministische programma's. Deze interface bestaat uit de `Nondet`-typeclass en open recursie. De interface laat genoeg vrijheid om Tabulatie op een dergelijke manier te implementeren dat oneindige recursie een halt kan worden toegeroepen (Hoofdstuk 3, Sectie 3.1).

- Een implementatie van de *Tabulatie Monad* die de hiervoor beschreven interface ondersteunt. Deze implementatie is gebaseerd op de "Effect Handlers"-aanpak [24]. Deze aanpak splitst het definiëren van een monad in twee taken:

1. Een abstracte syntax die alle ondersteunde operaties beschrijft (Hoofdstuk 3, Sectie 3.2 en verder). Concreet gaat het over de datatypes *Program* en *Table*.
2. De effect handler die een effect (bijvoorbeeld Tabulatie) toewijst aan de abstracte syntax (Hoofdstuk 4). In de eerste plaats wordt deze effect handler op een abstracte wiskundige manier gespecificeerd. Op die manier krijgt de abstracte syntax een *denotationele, monadische semantiek*. Samen vormen de syntax en semantiek de *Tabulatie Monad*. Deze monad gedraagt zich zoals Tabled Prolog.

   Ten tweede kan deze semantiek (bijna letterlijk) vertaald worden in Haskell. Zodoende ontstaat een eerste naïeve implementatie van deze tabulatie monad in Haskell.

- Hoofdstuk 4, Sectie 4.3.1 bespreekt eveneens een veel efficiëntere effect handler die gebruik maakt van afhankelijkheidsanalyse om snelheidswinst te boeken.
- Hoofdstuk 5 breidt de semantiek verder uit naar aggregaten. Aggregaten laten toe om bepaalde programma's efficiënter uit te voeren, zonder dat het programma zelf moet veranderen. In tegenstelling tot de afhankelijkheidsanalyse die hetzelfde resultaat berekent als de naïeve effect handler, kunnen sommige programma's met behulp van aggregaten compacter geschreven worden.
- Het nut van de eerder besproken optimalisaties (afhankelijkheidsanalyse en aggregaten) wordt empirisch aangetoond met benchmarks in Hoofdstuk 6.
- De code in deze tekst staat grotendeels op zichzelf. Op enkele plaatsten kan het nuttig zijn om extra codebestanden te raadplegen. Deze zijn online beschikbaar in een git-repository op bitbucket. [1]

1.2 Onderzoeksaanpak

Om te voorkomen dat niet-deterministische recursie ontspoort laten we ons inspireren door Prolog. Het nabootsten van Prolog lag namelijk aan de oorsprong van de interesse in niet-determinisme in Haskell [11]. Naar analogie moet het eveneens mogelijk zijn om Tabled Prolog na te bootsten met getabled niet-determinisme.

Niet-determinisme, gemodelleerd met lijsten van oplossingen [21], heeft een *Monad* instance:

$$
\begin{aligned}
&\textbf{instance } \mathit{Monad}\ [\,]\ \textbf{where}\\
&\quad \mathit{return}\ a = [\,a\,]\\
&\quad [\,] \gg\!= f = [\,]\\
&\quad (x : xs) \gg\!= f = (f\ x) +\!\!+ (xs \gg\!= f)
\end{aligned}
$$

Dit is de standaard monad instance voor lijsten [22]. Het is de bedoeling om Tabulatie ook met een monad te modelleren. Dit maakt het mogelijk om meerdere

[1] https://bitbucket.org/AlexanderV/thesis

getablede berekeningen op een gemakkelijke manier samen te stellen met de “bind”-operatie ($\gg=$).

Het ontwikkelen van een nieuwe monadische structuur die bovendien ook tabulatie implementeert en ook niet-determinisme en recursie ondersteunt is een moeilijk probleem dat veel creativiteit en inventiviteit vergt.

In plaats van dit probleem volledig van nul aan te pakken, maken we gebruik van de “Effect Handlers”-aanpak[24] om de monad te construeren.

Het idee van de effect handlers aanpak is om de creatie van de monad op te splitsen in twee fasen. Eerst wordt een *abstracte syntax* bedacht. Deze syntax is in essentie een ingebedde domeinspecifieke taal (Eng. Domain Specific Language, DSL). De syntax moet toelaten alle gewenste operaties kunnen beschrijven. In het bijzonder moeten niet-determinisme en recursie expliciet ondersteund worden. De abstracte syntaxboom van een dergelijke taal heeft altijd een eenvoudige monad instance.

Daarna definiëren we een *effect handler* die de abstracte syntaxboom afbeeldt op de gewenste opeenvolging van effecten. Deze aanpak splitst het bedenken van een monad dus op in een interface (de abstracte syntax) en een implementatie (de effect handler).

Dit proces kan eerst abstract uitgevoerd worden: we definiëren een grammatica die de abstracte syntax precies specificeert. Vervolgens formuleren we semantische functie die de syntax een precieze betekenis geeft. Pas hierna bekommeren we ons om de praktische details die eigen zijn aan het implementeren van de syntax en semantiek in Haskell.

In het abstracte geval concentreren we ons op correctheid: is het gewenste effect bereikt? In Haskell erven we deze correctheidseigenschap, zodat we ons met operationele eigenschappen zoals efficiëntie kunnen bezighouden. Het proces kan cyclisch zijn: performantievereisten die ontstaan in Haskell kunnen voldaan worden met formeel gecontroleerde maatregelen alvorens deze in Haskell worden geïmplementeerd.

Verschillende effect handlers kunnen verschillende effecten toewijzen aan dezelfde syntaxboom. Beschouw bijvoorbeeld een handler die een programma altijd verbindt met de verzameling van zijn oplossingen. Deze semantiek geeft hetzelfde resultaat as Tabled Prolog (modulo unificatie, negatie).

In plaats van steeds alle oplossingen in de verzameling te plaatsten kunnen we meer selectief zijn. We kunnen bijvoorbeeld enkel de optimale oplossing onthouden. Hierdoor zijn bepaalde aggregaten zoals de minimale of maximale oplossing efficiënter te berekenen.

Zelfs als exact hetzelfde effect berekend wordt, dan nog kunnen effect handlers verschillen in hun operationele eigenschappen: de ene effect handler is al sneller als de andere.

Dit zorgt ervoor dat we niet langer van één Tabulatie Monad kunnen spreken. Er is een hele familie van tabulatie monads, elk gerepresenteerd door hun effect handler. Deze monads kunnen we dan onderling vergelijken op basis van denotationele en operationele aspecten.

1.3 Literatuur

Tabulatie is reeds lang bekend als een manier om recursieve programma's te versnellen, in het bijzonder voor dynamisch programmeren. Bird [3] bespreekt een aantal implementatietechnieken voor verschillende soorten recursieve programma's, gebaseerd op de afhankelijkheidsgrafen van de recursieve oproepen.

Pareja-Flores, Peña en Velásquez-Iturbide beschrijven een programmatransformatie die een programmeur kan uitvoeren om de efficiëntie van zijn recursieve functie in Haskell te verbeteren door middel van Tabulatie [15]. Deze thesis probeert daarentegen om de last op de programmeur zoveel mogelijk te beperken. De programmeur moet geen expliciete transformaties doen, behalve dat het programma in een open recursieve stijl moet worden geschreven.

Omdat een goal in Prolog meerdere oplossingen kan hebben krijgt Tabulatie in Prolog een extra dimensie: Tabulatie beëindigt recursie op het moment dat alle mogelijke oplossingen gevonden zijn. Het (ongepubliceerde) boek "Programming in Tabled Prolog" [23] vormt een goede introductie tot Tabulatie in Prolog.

Sommige Tabled Prolog implementaties ondersteunen ook een vorm van aggregaten. In XSB-Prolog [20] gebeurd dit op een manier die sterk gelijkt op de manier die in deze thesis wordt gebruikt. Andere systemen gebruiken *mode-directed tabling* [6], [18], [25]. Sectie 5.3 gaat hier in meer detail op in.

Monads zijn een begrip uit *Category Theory*, een abstracte tak van de wiskunde. Het gebruik van monads om functionele programma's te structureren [22] is geïnspireerd door het gebruik van monads in denotationele semantiek. [14] Het gebruik ervan is inmiddels zeer sterk ingeburgerd in de Haskell-wereld, en in mindere mate (mogelijks in vermomming) in programmeren in het algemeen.

Effect Handlers zijn een actueel onderzoeksonderwerp [24], [16]. Ter vergelijking, Kiselyov, Shan, Friedman en Sabry [11] gebruiken de meer traditionele manier om monads (en monad transformers [10]) te construeren aan de hand van typeclasses.

Open recursie [4] is een techniek die toelaat om in te grijpen op de recursieve oproepen van een functie.

Scott en Strachey [19] gaven voor het eerst een formeel wiskundig fundament aan de denotationele semantiek van recursieve programma's, gesteund op tralies (Eng. lattices).

CSP[8] beschikt over een denotationele semantiek [5]. In CSP komen twee soorten niet-determinisme naar boven: enerzijds keuze uit verschillende "boolean-guards" die tegelijk true kunnen zijn, en minder interessant voor deze thesis, niet-determinisme gegenereerd door concurrente processen. Het wiskundige model in deze thesis is echter veel minder complex.

Deze thesis is lang niet de enige poging om aspecten van logische en functionele programmeertalen te verenigen. Functionele logische programmeertalen zoals Curry [2] bevatten elementen van beide. Curry is sterk geïnspireerd op Haskell en ondersteunt bijna alle Haskell syntaxconstructies. Daarnaast ondersteunt Curry ook de operator *?* die een niet-deterministische keuze voorstelt, of de definitie van zogenaamde "free" variabelen, die op een niet-deterministische manier worden ingevuld. Curry gebruikt geen tabulatie en lijdt daarom aan hetzelfde kwaal als Haskell en

Prolog: de recursieve oproepen in een niet-deterministische functie kunnen in een oneindige lus terechtkomen.

Hoofdstuk 2

Preliminaries

Het doel van dit hoofdstuk is om de lezer voldoende voorkennis te geven om de onderwerpen in het vervolg van deze tekst te kunnen begrijpen.

2.1 Haskell

Er bestaan veel goede inleidingen en tutorials voor Haskell. A gentle Introduction to Haskell[9] en Learn You a Haskell [12] zijn twee favorieten van de auteur.

Het doel van deze sectie is niet om een algemene introductie te geven tot Haskell. In plaatst daarvan is de bedoeling om te verzekeren dat de auteur en de lezer op dezelfde lijn zitten met betrekking tot een aantal concepten.

2.1.1 Monads

Monads vormen een standaard manier om effectvolle berekeningen te encapsuleren in functionele programmeertalen [22]. Effectvolle berekeningen zijn berekeningen die naast hun resultaat ook extra effecten kunnen hebben. Toestand, exceptions, niet-determinisme en (in deze tekst) Tabulatie zijn allemaal voorbeelden van effecten. Deze berekeningen worden samengesteld met een functie $\gg=$ (uitgesproken als "bind"). In Haskell beschrijft men monads met een typeclass:

$$
\begin{aligned}
&\textbf{class } \mathit{Monad}\ m \textbf{ where} \\
&\quad \mathit{return} :: a \to m\ a \\
&\quad (\gg=)\ \ :: m\ a \to (a \to m\ b) \to m\ b
\end{aligned}
$$

Het type $m\ a$ beschrijft berekeningen van het type m die resultaten van het type a heeft. *return* tilt een waarde a op naar een effectvolle berekening van type m die alleen a produceert en geen andere effecten heeft. bind $(\gg=)$ voegt een berekening van het type $m\ a$ die een waarde van type a produceert samen met een functie die een a accepteert en een berekening van type $m\ b$ teruggeeft.

Om geldig te zijn moet een monad instance voldoen aan de volgende wetten:

$$
\begin{array}{lll}
return\ x \gg\!= f & = f\ x & \text{-- Linker neutraal element} \\
m \gg\!= return & = m & \text{-- Rechter neutraal element} \\
(m \gg\!= f) \gg\!= g & = m \gg\!= \lambda x \rightarrow (f\ x \gg\!= g) & \text{-- Associativiteit}
\end{array}
$$

Omdat monads in Haskell zeer vaak gebruikt worden bestaat een speciale **do**-notatie, die het aaneenrijgen van binds eenvoudiger maakt: **do** $\{x \leftarrow f; g\}$ is syntactic sugar voor $f \gg\!= \lambda x \rightarrow g$.

2.1.2 Monoids

Monoids zijn een eenvoudige structuur die twee operaties aanbied: ε het lege element, en *mappend* (de operator $\diamond$ wordt vaak als synonym gebruikt) die twee monoids samenvoegt.

In Haskell beschrijft men Monoids eveneens met een typeclass:

$$
\begin{array}{l}
\textbf{class}\ Monoid\ m\ \textbf{where} \\
\quad \varepsilon \qquad\quad :: m \\
\quad mappend :: m \rightarrow m \rightarrow m
\end{array}
$$

Om geldig te zijn moeten Monoids ook 3 wetten respecteren:

$$
\begin{array}{lll}
\varepsilon \diamond a & = a & \text{-- Linker neutraal element} \\
a \diamond \varepsilon & = a & \text{-- Rechter neutraal element} \\
(a \diamond b) \diamond c & = a \diamond (b \diamond c) & \text{-- Associativiteit}
\end{array}
$$

Als de operatie *mappend* commutatief of idempotent is, dan spreekt men van een commutatieve of idempotente monoid.

2.2 Tabulatie in Prolog

De Tabulatie die in dit werk wordt gepresenteerd is geïnsprireerd door de Tabulatie in Prolog. Het is daarom nuttig om een eerste introductie tot Tabulatie in Prolog te geven. De lezer kan dan parallellen en contrasten tussen Prolog en Haskell beter kaderen. Voorkennis van Prolog-syntax is niet nodig, aangezien Prolog nauwelijks syntax heeft. Voor een diepgaandere introductie, zie [23] en de XSB Prolog manual. [1]

Het volgende programma definieert een predicaat `p` met twee argumenten. De eerste regel definiëert het feit `p(1,2)`, namelijk dat predicaat `p` waar is voor argumenten 1 en 2. De volgende regel definiëert een clause, die we als een omgekeerde implicatie kunnen interpreteren: als `p(Y,X)` waar is, dan is `p(X,Y)` ook waar.

```
p(1,2).
p(X,Y) :- p(Y,X).
```

[1] http://xsb.sourceforge.net/manual1/manual1.pdf

Er bestaan dus 2 mogelijke toewijzingen aan `A` en `B` die de goal `?-p(A,B)` waar maken: `A=1,B=2` en `A=2,B=1`. Maar zelfs op dit uiterst simpele programma gaat een gewone Prolog interpreter de mist in: de uitvoering stopt niet hoewel er geen nieuwe resultaten bijkomen!

```
?-p(A,B).
    A=1  B=2;
    A=2  B=1;
    A=1  B=2;
    A=2  B=1;
    ...
```

Het volgende programma toont hoe dit euvel met Tabulatie te verhelpen is. Aan het begin van het programma wordt een `table p/2.`-directive toegevoegd. Een Prolog interpreter die dit ondersteunt zal de nu de resultaten van `p` tabuleren.

```
:- table p/2.
p(1,2).
p(X,Y) :- p(Y,X).
```

De uitvoer wordt nu:

```
?-p(A,B).
    A=1  B=2;
    A=2  B=1;
    no.
```

Prolog merkt nu dat extra pogingen geen nieuwe oplossingen hebben en stopt. Sterker nog, Tabulatie kan ook links-recursieve programma's oplossen (links-recursief verwijst naar de Prolog-uitvoeringsboom, die clauses van links naar rechts diepte-eerst uitprobeert, en dus in een linkse recursie vast kan komen te zitten, zonder oplossingen te produceren).

```
:- table p/2.
p(X,Y) :- p(Y,X).
p(1,2).
```

Dit programma geeft hetzelfde resultaat als het vorige programma. Twee elementen die Tabulatie moeilijk, en dus interessant maken zijn in dit programma al aanwezig. Enerzijds moet een recursief predicaat getabled worden. Anderzijds heeft de goal die het predicaat oproept duidelijk meerdere (maar een eindig aantal) oplossingen. Het berekenen van een goal is dus niet-deterministisch, in de zin dat deze berekening meerdere oplossingen heeft.

2.3 Open Recursie

Open recursie [4] is een manier om functies op een zodanige manier te structureren dat de interne recursieve structuur open ligt. Dit maakt het gedrag van recursieve

oproepen uitbreidbaar. In dit werk wordt hiervan gebruik gemaakt om Tabulatie op recursieve oproepen te implementeren.

De volgende Haskell code definiëert open recursie:

```
type Open s = s → s
zero :: Open a
zero = id

new :: Open s → s
new a = let this = a this in this

(⊕) :: Open s → Open s → Open s
a1 ⊕ a2 = λsuper → a1 (a2 super)
```

Het basisgeval is *zero* en doet niets. Een open recursieve functie *a2* kan uitgebreid worden door een andere open recursieve functie *a1* met de ⊕-combinator: *a1* ⊕ *a2* is een nieuwe open recursieve functie waarin het gedrag van *a2* uitgebreid is met dat van *a1*. Een open recursieve functie *f* wordt afgesloten door `new`: *new f* is een gewone functie en kan als dusdanig gebruikt worden.

Een eenvoudig voorbeeld, overgenomen uit [4] is het volgende:

```
fib :: Open (Int → Int)
fib this n = case n of
  0 → 0
  1 → 1
  _ → this (n − 1) + this (n − 2)
```

fib n is een recursieve berekening van het n-de fibonaccigetal waarin de recursieve oproepen vervangen zijn door *this*.

De volgende functie *optfib* implementeert twee hardgecodeerde gevallen, om de implementatie te "versnellen". *optfib* kan duidelijk niet op zichzelf gebruikt worden voor andere waarden dan 10 en 30.

```
optfib :: Open (Int → Int)
optfib super n = case n of
  10 → 55
  30 → 8320140
  _  → super n
```

De functies kunnen na combinatie met ⊕ gesloten worden met *new* en zijn dan klaar voor gebruik.

```
slowfib :: Int → Int
slowfib = new fib
fastfib  :: Int → Int
fastfib  = new (optfib ⊕ fib)
```

2.4 Effect Handlers

De Effect Handlers aanpak splitst het modelleren van Effecten op in twee delen. De eerste stap definiëert een syntax die alle relevante acties bevat. De tweede stap definiëert een functie die deze syntax omzet naar de gewenste effecten.

De syntax is vaak een abstracte syntax tree datatype, waarvoor gemakkelijk een monad instance kan worden gedefiniëerd.

In navolging van de paper "Effect Handlers in Scope" [24] beschouwen we backtrackende berekeningen als een eenvoudig voorbeeld.

De volgende typeclass is een abstractie van backtracking:

$$
\begin{aligned}
&\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{Nondet}\ m \textbf{ where}\\
&\quad \mathit{fail} :: m\ a\\
&\quad (\vee) :: m\ a \to m\ a \to m\ a
\end{aligned}
$$

De functie $\vee$ stelt een niet-deterministische keuze voor en *fail* een mislukte berekening. De constraint *Monad m* verzekert dat iedere instance van *Nondet* ook een instance is van *Monad*.

Het voorbeeld uit Sectie 2.2 laat zich nu in Haskell schrijven:

$$
\begin{aligned}
&p :: \mathit{Nondet}\ m \Rightarrow m\ (\mathit{Int}, \mathit{Int})\\
&p = \mathit{return}\ (1,2) \vee \textbf{do}\ (x,y) \leftarrow p\\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ (y,x)
\end{aligned}
$$

De keuze tussen de twee clauses in het Prolog voorbeeld is nu een keuze tussen twee backtrackende berekeningen verbonden met $\vee$.

De volgende twee instances voor *Monad* en *Nondet* interpreteren een backtrackende berekening als een lijst van alle mogelijke oplossingen:

$$
\begin{aligned}
&\textbf{instance } \mathit{Monad}\ [\,] \textbf{ where}\\
&\quad \mathit{return}\ a = [a]\\
&\quad [\,] \qquad \gg\!= \_ = [\,]\\
&\quad (x : xs) \gg\!= f = f\ x +\!\!+ (xs \gg\!= f)\\
&\textbf{instance } \mathit{Nondet}\ [\,] \textbf{ where}\\
&\quad \mathit{fail} \quad = [\,]\\
&\quad a \vee b = a +\!\!+ b
\end{aligned}
$$

Deze *Monad* instantie is de standaard instance voor lijsten, men noemt dit daarom de *lijst-monad*. Om Haskell de juiste instances te laten kiezen voegen we een typesignatuur toe: $p :: [(\mathit{Int}, \mathit{Int})] = [(1,2),(2,1),...]$, dus de berekening is eveneens oneindig, net zoals in Prolog. In het algemeen een oplossing van de vorm $[(1,2),(2,1)]$ kunnen bekomen (zoals Tabled Prolog) vormt het onderwerp van deze thesis en daar komt deze tekst later uitgebreid op terug.

Nu gaan we over op de Effect Handlers aanpak. De syntax voor Backtracking is de volgende:

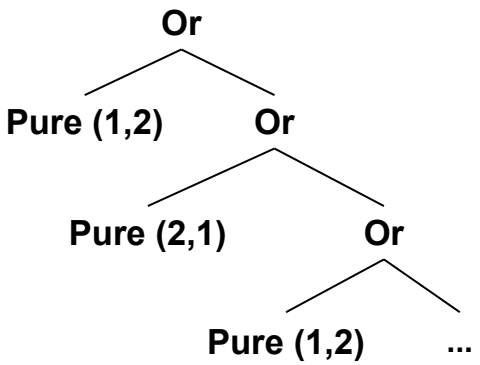
$$\begin{aligned}&\textbf{data } \mathit{Backtrack}\ a = \mathit{Pure}\ a\\ &\qquad\qquad\quad \mid \mathit{Fail}\\ &\qquad\qquad\quad \mid \mathit{Or}\ (\mathit{Backtrack}\ a)\ (\mathit{Backtrack}\ a)\end{aligned}$$

Pure a stelt een succesvolle berekening met resultaat a voor, *Fail* stelt een mislukte berekening voor, en *Or l r* is een keuze tussen l of r.

De voor de hand liggende *Monad*- en *Nondet*-instances zijn:

$$\begin{aligned}&\textbf{instance } \mathit{Monad}\ \mathit{Backtrack}\ \textbf{where}\\ &\quad \mathit{return}\ a = \mathit{Pure}\ a\\ &\quad (\mathit{Pure}\ a) \gg\!= f = f\ a\\ &\quad \mathit{Fail} \qquad \gg\!= \_ = \mathit{Fail}\\ &\quad (\mathit{Or}\ l\ r) \gg\!= f = \mathit{Or}\ (l \gg\!= f)\ (r \gg\!= f)\\ &\textbf{instance } \mathit{Nondet}\ \mathit{Backtrack}\ \textbf{where}\\ &\quad \mathit{fail} \quad = \mathit{Fail}\\ &\quad a \vee b = \mathit{Or}\ a\ b\end{aligned}$$

Het resultaat van $p :: \mathit{Backtrack}\ (\mathit{Int}, \mathit{Int})$ is nu een abstracte syntaxboom van de vorm: $\mathit{Pure}\ (1,2)\ `\mathit{Or}`\ (\mathit{Pure}\ (2,1)\ `\mathit{Or}`\ (\mathit{Pure}\ (1,2)\ `\mathit{Or}`\ ...))$, of grafisch in Figuur 2.1



Figuur 2.1: De abstracte syntaxboom die overeenkomt met $p :: \mathit{Backtrack}\ (\mathit{Int}, \mathit{Int})$.

We kunnen nu een interpreter bedenken die alle mogelijke oplossingen in een lijst teruggeeft (net zoals de instances voor $[\,]$).

$$\begin{aligned}&\mathit{solutions} :: \mathit{Backtrack}\ a \to [a]\\ &\mathit{solutions}\ (\mathit{Pure}\ a) = [a]\\ &\mathit{solutions}\ \mathit{Fail} \qquad = [\,]\\ &\mathit{solutions}\ (\mathit{Or}\ l\ r) = \mathit{solutions}\ l +\!\!+ \mathit{solutions}\ r\end{aligned}$$

Het effect van p is dan $\mathit{solutions}\ p = [(1,2),(2,1),...]$, een oneindige lijst van oplossingen. Een voorbeeld van een triviale (en dus niet algemene) uitbreiding die wel $[(1,2),(2,1)]$ geeft is de volgende interpreter:

$$\begin{aligned}&\mathit{nsolutions} :: \mathit{Int} \to \mathit{Backtrack}\ a \to [a]\\ &\mathit{nsolutions}\ n = \mathit{take}\ n \circ \mathit{solutions}\end{aligned}$$

Want $\mathit{nsolutions}\ 2\ p = [(1,2),(2,1)]$.

Andere interpreters In de vorige paragrafen werden al twee mogelijke effecten besproken: alle oplossingen, of alleen de eerste twee oplossingen. In dit specifieke geval is het vrij eenvoudig om het nieuwe effect voor p te definiëren op basis van de lijst-monad (namelijk, $take\ 2\ p = [(1,2),(2,1)]$).

In het volgende voorbeeld is dit niet zo eenvoudig. *leftBranch* is een interpreter die altijd kiest voor de linkse optie.

```
leftBranch :: Backtrack a → [a]
leftBranch (Pure a) = [a]
leftBranch Fail     = []
leftBranch (Or a _) = leftBranch a
```

Het nieuwe resultaat is $leftBranch\ p = [(1,2)]$. De uitdrukking $Fail\ `Or`\ p$ illustreert het verschil tussen *leftBranch* en *solutions*: $leftBranch\ (Fail\ `Or`\ p) = [\,]$, terwijl $solutions\ (Fail\ `Or`\ p) = [(1,2),(2,1),...]$). De linkse keuze (en alleen de linkse keuze) wordt altijd genomen, zelfs als dit een mislukking is.

In de lijst-monad gaat dit echter niet, want in de *Nondet* instance verliezen we informatie wanneer we $\vee$ gebruiken. Met andere woorden, er is een nieuwe *Nondet* instance nodig.

```
instance Nondet [] where
  fail  = []
  a ∨ _ = a
```

Maar hier ontstaat in Haskell een probleem: er kan niet meer dan één instance van een typeclass gedefiniëerd worden per type.

Er is dus geen andere keuze dan over te schakelen op een ander type *Left*.

```
data Left a = Empty | Value a
instance Monad Left where
  return = Value
  Empty   >>= _ = Empty
  (Value a) >>= f = f a
instance Nondet Left where
  fail  = Empty
  a ∨ _ = a
```

Het effect is nu zoals *leftBranch*: $p :: Left\ (Int, Int) = Value\ (1,2)$ en $fail \vee p :: Left\ (Int, Int) = Empty$.

Deze uiteenzetting illustreert het voordeel van de Effect Handlers aanpak tegenover de typeclasses en instances aanpak. Iedere keer als een ander effect gewenst is moet men op zoek gaan naar een ander datatype dat dit effect kan ondersteunen, en daarvoor de nodige *Monad*- en *Nondet*-instances definiëren. In deze simpele voorbeelden valt dit nog goed mee, maar voor complexe effecten (waarvan Tabulatie zeker een voorbeeld is) wordt het als maar moeilijker om correcte instances te vinden. Bovendien is in deze sectie het controleren van de monad wetten onder de mat

geveegd. Hoe complexer de effecten en structuren, hoe moeilijker het wordt om dit na te gaan. Toch zou een echte applicatie dit moeten doen.

Bij de Effect Handlers aanpak kan men gewoon overstappen op een andere interpreter. Het voldoen aan de monad wetten valt nu ten deel aan de interpreters in plaats van de syntax.

Tenslotte wil de auteur nog opmerken dan beide technieken valabel zijn, en nuttige monads kunnen opleveren. Zie [11] als voorbeeld van de aanpak gebaseerd op typeclasses.

2.5 Denotationele Semantiek

Het doel van een denotationele semantiek is om een programma te voorzien van een eenduidige formele betekenis. Deze betekenis bestaat uit operaties op objecten uit een wiskundig domein.

Concreet moet een denotationele semantiek dus een mapping voorzien tussen een programmasyntax en wiskundig objecten.

In deze tekst neemt de semantiek de vorm aan van een functie die een syntaxboom datatype afbeeld op het wiskundige domein.

Bijvoorbeeld, de volgende grammatica voor een simpele expressietaal bestaat enkel uit gehele getallen en plus:

$$Expr ::= \texttt{plus}\ \ Expr\ Expr \mid \texttt{num}\ \ \mathbb{Z}$$

Met als semantische functie $\mathcal{E}[\![\,.\,]\!]$ van $Expr$ naar de gehele getallen:

$$\mathcal{E}[\![\,.\,]\!] : Expr \to \mathbb{Z}$$
$$\mathcal{E}[\![\,e\,]\!] = \begin{cases} a & \text{als } e = \texttt{num}\ \ a \\ \mathcal{E}[\![\,a\,]\!] + \mathcal{E}[\![\,b\,]\!] & \text{als } e = \texttt{plus}\ \ a\ b \end{cases}$$

De semantiek ontstaat dus door de grammatica productie per productie te analyseren. In deze tekst wordt de grammatica meestal gespecificeerd als algebraïsch datatype in Haskell, met één constructor per productie:

data *Expr* = *Plus Expr Expr* | *Num Int*

De semantische functie is dan een gewone Haskell functie, die pattern matching doet op de verschillende constructors.

2.5.1 Vaste punten en tralies

Deze sectie geeft theoretische achtergrond bij Hoofdstukken 4 en 5. Ook zonder deze sectie te hebben doorgenomen kan de lezer de denotationele semantiek in voorgenoemde hoofdstukken te begrijpen. Deze sectie kan dan opnieuw bezocht worden als tijdens het lezen van deze hoofdstukken meer achtergrond gewenst is.

Inleiding

Als een punt niet verandert nadat er een functie op wordt uitgevoerd, dan noemen we dit een vast punt van die functie:

Definitie 1 (Vast Punt)**.** *Een punt x is een vast punt (eng. fixed point of fixpoint) van een functie f als $x = f(x)$.*

Het vervolg van deze sectie draait rond het berekenen van dergelijke vaste punten. Meer specifiek, het doel is om het *kleinste* van alle vaste punten te vinden. In de eerste plaats moet getoond worden dat vaste punten van een bepaalde functie bestaan, en dat één van hen kleiner is dan alle andere. In de tweede plaats kunnen we demonstreren hoe dit kleinste vaste punt geconstrueerd kan worden.

Bestaan van het kleinste vaste punt

Een geordende verzameling is een volledige tralie als voor elke deelverzameling een element kan worden aangeduid dat "groter" is dan alle elementen in de verzameling (dus ook de lege deelverzameling).

Bijvoorbeeld, de machtsverzameling van een verzameling A vormt een volledige tralie: ze is geordend door deelverzameling-inclusie $\subseteq$, en van elke verzameling van deelverzamelingen van A kunnen we altijd een "groter" element aanduiden door de unie van de deelverzamelingen te nemen.

Definitie 2 (Volledige Tralie)**.** *Een volledige tralie (eng. complete lattice) $(L, \sqsubseteq, \sqcup)$ is een partieel geordende verzameling $(L, \sqsubseteq)$ zodat voor iedere deelverzameling X van L een least upper bound (l.u.b.) $\sqcup X$ bestaat zodat*

$$\forall z \in L, \forall x \in X : \sqcup X \sqsubseteq z \iff x \sqsubseteq z.$$

$\sqcup X$ is hierdoor uniek gespecificeerd.

De extreme gevallen van de least upper bound zijn $\sqcup \phi$ en $\sqcup L$. Deze noteert men $\bot$ en $\top$ en spreekt men uit als "bottom" en "top". Bovendien geldt

$$\forall x \in L : \bot \sqsubseteq x \sqsubseteq \top$$

De machtsverzameling van A is dus de volledige tralie $(\mathcal{P}(A), \subseteq, \cup)$, met $\bot = \sqcup\emptyset = \cup\emptyset = \emptyset$, $\top = \sqcup(A) = \cup\mathcal{P}(A) = A$. Dus de lege verzameling is $\bot$, en de verzameling A is $\top$.

Partiële functies op volledige tralies vormen eveneens een volledige tralie. Stel dat de onderliggende tralie $(B, \preceq, \uplus)$ is, dan is $[A \to B]$ de verzameling van alle partiële functies van A naar B. Als $f(\sigma)$ niet gedefiniëerd is voor $\sigma \in A$, dan schrijven we $f(\sigma) = \bot$. De volgende partiële ordening $\sqsubseteq$ en l.u.b. $\sqcup$ maken dan van $([A \to B], \sqsubseteq, \sqcup)$ een volledige tralie:

Zij $f, g \in [A \to B]$, dan is $\sqsubseteq$, met

$$f \sqsubseteq g \iff \forall \sigma \in A : f(\sigma) \preceq g(\sigma),$$

Voor $F \subseteq [A \to B]$ is $\sqcup F$ gedefiniëerd als:

$$\sqcup F : A \to B$$
$$\sqcup F(\sigma) = \uplus\{f(\sigma) | f \in F\}$$

De semantische functies in Hoofdstukken 4 en 5 zijn dergelijke partiële functies. Het $\bot$ element voor deze tralie is de functie die alle σ afbeeldt op het $\bot$ element van B. Het $\top$ element is de functie die alle σ afbeeldt op $\top$ van B.

Als een functie de orde van zijn argumenten bewaart, dan zeggen we dat deze functie monotoon is.

Definitie 3 (Monotoniciteit)**.** *Zij $(A, \leqslant)$ en $(B, \preceq)$ partieel geordende sets. Een functie $f : A \to B$ is monotoon als en slechts als*

$$\forall x, y \in A : x \leqslant y \Rightarrow f(x) \preceq f(y)$$

We noteren het kleinste vaste punt van een functie f als $lfp(f)$. Het volgende theorema stelt dat het kleinste vaste punt altijd bestaat als $f : D \to D$ monotoon is en D een volledige tralie:

Theorema 1 (Knaster-Tarski)**.** *Zij $f : D \to D$ een monotone functie op volledige tralie D, dan bestaat het kleinste vaste punt (eng. least fixed point) $lfp(f) \in D$ van f.*

In zijn meest algemene vorm is dit bekend als *het Theorema van Knaster-Tarski* Voor meer informatie en het bewijs (van deze beperkte vorm), zie [19].

Constructie van het kleinste vaste punt

Een simpel idee om het kleinste vaste punt van de functie f te berekenen is door herhaaldelijk f toe te passen op een bepaalde startwaarde. Als het resultaat niet meer verandert, dan is het kleinste vaste punt gevonden.

Als het domein van f een volledige tralie is, dan ligt het $\bot$ element voor de hand als startwaarde.

De notie van een Toenemende Kleene Ketting maakt dit formeler:

Definitie 4 (Toenemende Kleene Ketting)**.** *Zij $(L, \sqsubseteq, \sqcup)$ een volledige tralie, en zij $f : L \to L$ een functie, zodat $\forall x \in L : x \sqsubseteq f(x)$, dan is de* Toenemende Kleene Ketting *(eng. Ascending Kleene Chain) van f:*

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots \sqsubseteq f^n(\bot) \sqsubseteq \cdots$$

waarbij $f^0(x) = x$, $f^{n+1}(x) = f(f^n(x))$, $\forall n \in \mathbb{N}$.

Noteer $\mathcal{KC}(f) = \{f^n(\bot) \mid n \in \mathbb{N}\}$. Merk op dat $\mathcal{KC}(f) \subseteq L$, dus $\sqcup \mathcal{KC}(f)$ is uniek gedefiniëerd.

Definitie 5 (Continuïteit)**.** *Zij* $(L, \sqsubseteq, \sqcup)$ *een volledige tralie. Een functie* $f : L \to L$ *is continu als*

$$\forall s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq \cdots \in L : f(\bigsqcup_{i=0}^{\infty} s_i) = \bigsqcup_{i=0}^{\infty} f(s_i)$$

Met deze notatie wordt het volgende bedoeld:

$$\lim_{n \to +\infty} f(\bigsqcup_{i=0}^{n} s_i) = \lim_{n \to +\infty} \bigsqcup_{i=0}^{n} f(s_i)$$

Als een functie f continu is, dan is het in de limiet dus eender of we eerst de l.u.b. berekenen en dan f toepassen, of eerst f toepassen en dan de l.u.b. berekenen. Hieruit volgt dat

$$\begin{aligned}
f(\sqcup \mathcal{KC}(f)) &= \sqcup f(\mathcal{KC}(f)) && [\text{ continuïteit } f \text{ }] \\
&= \sqcup\{f^{n+1}(\bot) \mid n \in \mathbb{N}\} && [\text{ } f \text{ toepassen }] \\
&= \bot \sqcup \{f^{n+1}(\bot) \mid n \in \mathbb{N}\} && [\text{ } \bot \sqcup a = a, \forall a \in L \text{ }] \\
&= \sqcup(\{\bot\} \cup \{f^{n+1}(\bot) \mid n \in \mathbb{N}\}) \\
&= \sqcup\{f^n(\bot) \mid n \in \mathbb{N}\} && [\text{ } f^0(\bot) = \bot \text{ }] \\
&= \sqcup \mathcal{KC}(f)
\end{aligned}$$

Dus $\sqcup \mathcal{KC}(f)$ is een vast punt.

Theorema 2 (Kleene's vaste-puntstheorema)**.** *Als* $f : L \to L$ *monotoon en continu is, dan geldt:*

$$\sqcup \mathcal{KC}(f) = \sqcup\{f^n(\bot) \mid n \in \mathbb{N}\} = \mathit{lfp}(f).$$

Bewijs. Uit de bovenstaande redenering weten we dat $\sqcup \mathcal{KC}(f)$ een vast punt is. Dan moet nog aangetoond worden dat dit ook het *kleinste* vaste punt is.

We tonen met inductie dat voor alle vaste punten $z \in L$ van f geldt: $\forall n \in \mathbb{N} : f^n(\bot) \sqsubseteq z$.

basis $n = 0$

$$f^n(\bot) = f^0(\bot) = \bot \sqsubseteq z$$

inductie Veronderstel dat $f^n(\bot) \sqsubseteq z$, dan geldt:

$$\begin{aligned}
f^n(\bot) \sqsubseteq z &\iff f(f^n(\bot)) \sqsubseteq f(z) && [\text{ } f \text{ is monotoon. }] \\
&\iff f^{n+1}(\bot) \sqsubseteq z && [\text{ } z \text{ is een vast punt, dus } f(z) = z \text{ }]
\end{aligned}$$

Tenslotte volgt dan uit de definitie van de l.u.b. :

$$\forall n \in \mathbb{N} : f^n(\bot) \sqsubseteq z \iff \forall x \in \mathcal{KC}(f) : x \sqsubseteq z \iff \sqcup \mathcal{KC}(f) \sqsubseteq z$$

□

Hoofdstuk 3

Syntax voor Tabulatie in Haskell

De “Effect Handlers” aanpak splitst het modelleren van tabulatie op in twee delen: syntax en effect handlers. De syntax beschrijft de acties waaruit een getabled programma bestaat. De effect handlers interpreteren deze acties en verlenen zo semantiek aan het programma. Het is van groot belang dat de syntax voldoende algemeen of abstract is, zodat een grote variëteit aan handlers mogelijk is.

Programeertalen hebben een abstracte een een concrete syntax. Een programma wordt geschreven in concrete syntax en compilers interpreteren abstracte syntax. De omzetting van concrete naar abstracte syntax gebeurt gewoonlijk in de parser-stap van de compiler. Zowel de concrete als de abstracte syntax wordt in Haskell opgesteld. Hiermee wordt de expliciete implementatie van een parser vermeden. De concrete syntax is de interface die de programmeur gebruikt om de abstracte syntax te construeren. Dit geheel noemt men dan een ingebedde domeinspecifieke taal (Embedded Domain Specific Language, EDSL). Dit in tegenstelling tot een DSL die in een apart broncodebestand moet worden geschreven.

De syntax kan het volgende Prolog programma uitdrukken in Haskell:

```
:- table p/2.

p(1,2).
p(Y,X) :- p(X,Y).
p(X,Y) :- fail.

?- p(A,B).
```

Hierin kunnen we twee verschillende doelen onderscheiden: enerzijds is er het programma met tabulatie, niet-determinisme en faling, anderzijds is er de oproep van `p(A,B)`.

3.1 Concrete syntax: niet-determinisme en recursie

Zoals eerder reeds is aangehaald zijn er twee belangrijke concepten die ondersteund moeten worden: *niet-determinisme* en *recursie*.

Onder niet-determinisme begrijpen we een berekening die nul, één of meerdere resultaten kan hebben. We zeggen dat een berekening mislukt is als ze geen resultaten heeft. Om meer resultaten te hebben volstaat het om één resultaat terug te kunnen geven en een niet-deterministische keuze te kunnen maken tussen twee berekeningen (die op hun beurt weer meerdere resultaten kunnen hebben). Deze drie fundamentele operaties zitten vervat in de volgende typeclass:

$$\begin{aligned}&\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{Nondet}\ m \textbf{ where}\\ &\quad \mathit{fail} :: m\ a\\ &\quad (\vee) :: m\ a \to m\ a \to m\ a\end{aligned}$$

Met *fail* bekomen we een mislukte berekening, de operator $\vee$ maakt een niet-deterministische keuze en met *return* uit de `Monad`-typeclass (waarvan `Nondet` een subclass is) kunnen we één deterministisch resultaat teruggeven.

Omdat een instance van `Nondet` ook een instance van `Monad` is kunnen we gemakkelijk het volgende programma schrijven:

$$\begin{aligned}&p_{nd} :: \mathit{Nondet}\ m \Rightarrow m\ (\mathit{Int}, \mathit{Int})\\ &p_{nd} = \mathit{return}\ (1,2) \vee \textbf{do}\ (x,y) \leftarrow p_{nd}\\ &\qquad\qquad\qquad\qquad\quad \mathit{return}\ (y,x)\end{aligned}$$

Dit programma kiest om ofwel deterministisch $(1,2)$ terug te geven, ofwel om de posities in een recursief verkregen paar (x,y) te verwisselen.

Dit programma is recursief in p_{nd}. Van buitenaf is de recursie echter ongrijpbaar, de recursie is *gesloten*. Tabulatie moet noodzakelijkerwijs kunnen ingrijpen op de recursieve oproepen. De recursie moet dus op een *open* manier gespecificeerd worden. Het volgende type maakt deze intentie duidelijk:

$$\textbf{type } \mathit{Open}\ s = s \to s$$

Namelijk, een open recursieve uitdrukking neemt de expressie die het recursief zal oproepen als een argument. Om dit te verduidelijken herschrijven we p_{nd} als p_{open}:

$$\begin{aligned}&p_{open} :: \mathit{Nondet}\ m \Rightarrow \mathit{Open}\ (m\ (\mathit{Int}, \mathit{Int}))\\ &p_{open}\ r = \mathit{return}\ (1,2) \vee \textbf{do}\ (x,y) \leftarrow r\\ &\qquad\qquad\qquad\qquad\quad \mathit{return}\ (y,x)\end{aligned}$$

Merk op dat $p_{nd} = \mathit{fix}\ p_{open}$ waarbij $\mathit{fix}\ f = \textbf{let}\ a = f\ a\ \textbf{in}\ a$ het vaste punt van p_{open} berekent.

3.2 Abstracte syntax

De abstracte syntax is een Haskell Algebraïsch Datatype:

$$\begin{aligned}\textbf{data}\ \mathit{Table}\ p\ b\ c\ a = {} & \mathit{Pure}\ a \\ \mid {} & \mathit{Fail} \\ \mid {} & \mathit{Or}\ (\mathit{Table}\ p\ b\ c\ a)\ (\mathit{Table}\ p\ b\ c\ a) \\ \mid {} & \mathit{Call}\ p\ b\ (c \to \mathit{Table}\ p\ b\ c\ a)\end{aligned}$$

Juister gezegd kan men de bovenstaande specificatie zien als een *context vrije grammatica (Context Free Grammar, CFG)*, en concrete waarden van dit datatype als *abstracte syntaxbomen (Abstract Syntax Tree, AST)*.

Elke constructor stelt een bepaalde operatie voor:

- *Pure a*: een deterministische berekening met resultaat a
- *Fail*: een mislukte berekening
- *Or l r*: een niet-deterministische keuze tussen twee berekeningen l en r
- *Call p b* $(c \to \mathit{Table}\ p\ b\ c\ a)$: Een recursieve oproep naar het predicaat p, met argument b. De resultaten van het predicaat zijn van het type c. De laatste parameter is de continuatie, die beschrijft hoe verder gerekend moet worden met deze resultaten.

Omdat voor iedere operatie van `Nondet` een constructor aanwezig laat *Table* een eenvoudige instance toe:

$$\begin{aligned}&\textbf{instance}\ \mathit{Nondet}\ (\mathit{Table}\ p\ b\ c)\ \textbf{where} \\ &\quad \mathit{fail} = \mathit{Fail} \\ &\quad (\vee) = \mathit{Or}\end{aligned}$$

De `Monad`-constraint wordt voldaan door de volgende instance:

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ (\mathit{Table}\ p\ b\ c)\ \textbf{where} \\ &\quad \mathit{return} = \mathit{Pure} \\ &\quad \mathit{Pure}\ a \qquad\quad \gg\!= f = f\ a \\ &\quad \mathit{Fail} \qquad\quad \gg\!= f = \mathit{Fail} \\ &\quad \mathit{Or}\ l\ r \qquad\quad \gg\!= f = \mathit{Or}\ (l \gg\!= f)\ (r \gg\!= f) \\ &\quad \mathit{Call}\ p\ b\ \mathit{cont} \gg\!= f = \mathit{Call}\ p\ b\ (\lambda c \to \mathit{cont}\ c \gg\!= f)\end{aligned}$$

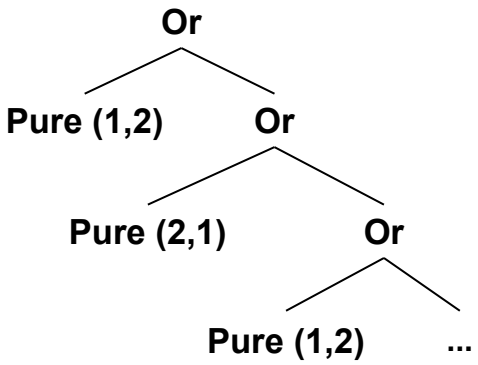
De definitie voor *return* is triviaal. De definitie van $t \gg\!= f$ komt overeen met het substitueren van elk *Pure a*-blad in de boom t met de nieuwe deelbomen gevormd door $f\ a$.

Tenslotte definiëren we nog de volgende hulp functie:

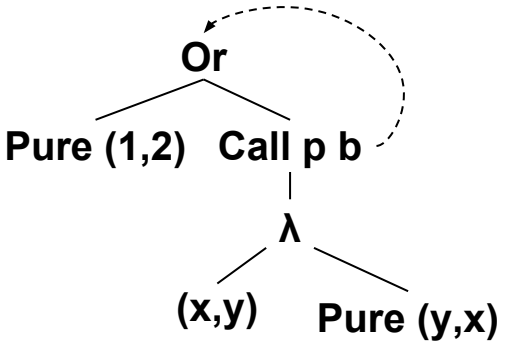
$$\begin{aligned}&\mathit{call} :: p \to b \to \mathit{Table}\ p\ b\ c\ c \\ &\mathit{call}\ p\ b = \mathit{Call}\ p\ b\ \mathit{Pure}\end{aligned}$$

Die na een *Call* gewoon het resultaat ervan teruggeeft.

Een concrete waarde van het type *Table p b c a* is een abstracte syntaxboom, dus kunnen we dit soort waarden grafisch voorstellen. In Figuur 3.1 wordt bij wijze van voorbeeld de AST van de uitdrukking $p_{nd} :: \mathit{Table}\ p\ b\ c\ (\mathit{Int}, \mathit{Int})$ getoond.



Figuur 3.1: De abstracte syntaxboom van p_{nd}.



Figuur 3.2: De abstracte syntaxboom van p_{fin}.

Deze syntaxboom is oneindig! Oneindige syntaxbomen zijn niet geschikt als invoer voor een effect handler. Herinner u dat $p_{nd} = \mathit{fix}\ p_{open}$. De functie *close* is een slimmere manier (dan *fix*) om de open recursie af te sluiten en produceert wel een eindige syntaxboom:

$$\textbf{newtype}\ P\ b\ c = P\ (b \to \mathit{Table}\ (P\ b\ c)\ b\ c\ c)$$
$$\mathit{close} :: \mathit{Open}\ (b \to \mathit{Table}\ (P\ b\ c)\ b\ c\ c) \to b \to \mathit{Table}\ (P\ b\ c)\ b\ c\ c$$
$$\mathit{close}\ o = \textbf{let}\ c\ b = \mathit{Call}\ (P\ (o\ c))\ b\ \mathit{Pure}\ \textbf{in}\ o\ c$$
$$p_{fin} :: \mathit{Table}\ (P\ ()\ (\mathit{Int}, \mathit{Int}))\ ()\ (\mathit{Int}, \mathit{Int})\ (\mathit{Int}, \mathit{Int})$$
$$p_{fin} = \mathit{close}\ (\lambda r\ () \to p_{open}\ (r\ ()))\ ()$$

De constructie $\lambda r\ () \to p_{open}\ (r\ ())$ verenigt p_{open} (dat geen argument verwacht) en *close* (dat een open functie met één argument verwacht). Hierbij maken we gebruik van het isomorfisme tussen de types a en $() \to a$. Figuur 3.2 stelt de eindige AST voor die op deze manier bekomen wordt. De gestreepte pijl verwijst naar het predicaat p dat een pointer bevat naar de root van de syntaxboom.

Verschillende waarden van het type $P\ b\ c$ zijn niet eenvoudig van elkaar te onderscheiden. Dit is problematisch als er meerdere predicaten in het programma voorkomen. Sectie 3.4.1 lost dit op door $P\ b\ c$ uit te breiden met een rangnummer.

3.3 Een getabled programma

In Prolog bestaat een getabled programma uit een reeks clauses die een predicaat definiëren, eventueel aangeduid met een `table`-directieve, gevolgd door een goal (`?- p(A,B)`) die van deze predicaten gebruik maakt:

```
:- table p/2.

p(1,2).
p(Y,X) :- p(X,Y).
p(X,Y) :- fail.

?- p(A,B).
```

De volgende syntax maakt dezelfde structuur mogelijk in Haskell:

$$\begin{aligned}&\textbf{data}\ \mathit{Program}\ b\ c\ m\ a\\&\quad = \mathit{Decl}\ (\mathit{Open}\ [b \to m\ c])\ ([b \to m\ c] \to \mathit{Program}\ b\ c\ m\ a)\\&\quad \mid \mathit{Expr}\ (m\ a)\end{aligned}$$

Deze syntax heeft twee constructors:

- *Decl ps cont* declareert een lijst *ps* van *Open* predicaten die b als argument nemen en $m\ c$ als resultaat hebben. De continuatie *cont* verwacht dat deze predicaten in gesloten vorm.
- *Expr m* omvat een expressie van het type $m\ a$. Een expressie heeft geen continuatie. Na een expressie kan dus geen declaratie of andere expressie meer volgen. De expressie bepaalt dus het resultaat van het hele programma.

Declaraties komen overeen met `table`-directieven en clauses. De expressie komt overeen met de goal.

Voor *Program* bestaat een *Applicative* instance:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ m \Rightarrow \mathit{Functor}\ (\mathit{Program}\ b\ c\ m)\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathit{Decl}\ p\ \mathit{cont}) = \mathit{Decl}\ p\ (\mathit{fmap}\ f \circ \mathit{cont})\\&\quad \mathit{fmap}\ f\ (\mathit{Expr}\ m) \quad\ = \mathit{Expr}\ (\mathit{fmap}\ f\ m)\\&\textbf{instance}\ \mathit{Applicative}\ m \Rightarrow \mathit{Applicative}\ (\mathit{Program}\ b\ c\ m)\ \textbf{where}\\&\quad \mathit{pure} = \mathit{Expr} \circ \mathit{pure}\\&\quad \mathit{Decl}\ f\ \mathit{contf} \circledast m \qquad\qquad = \mathit{Decl}\ f\ (\lambda cf \to \mathit{contf}\ cf \circledast m)\\&\quad \mathit{Expr}\ m \quad \circledast \mathit{Decl}\ f\ \mathit{conta} = \mathit{Decl}\ f\ (\lambda ca \to \mathit{Expr}\ m \circledast \mathit{conta}\ ca)\\&\quad \mathit{Expr}\ \mathit{ff} \quad \circledast \mathit{Expr}\ \mathit{fa} \quad\ = \mathit{Expr}\ (\mathit{ff} \circledast \mathit{fa})\end{aligned}$$

De volgende slimme constructoren *declare*, *declareRec* en *expression* zijn vereenvoudigen het schrijven van *Programs*.

$$\begin{aligned}&\mathit{declare} :: \mathit{Open}\ (b \to m\ c)\\&\qquad \to ((b \to m\ c) \to \mathit{Program}\ b\ c\ m\ a)\end{aligned}$$

$$\rightarrow \mathit{Program}\ b\ c\ m\ a$$
$$\mathit{declare}\ p\ c = \mathit{Decl}\ (\mathit{singleton} \circ p \circ \mathit{head})\ (c \circ \mathit{head})\ \textbf{where}\ \mathit{singleton}\ x = [x]$$

De constructor *declare* declareert een enkel predicaat door de body te verpakken in een lijst met één element. De andere twee constructors zijn slechts synoniemen voor de echte constructors:

$$\mathit{declareRec} = \mathit{Decl}$$
$$\mathit{expression} = \mathit{Expr}$$

Nu kan het volledige Tabled Prolog voorbeeld in Haskell worden uitgedrukt:

$$\mathit{program} :: \mathit{Nondet}\ m \Rightarrow \mathit{Program}\ ()\ (\mathit{Int}, \mathit{Int})\ m\ (\mathit{Int}, \mathit{Int})$$
$$\mathit{program} = \mathit{declare}\ (\lambda p\ () \rightarrow \mathit{return}\ (1,2) \vee \mathit{fail} \vee \textbf{do}\ (x, y) \leftarrow p\ ()\ \mathit{return}\ (y, x))\ (\lambda p \rightarrow \mathit{expression}\ (p\ ()))$$

De functie *declareRec* accepteert een lijst van Open predicaten. Dit maakt het mogelijk om wederzijds recursieve predicaten te definiëren. Het volgende voorbeeld demonstreert de definitie van twee wederzijds recursieve predicaten p en q.

$$\mathit{programRec} :: \mathit{Program}\ \mathit{Int}\ \mathit{Int}\ (\mathit{Table}\ p\ \mathit{Int}\ \mathit{Int})\ \mathit{Int}$$
$$\mathit{programRec} =$$
$$\quad \mathit{declareRec}\ (\lambda{\sim}[p, q] \rightarrow [\lambda x \rightarrow q\ (x-1),\ \lambda x \rightarrow \textbf{if}\ x \leqslant 0\ \textbf{then}\ \mathit{return}\ x\ \textbf{else}\ p\ x])\ (\lambda[p, q] \rightarrow \mathit{expression}\ \$\ p\ 42)$$

De lazyness annotatie($\sim$) in $\lambda{\sim}[p, q]$ is noodzakelijk, omdat de bodies (van de predicaten p en q) anders onnodig strikt zouden zijn, wat tot problemen kan leiden wanneer de syntax uitgevoerd moet worden. Concreet wil dit zeggen dat het argument $[p, q]$ pas geëvalueerd wordt op het moment dat p of q echt nodig zijn. Dit wil zeggen dat in het basisgeval (i.e. de `then`-branch van q) van de recursie het argument niet meer geëvalueerd moeten worden, waardoor de recursie ook effectief stopt.

De *Program*-syntax zoals deze hier is voorgesteld heeft twee belangrijke beperkingen. Ten eerste, alle predicaten moeten hetzelfde argumentstype b en resultaatstype c hebben. Ten tweede moeten alle predicaten gedeclareerd zijn voor de uiteindelijke *expression*. Het is dus niet mogelijk om predicaten te definiëren "on-the-fly".

3.4 Polymorfisme afdwingen

Het eerdere voorbeeld had *program* het type

$$\mathit{program} :: \mathit{Nondet}\ m \Rightarrow \mathit{Program}\ ()\ (\mathit{Int}, \mathit{Int})\ m\ (\mathit{Int}, \mathit{Int})$$

Deze code is dus polymorf in de parameter m (het type van de monad waaruit expressies bestaan). Dit is mogelijk omdat *Nondet* alle nodige operaties (succes, faling en keuze) kan uitdrukken.

Met behulp van een `newtype`-constructie kan het typesysteem deze beperking afdwingen:[1]

[1] Met de GHC extensie `RankNTypes`.

```
newtype TabledProgram b c a = TP {
  unTP :: ∀ m . Nondet m ⇒ Program b c m a
}
```

Wanneer de programmeur moet werken binnen *TabledProgram*, dan beschikt deze alleen over de constructies van *Program* (*declare*, *declareRec*, *expression*), *Nondet* (*fail* en ∨) en *Monad* (*return* en ≫=).

```
programTP :: TabledProgram () (Int, Int) (Int, Int)
programTP = TP $
  declare (λp () → return (1, 2) ∨ fail ∨ do (x, y) ← p ()
                                             return (y, x))
          (λp → expression (p ()))
```

Dit verhindert de programmeur natuurlijk niet om buiten *TabledProgram* te werken, bijvoorbeeld wanneer er naast de operaties van *Nondet* nog andere operaties gewenst zijn.

3.4.1 Vertaling van *TabledProgram* naar *Table*

Deze sectie bespreekt hoe een *TabledProgram b c a* naar een *Table p b c a* wordt omgezet. Eerst moet een concreet type voor de parameter *p* gekozen worden. Dit type moet toe laten om enerzijds het predicaat uniek te identificeren en anderzijds de body van het predicaat uit te voeren. Deze functies worden vervuld door respectievelijk *pId* en *pBody* in het datatype *Predicate*.

```
data Predicate b c = Predicate {
  pId    :: Int,
  pBody :: b → Table (Predicate b c) b c c
}
instance Eq (Predicate b c) where
  Predicate i1 b1 ≡ Predicate i2 b2 = i1 ≡ i2
```

Twee verschillende *Predicate*s kunnen nu worden onderscheiden aan de hand van hun identifiers.

Het vertalen van `TabledProgram b c a` naar `Table (Predicate b c) b c a` verloopt redelijk eenvoudig. Na het uitpakken van een *TabledProgram* blijft een *Program* over. Dit *Program* valt in één van de volgende twee gevallen:

- De vertaling van een *Expr e* met type ∀ *m* . *Nondet m* ⇒ *Program b c m a* is *e* zelf, aangezien `Table` een instance is van *Nondet*.
- De vertaling van een *Decl ps cont* is de vertaling van de continuatie *cont* toegepast op *predicates*, waar *predicates* de lijst van *Predicate*s is die verkregen wordt door in ieder predicaat van *ps* de recursieve calls te vervangen door een *call*[2] naar het *Predicate* dat met dit predicaat overeenkomt.

[2] De hulpfunctie *call*, niet de constructor *Call*.

```
tpToTable :: TabledProgram b c a → Table (Predicate b c) b c a
tpToTable p = tpToTable' 0 (unTP p) where
  tpToTable' i (Expr e)        = e
  tpToTable' i (Decl ps cont) = tpToTable' (i + n) (cont predicates) where
    predicates = map call (zipWith Predicate [i..] (ps predicates))
    n          = length predicates
```

Eens een waarde van het type *Table* bekomen is, kunnen hierop effect handlers losgelaten worden.

Typisch is dus de manier om een programma uit te voeren:

```
⋙ effectHandler $ tpToTable $ program
< Het gewenste effect . >
```

Deze vertaling brengt een ander probleem aan het licht. Stel dat er twee verschillende *TabledPrograms* *p1* en *p2* zijn, die elk minstens één predicaat declareren. Dan kunnen we op beide *tpToTable* toepassen:

```
t1 :: Table (Predicate b c) b c a
t1 = tpToTable p1
t2 :: Table (Predicate b c) b c a
t2 = tpToTable p2
```

De sequentie $t1 \gg t2$ is dan eveneens een *Table* (*Predicate b c*) *b c a*. Het eerste predicaat van *t1* en het eerste predicaat van *t2* hebben dezelfde identifier. Een effect handler kan ze dus onmogelijk van elkaar onderscheiden. Wanneer $t1 \gg t2$ door de effect handler wordt uitgevoerd kan er ongewenste interferentie ontstaan tussen de twee predicaten. Dit is een goede reden om steeds binnen *TabledProgram* te werken, aangezien het resultaat van *tpToTable* niet opnieuw in een *TabledProgram* gebruikt kan worden. *TabledProgram* ($t1 \gg t2$) geeft namelijk een type-fout.

3.5 Samenvatting

De programmeur stelt steeds zijn programma op in het type *TabledProgram b c a*. Hierbij wordt enkel gebruik gemaakt van de *Nondet* interface en *Open* recursie en wat Haskell verder aan taalconstructies te bieden heeft.

De syntax is eenvoudig, wat de programmeur en de effect handlers ten goede komt. Voor de programmeur is het aantal nieuwe concepten erg beperkt, zodat hij of zij snel zijn of haar weg vindt binnen het systeem.

De complexiteit van een effect handler staat in rechtstreekse verhouding tot de complexiteit van de syntax waarop ze betrekking hebben. Een eenvoudige syntax maakt dus eenvoudigere handlers mogelijk. Desondanks zijn nog steeds sterk verschillende (interessante) interpretaties van de syntax mogelijk. Deze interpretaties zullen in het vervolg van deze tekst aan bod komen.

Hoofdstuk 4

Denotationele Semantiek

Een compiler zet een abstracte syntaxboom om een machinecode. De compiler geeft dus een betekenis, een semantiek, aan de syntaxboom. Analoog geeft een effect handler een semantiek aan de abstracte syntax. In tegenstelling tot machinecode is het in eerste instantie de bedoeling dat deze semantiek voor mensen begrijpbaar is. Om dit te bewerkstelligen moet de semantiek dus abstractie maken van overbodige details.

De abstracte taal bij uitstek is de wiskunde. Het doel van een wiskundige semantiek is om een correcte en waardevolle correspondentie te geven tussen getablede programma's (voorgesteld door de syntax) en objecten uit een wiskundig domein. De semantiek is geïnspireerd de aanpak van Scott en Strachey [19]. In dit kader proberen we de operationele aspecten te negeren en een vooral een denotationele semantiek te geven. Concreet heeft de semantiek daarom de eenvoudige vorm van een functie $\mathcal{T}[\![\,.\,]\!]$ die een abstracte syntaxboom afbeeldt op de verzameling van zijn oplossingen.

Eens een semantiek is opgesteld kunnen we deze vertalen in Haskell. De semantiek is dus uitvoerbaar. Door het functionele karakter van de semantiek is deze vertaling redelijk eenvoudig.

In de tweede plaats kunnen we controleren in hoeverre de semantiek operationeel overeenkomt met Tabled Prolog. Namelijk, beschouw opnieuw het volgende Prolog voorbeeld:

```
:- table p/2.
p(1,2).
p(Y,X) :- p(X,Y).
?- p(A,B).
```

Dit geeft `A=1, B=2` en `A=2, B=1` als oplossingen. In Hoofdstuk 3 wordt een Haskell-tegenhanger besproken:

$$
\begin{aligned}
&\mathit{program} :: \mathit{TabledProgram}\ ()\ (\mathit{Int}, \mathit{Int})\ (\mathit{Int}, \mathit{Int})\\
&\mathit{program} = \mathit{TP}\ \$\ \mathit{declare}\ (\lambda p\ () \rightarrow \mathit{return}\ (1,2) \vee \mathbf{do}\ (x,y) \leftarrow p\ ()\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{return}\ (y,x))\\
&\qquad\qquad\qquad\qquad (\lambda p \rightarrow \mathit{expression}\ (p\ ()))
\end{aligned}
$$

Stel dat de functie $\mathcal{T}[\![\,.\,]\!]$ de semantiek is die een getabled programma afbeeldt op de verzameling van al zijn oplossingen.

In dit concrete geval willen we dus dat $\mathcal{T}[\![\, tpToTable\ program \,]\!]$ de verzameling $\{(1,2),(2,1)\}$ geeft, symbolisch:

$$\begin{aligned}
&\mathcal{T}[\![\, tpToTable\ program \,]\!] = \{(1,2),(2,1)\} \\
&\iff \quad [\text{ vertaling door } \texttt{tpToTable}.] \\
&\mathcal{T}[\![\, Call\ p\ ()\ (\lambda a \to Pure\ a) \,]\!] = \{(1,2),(2,1)\}
\end{aligned}$$

waarbij $p = Predicate\ 0\ (\lambda() \to Pure\ (1,2)\ `Or`\ Call\ p\ ()\ (\lambda(x,y) \to Pure\ (y,x))$

Tenslotte geven we nog een optimalisatie voor de semantiek in Haskell: afhankelijheidsanalyse. Afhankelijkheidsanalyse is een meer fundamentele verandering aan het algoritme dat een hoop overbodig werk kan vermijden.

4.1 Opbouw van de Denotationele Semantiek

Herriner u de `Table` en `Predicate` uit Hoofdstuk 3.

$$\begin{aligned}
&\textbf{data}\ Table\ p\ b\ c\ a = Pure\ a \\
&\qquad\qquad\qquad\qquad\ \ |\ \ Fail \\
&\qquad\qquad\qquad\qquad\ \ |\ \ Or\ (Table\ p\ b\ c\ a)\ (Table\ p\ b\ c\ a) \\
&\qquad\qquad\qquad\qquad\ \ |\ \ Call\ p\ b\ (c \to Table\ p\ b\ c\ a)
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ Predicate\ b\ c = Predicate\ \{ \\
&\quad pId\ \ \ \ :: Int, \\
&\quad pBody :: b \to Table\ (Predicate\ b\ c)\ b\ c\ c \\
&\}
\end{aligned}$$

Voor de gemakkelijkheid noteren we $Table\ p\ b\ c\ a$ als $Table\ A$ en $predicateBody\ p\ b$ als $p(b)$. De effect handler of semantiek is een functie $\mathcal{T}[\![\,.\,]\!] : Table\ A \to \mathcal{P}(A)$. Deze functie beeldt een programma af op de verzameling van al zijn oplossingen. Omdat de de `Table` datastructuur recursief is, is het mogelijk om $\mathcal{T}[\![\,.\,]\!]$ op een inductieve manier te definiëren.

basisgevallen We onderscheiden twee basisgevallen: $Pure\ a$ en $Fail$. Hun oplossingsverzamelingen zijn respectievelijk het singleton $\{a\}$ en de lege verzameling $\emptyset$.

$$\mathcal{T}[\![\, Pure\ a \,]\!] = \{a\} \tag{4.1}$$
$$\mathcal{T}[\![\, Fail \quad \,]\!] = \emptyset \tag{4.2}$$

inductieve gevallen De oplossingsverzameling van *Or a b* is de unie van de oplossingsverzamelingen van beide keuzes.

$$\mathcal{T}[\![\ Or\ l\ r\ ]\!] = \mathcal{T}[\![\ l\ ]\!] \cup \mathcal{T}[\![\ r\ ]\!]$$

Wat is nu het resultaat van een *Call p b cont*? Veronderstel dat we beschikken over een functie s die de oplossingen weet voor alle combinaties van predicaten en agrumenten die voorkomen in $p(b)$. Dan kunnen we de functie $\mathcal{C}[\![\ .\ ]\!]$ bedenken, zodat $\mathcal{C}[\![\ .\ ]\!]$ het resultaat van $p(b)$ berekent gegeven s. Het gedrag van $\mathcal{C}[\![\ .\ ]\!]$ is identiek aan $\mathcal{T}[\![\ .\ ]\!]$, behalve voor *Call q k cont*. Een dergelijke call wordt berekent door *cont* toe te passen op elke oplossing die in $s(q,k)$ voorkomt, en dan de unie te nemen van de oplossingsverzamelingen hiervan.

$$\begin{aligned}
&\mathcal{C}[\![\ .\ ]\!] : \mathit{Table}\ C \to (\mathit{Predicate}\ B\ C \times B \to \mathcal{P}(C)) \to \mathcal{P}(C)\\
&\mathcal{C}[\![\ \mathit{Pure}\ c\ ]\!](s) = \{c\}\\
&\mathcal{C}[\![\ \mathit{Fail}\quad\ ]\!](s) = \emptyset\\
&\mathcal{C}[\![\ \mathit{Or}\ l\ r\ ]\!](s) = \mathcal{C}[\![\ l\ ]\!](s) \cup \mathcal{C}[\![\ r\ ]\!](s)\\
&\mathcal{C}[\![\ \mathit{Call}\ q\ k\ \mathit{cont}\ ]\!](s) = \bigcup_{c \in s(q,k)} \mathcal{C}[\![\ \mathit{cont}(c)\ ]\!](s)
\end{aligned} \tag{4.3}$$

Stel dat s niet enkel de resultaten bevat voor de predicaten die in $p(b)$ voorkomen, maar *alle* predicaten die in het programma voorkomen, dan moet het resultaat van $\mathcal{C}[\![\ .\ ]\!]$ gekend zijn door s, i.e. $\mathcal{C}[\![\ p(b)\ ]\!](s) = s(p,b)$ voor alle predicaten p en $b \in B$. Dan moet s een vast punt zijn van de functie $\lambda s.\lambda(q,k).\mathcal{C}[\![\ q(k)\ ]\!](s)$. In Appendix A.1 wordt aangetoond dat deze functie inderdaad monotoon en continu is, bijgevolg bestaat het kleinste vaste punt van deze functie.

Definieer nu de functie *table* die dit kleinste vaste punt gebruikt:

$$\begin{aligned}
&\mathit{table} : \mathit{Predicate}\ B\ C \times B \to \mathcal{P}(C)\\
&\mathit{table}(p,b) = \mathit{lfp}(\lambda s.\lambda(q,k).\mathcal{C}[\![\ q(k)\ ]\!](s))(p,b)
\end{aligned} \tag{4.4}$$

Uiteindelijk wordt de semantiek van *Call p b cont* dan:

$$\mathcal{T}[\![\ \mathit{Call}\ p\ b\ \mathit{cont}\ ]\!] = \bigcup_{c \in \mathit{table}(p,b)} \mathcal{T}[\![\ \mathit{cont}(c)\ ]\!] \tag{4.5}$$

Net zoals in vergelijking (4.3) berekenen we hier de oplossingsverzameling door eerst *cont* toe te passen op elke oplossing in *table* (p, b) en namen vervolgens de unie van de oplossingsverzamelingen hiervan.

4.2 Vergelijking met Tabled Prolog

In deze sectie komen we terug op de vraag uit de inleiding: is het resultaat dat de semantiek bekomt equivalent aan het antwoord dat Tabled Prolog berekent?

Bewijs. Eerst vertalen we *program* naar een *Table*.

$$\begin{aligned}
&\mathcal{T}[\![\ tpToTable\ program\ ]\!] \equiv \{(1,2),(2,1)\} \\
&\iff \quad [\ \text{Vertaling door } \texttt{tpToTable}.\ ] \\
&\mathcal{T}[\![\ Call\ p\ ()\ (\lambda a \to Pure\ a)\ ]\!] \equiv \{(1,2),(2,1)\}
\end{aligned}$$

waarbij

$$\begin{aligned}
&p :: Predicate\ ()\ (Int, Int) \\
&p = Predicate\ 0\ (\lambda() \to Pure\ (1,2)\ `Or`\ Call\ p\ ()\ (\lambda(x,y) \to Pure\ (y,x)))
\end{aligned}$$

Toepassing van $\mathcal{T}[\![\,.\,]\!]$, $\mathcal{C}[\![\,.\,]\!]$ en *table* geeft het antwoord:

$$\begin{aligned}
&\mathcal{T}[\![\ Call\ p\ ()\ (\lambda a \to Pure\ a)\ ]\!] \\
&\equiv\ [\ (4.5)\ \text{toepassen.}\ ] \\
&\bigcup_{c\,\in\, table((p,()))} \mathcal{T}[\![\ Pure\ c\ ]\!] \\
&\equiv\ [\ (4.1)\ \text{toepassen.}\ ] \\
&\bigcup_{c\,\in\, table((p,()))} \{c\} \\
&\equiv\ [\ \cup_{a\in A}\{a\}\ \text{is identiteit.}\ ] \\
&table((p,())) \\
&\equiv\ [\ table\ (4.4)\ \text{toepassen.}\ ] \\
&lfp\Big(\lambda s.\lambda(q,k).\mathcal{C}[\![\ q(k)\ ]\!](s)\Big)(p,())
\end{aligned}$$

Het kleinste vaste punt van $\lambda s.\lambda(q,k).\mathcal{C}[\![\ q(k)\ ]\!](s)$ is de functie f:

$$\begin{aligned}
&f : (Predicate\ ()\ (Int,Int),()) \to \mathcal{P}((Int,Int)) \\
&f(q,()) = \begin{cases} \{(1,2),(2,1)\} & \text{als q = p} \\ \bot & \text{anders} \end{cases}
\end{aligned}$$

want

$$
\begin{aligned}
&\mathcal{C}[\![\, p(\,()\,) \,]\!](f)\\
&\equiv\\
&\mathcal{C}[\![\, \mathit{Pure}\ (1,2)\ `\mathit{Or}`\ \mathit{Call}\ p\ ()\ (\lambda(x,y) \to \mathit{Pure}\ (y,x)) \,]\!](f)\\
&\equiv\\
&\mathcal{C}[\![\, \mathit{Pure}\ (1,2) \,]\!](f) \cup \mathcal{C}[\![\, \mathit{Call}\ p\ ()\ (\lambda(x,y) \to \mathit{Pure}\ (y,x)) \,]\!](f)\\
&\equiv\\
&\{(1,2)\} \cup \Big(\bigcup_{(x,y)\in f(p,())} \mathcal{C}[\![\, \mathit{Pure}\ (y,x) \,]\!](f) \Big)\\
&\equiv\ [\ f(p,()) = \{(1,2),(2,1)\}\ ]\\
&\{(1,2)\} \cup \mathcal{C}[\![\, \mathit{Pure}\ (2,1) \,]\!](f) \cup \mathcal{C}[\![\, \mathit{Pure}\ (1,2) \,]\!](f)\\
&\equiv\\
&\{(1,2)\} \cup \{(2,1)\} \cup \{(1,2)\} \equiv \{(1,2),(2,1)\}
\end{aligned}
$$

Dan volgt:

$$
\mathit{lfp}\Big(\lambda s.\lambda(p,k).\mathcal{C}[\![\, p(k) \,]\!](s)\Big)(p,()) \equiv f(p,()) \equiv \{(1,2),(2,1)\}
$$

□

De semantiek $\mathcal{T}[\![\,.\,]\!]$ bereikt inderdaad het beoogde resultaat.

4.3 Uitvoerbaarheid

De formele semantiek geeft een werkende, zij het gruwelijk inefficiënte, effect handler voor de `Table`-syntax in Haskell. De interpreter begrijpt de volgende tabulatie monad: *Table* (*Predicate b c*) *b c c*, met `Predicate` gedefiniëerd in Sectie 3.4.1.

De functie *ts* is een vrij rechttoe-rechtaan vertaling van $\mathcal{T}[\![\,.\,]\!]$, dankzij het datatype *Set* uit de standaard module `Data.Set` en de functie *foldMap* uit `Data.Foldable`.

```
ts :: (Ord a, Ord b, Ord c) ⇒ Table (Predicate b c) b c a → S.Set a
ts (Pure a)        = S.singleton a
ts Fail            = S.empty
ts (Or t1 t2)      = ts t1 `S.union` ts t2
ts (Call p b cont) = F.foldMap (ts . cont) (table p b)
```

met *table*:

```
table :: (Ord b, Ord c) ⇒ Predicate b c → b → S.Set c
table = lfp (λs q k → cs (pBody q k) s) (λq k → S.empty)
```

De functie *cs* is de Haskell vertaling van $\mathcal{C}[\![\,.\,]\!]$:

```
cs :: (Ord b, Ord c)
   ⇒ Table (Predicate b c) b c c
   → (Predicate b c → b → S.Set c)
   → S.Set c
cs (Pure a)          s = S.singleton a
cs  Fail             s = S.empty
cs (Or t1 t2)        s = cs t1 s `S.union` cs t2 s
cs (Call p b cont) s = F.foldMap (flip cs s . cont) (s p b)
```

De definitie van *lfp* laten we bewust leeg. In de volgende sectie bespreken we een manier om dit te implementeren.[1]

```
lfp :: (a → a) → a → a
```

4.3.1 Afhankelijkheidsanalyse

Een uiterst naïeve berekening an het kleinste vaste punt bestaat erin de semantische functie $\mathcal{C}[\![\,.\,]\!]$ over alle predicaten afwisselend te itereren totdat de gevonden oplossingen niet meer veranderen. Een dergelijke implementatie doet echter veel overbodig werkt. Veel snelheidswinst is te halen uit afhankelijkheidsanalyse van de predicaten onderling.

Een predicaat p is afhankelijk van een predicaat q als ergens in de body van p een call naar q voorkomt. Merk op dat predicaten dus ook van zichzelf afhankelijk kunnen zijn.

De volgende semantiek beschrijft de analyse formeel:

$$\mathcal{D}[\![\,.\,]\!] : \mathit{Table}\ C \to (\mathit{Predicate}\ B\ C \times B \to C) \to \mathcal{P}(\mathit{Predicate}\ B\ C \times C)$$
$$\mathcal{D}[\![\ \mathit{Pure}\ a\ ]\!] = \emptyset$$
$$\mathcal{D}[\![\ \mathit{Fail}\ \ ]\!] = \emptyset$$
$$\mathcal{D}[\![\ \mathit{Or}\ a\ b\ ]\!] = \mathcal{D}[\![\ a\ ]\!](s) \cup \mathcal{D}[\![\ b\ ]\!](s)$$
$$\mathcal{D}[\![\ \mathit{Call}\ p\ b\ \mathit{cont}\ ]\!] = \underbrace{\{(p,b)\}}_{\text{huidige call}} \bigcup \Big(\underbrace{\cup\{\mathcal{D}[\![\ \mathit{cont}(c)\ ]\!](s) \mid c \in{\downarrow}s(p,b)\}}_{\text{continuaties}} \Big)$$

Pure resultaten (*Pure*) of mislukte berekeningen (*Fail*) hebben geen afhankelijkheden. Een keuze (*Or*) is afhankelijk van de afhankelijkheden van zijn opties. Een recursieve oproep (*Call*) is afhankelijk van het predicaat en argument dat het onderwerp vormen van de oproep en van de afhankelijkheden van de continuaties.

[1] Andere implementaties staan in de online repository. https://bitbucket.org/AlexanderV/thesis

Beschouw de opeenvolgende stappen van een naïeve vaste-puntsberekening ($table^0$ is het resultaat na nul iteraties, $table^1$ na één iteratie, enzovoorts):

$$\begin{aligned} table^0(p,b) &= \emptyset \\ &\subseteq \\ table^1(p,b) &= \mathcal{C}[\![\, p(b) \,]\!](table^0) \\ &\subseteq \\ &\dots \\ &\subseteq \\ table^{n+1}(p,b) &= \mathcal{C}[\![\, p(b) \,]\!](table^n) \\ &\subseteq \\ &\dots \end{aligned}$$

Dit kunnen we herschrijven als:

$$\begin{aligned} table^0(p,b) &= \emptyset \\ table^1(p,b) &= \mathcal{C}[\![\, p(b) \,]\!](table^0) \\ table^{n+2}(p,b) &= \mathcal{C}[\![\, p(b) \,]\!](table^{n+1}) \quad (\forall n \in \mathbb{N}) \end{aligned}$$

Het cruciale idee is nu dat als in $table^{n+1}$ niets verandert aan de afhankelijkheden van een predicaat, dan veranderen de oplossingen van het predicaat zelf ook niet veranderen! Meer formeel:

$$\overbrace{table^{n+2}(p,b) = table^{n+1}(p,b)}^{(p,b) \text{ verandert niet.}}$$
$$\Longleftrightarrow$$
$$\underbrace{\forall (q,k) \in \mathcal{D}[\![\, p(b) \,]\!](table^{n+1}) :\ table^{n+1}(q,k) = table^n(q,k)}_{\text{Dependencies van } (p,b) \text{ veranderen niet.}}$$

Dan kan $table^{n+2}$ als volgt herscherven worden:

$$table^{n+2}(p,b) = \begin{cases} table^{n+1}(p,b) & \text{als } \forall d \in \mathcal{D}[\![\, p(b) \,]\!](table^{n+1}) : \\ & \quad table^{n+1}(d) = table^n(d) \\ \mathcal{C}[\![\, p(b) \,]\!](table^{n+1}) & \text{anders} \end{cases}$$

Waardoor een nutteloze evaluatie van $\mathcal{C}[\![\, . \,]\!]$ vermeden wordt. Op een gelijkaardige manier verandert de verzameling van afhankelijkheden niet als de afhankelijkheden zelf niet veranderen. Dit wil zeggen dat afhankelijkheden eveneens gedeeld kunnen worden tussen opeenvolgende stappen.

De implementatie van deze ideëen evalueert in iedere iteratie één predicaat en onthoudt de nieuwe resultaten en afhankelijkheden. Daarna voert het de afhankelijke predicaten uit, die op hun beurt hun afhankelijkheden uitvoeren. Een predicaat wordt dus alleen geëvalueerd als in de voorgaande iteratie een afhankelijkheid veranderde.

In Haskell onthouden twee datastructuren T en D de tussentijdse resultaten en tussentijdse omgekeerde afhankelijkheden. [2]

$$\textbf{data}\ T\ b\ c\ s$$
$$\textbf{data}\ D\ b\ c$$

Deze types ondersteunen een aantal operaties:

- Aanmaak van een lege datastructuur:

$$emptyT :: T\ b\ c\ s$$
$$emptyD :: D\ b\ c$$

- Opzoekwerk (al dan niet in een *State* monad):

$$\begin{aligned}
&findInD \ \ :: Ord\ b \Rightarrow Predicate\ b\ c \to b \to D\ b\ c \to S.Set\ (Predicate\ b\ c, b)\\
&findInT \ \ :: (Ord\ b, Monoid\ s) \Rightarrow Predicate\ b\ c \to b \to T\ b\ c\ s \to s\\
&findInTS :: (Ord\ b, Monoid\ s)\\
&\qquad\qquad \Rightarrow Predicate\ b\ c \to b \to State\ (T\ b\ c\ s, D\ b\ c)\ s\\
&findInTS\ p\ b = gets\ (findInT\ p\ b\,.\,fst)
\end{aligned}$$

- Updates:

$$\begin{aligned}
&updateT :: (Ord\ b, Monoid\ s) \Rightarrow Predicate\ b\ c \to b \to s \to T\ b\ c\ s \to T\ b\ c\ s\\
&updateD :: Ord\ b\\
&\qquad\qquad \Rightarrow Predicate\ b\ c \to b \to [(Predicate\ b\ c, b)] \to D\ b\ c \to D\ b\ c
\end{aligned}$$

- Filteren op nieuwe afhankelijkheden:

$$new :: Ord\ b \Rightarrow (((Predicate\ b\ c, b) \to Bool) \to w) \to T\ b\ c\ s \to w$$

De Haskell functie `deps` berekent afhankelijkheden, zoals $\mathcal{D}[\![\,.\,]\!]$:

$$\begin{aligned}
&deps :: (Ord\ p, Ord\ b) \Rightarrow Table\ p\ b\ c\ c \to (p \to b \to S.Set\ c) \to S.Set\ (p, b)\\
&deps\ (Pure\ c) \qquad\quad s = S.empty\\
&deps\ \ Fail \qquad\quad\ \ s = S.empty\\
&deps\ (Or\ l\ r) \qquad\ s = deps\ l\ s \diamond deps\ r\ s\\
&deps\ (Call\ p\ b\ cont)\ s =\\
&\quad S.insert\ (p, b)\ \$\ F.foldMap\ (flip\ deps\ s\,.\,cont)\ \$\ s\ p\ b
\end{aligned}$$

[2] De implementatie van deze datastructuren is beschikbaar in de online repository https://bitbucket.org/AlexanderV/thesis

De functie *tableA* vervangt *table* en berekent het vaste punt:

```
tableA :: (Eq b, Ord b, Ord c)
       ⇒ Predicate b c → b → State (T b c (S.Set c), D b c) (S.Set c)
tableA p b = do
  (t0, d0) ← get
  let findInT0 q c = findInT q c t0
      v  = cs   (pBody p b) findInT0
      w  = deps (pBody p b) findInT0
      t1 = updateT p b v t0
      d1 = updateD p b (S.toList w) d0
  put (t1, d1)
  if findInT p b t0 ≡ findInT p b t1 then
    tables (new (flip S.filter w) t0)
  else
    tables (new (flip S.filter w) t0 ⋄ findInD p b d1)
  findInTS p b   -- result after recursion
```

Met de oude toestand $(t0, d0)$ worden voor p en b nieuwe resultaten v en afhankelijkheden w berekend. De nieuwe toestand $(t1, d1)$ wordt opgeslagen. Als de toestand onveranderd is, dan worden enkel de nieuwe predicaten uitgevoerd, anders worden alle afhankelijke predicaten (en de nieuwe predicaten) uitgevoerd.

De hulpfunctie *tables* voert *tableA* uit op een verzameling predicaat-argument paren. Het resultaat wordt opgeslagen in een *State*-monad, de functie geeft geen andere nuttige waarde terug.

```
tables :: (Ord b, Ord c)
       ⇒ S.Set (Predicate b c, b) → State (T b c (S.Set c), D b c) ()
tables = void . traverse (uncurry tableA) . S.toList
```

De functie *traverse* komt uit *Data.Traversable*, de functie $void :: Functor\ f \Rightarrow f\ a \rightarrow f\ ()$ uit *Control.Monad*.

Tenslotte is er de aangepaste semantische functie *tsA*. Deze gebruikt nu *tableA* in plaats van *table* en *fixF*. De interne functie *go* loopt ook in de *State*-monad.

```
tsA :: (Ord a, Ord b, Ord c, Show b) ⇒ Table (Predicate b c) b c a → S.Set a
tsA t = evalState (go t) (emptyT, emptyD) where
  go (Pure a) = return (S.singleton a)
  go Fail = return S.empty
  go (Or l r) = S.union `fmap` go l ⊛ go r
  go (Call p b cont) = do
    sols ← fmap S.toList (tableA p b)
    fmap S.unions (mapM (go . cont) sols)
```

Waarbij ⊛ afkomstig is uit de `Applicative` typeclass.

4.4 Conclusie

Het voornaamste resultaat van dit hoofdstuk is een denotationele semantiek $\mathcal{T}[\![\,.\,]\!]$ voor de abstracte `Table` syntax uit Hoofdstuk 3. De semantiek is de "handler"-helft van de effect handlers aanpak. Samen vormen `table` en $\mathcal{T}[\![\,.\,]\!]$ de *Tabulatie Monad*.

Deze monad gedraagt zich zoals niet-determinisme met Tabulatie. Beschouw Tabled Prolog als een opeenstapeling van effecten: Tabulatie, Backtracking(niet-determinisme), Unificatie en Negatie. De monad beschrijft de tabulatie en niet-determinisme van een Tabled Prolog programma. De overige componenten kunnen we vaak op een ad-hoc manier beschrijven (zoals in het voorbeeld met het predicaat p) of met hun eigen effect handler. De samenstelling van deze effect handlers bekomt een samenstelling van de effecten [24].

Door de functionele vorm van de semantiek ligt een vertaling in Haskell voor de hand. De grootste moeilijkheid ligt in het juist en efficiënt berekenen van het kleinste vaste punt. Op dit vlak is afhankelijkheidsanalyse interessant : de efficiëntie verbetert enorm, maar de correctheid blijft gegarandeerd.

Hoofdstuk 5

Aggregaten

De denotationele semantiek $\mathcal{T}[\![\,.\,]\!]$ uit Hoofdstuk 4 associeert een predicaat eenduidig met de verzameling van al zijn oplossingen. Vaak zijn we slechts geïnteresseerd in één specifiek element, in plaats van de hele verzameling. In veel gevallen is bijvoorbeeld enkel het optimale element relevant. Hoe kan een oplossingenverzameling herleid worden tot slechts één element? Dit is precies wat een aggregaat doet. Het vat een verzameling samen in één of meer elementen.

Het *subset sum* probleem illustreert dit verschil:

> Gegeven een lijst van gehele getallen, en een bepaald geheel getal s, vind dan alle niet-lege sublijsten waarvan de som gelijk is aan s.

```
type Pos = Int
sss :: Nondet m ⇒ [Int] → Open ((Int, Pos) → m [Int])
sss ss osss (s, i) = when (i ⩾ 0) $ do
  osss (s, i − 1) ∨ do
    x ← ss !? i
    when (s ≡ x) (return [x]) ∨ do xs ← osss (s − x, i − 1)
                                   return (x : xs)
when :: Nondet m ⇒ Bool → m a → m a
when b m = if b then m else fail
(!?) :: Nondet m ⇒ [a] → Int → m a
xs !? i | i < 0 ∨ i ⩾ length xs = fail
        | otherwise             = return (xs !! i)
program :: [Int] → Int → TabledProgram (Int, Pos) [Int] [Int]
program ss s = TP $ declare (sss ss) (λp → expression (p (s, length ss)))
```

when en !? zijn twee hulpfuncties. Zoals de naam zegt berekent *when* de zijn tweede argument alleen als zijn eerste argument waar is. De operator !? is een veilige manier om een lijst te benaderen: als de index buiten het geldige interval valt, dan *fail*t de functie. Het predicaat *sss* verwacht een lijst xs en een paar (s, i). Dit geeft alle niet-lege sublijsten van *take* $(i + 1)$ *xs* die sommeren tot s.

De eerdere semantiek $\mathcal{T}[\![\,.\,]\!]$ (*ts* genoemd in Haskell) geeft:

$$\begin{array}{l} \ggg ts \$ tpToTable \$ program\ [-1,2,3]\ 2 \\ [[2],[3,-1]] \\ \ggg ts \$ tpToTable \$ program\ [-1,2,3]\ 6 \\ [\,] \end{array}$$

Als we enkel geïntereseerd zijn in de kortste sublijst, kunnen we dit als volgt schrijven met de functies *minimumBy* (uit `Data.List`) en *on* (uit `Data.Function`)

$$\begin{array}{l} minimumBy :: Ord\ a \Rightarrow (a \to a \to Ordering) \to [a] \to Maybe\ a \\ on :: (b \to b \to c) \to (a \to b) \to a \to c \end{array}$$

De samenstelling $compare\ `on`\ length$ vergelijkt lijsten op basis van hun lengte.

$$\begin{array}{l} \ggg minimumBy\ (compare\ `on`\ length) \$ tpToTable \$ program\ [-1,2,3]\ 2 \\ Just\ [2] \end{array}$$

Alle langere lijsten worden dan voor niets berekend. Veronderstel daarentegen dat we beschikken over een aggregaat $t_{shortest}$ dat onmiddellijk de kortste lijst berekend,

$$\begin{array}{l} t_{shortest} :: Table\ (Predicate\ (Int, Pos)\ [Int])\ [Int]\ [Int]\ [Int] \to [Int] \\ t_{shortest}\ t = ... \\ \ggg tmax \$ tpToTable \$ program\ [-1,2,3]\ 2 \\ [2] \end{array}$$

dan kunnen de overbodige berekeningen vermeden worden. In feite worden hier alle deelbomen die met niet-optimale oplossingen overeenkomen weggesnoeid Dit gebeurt niet enkel op het hoogste niveau, maar ook op alle niveaus tussenin.

Een aggregaat is dus een nieuw effect, geïmplementeerd door een nieuwe semantische functie of effect handler $\mathcal{T}_{\mathcal{A}}[\![\,.\,]\!] : Table\ A \to \mathcal{A}$. Het beeld van de nieuwe semantiek is de waardeverzameling $\mathcal{A}$ van het aggregaat in kwestie, ter vervanging van $\mathcal{P}(A)$ Eerder moest $\mathcal{P}(A)$ al een tralie zijn omwille van technische redenen: om het bestaan van het kleinste vaste punt $lfp(.)$ te garanderen. Nu is er ook een intrinsieke reden: de l.u.b. ($\sqcup$) aggregeert uit een deelverzameling één uniek element. Er bestaat dus een conceptueel verband tussen aggregaten en tralies.

XSB-Prolog kent het gelijkaardige concept *Lattice Answer Subsumption*. Hierbij worden oude en nieuwe tabelitems samengevoegd met behulp van tralies.

5.1 Tralies als aggregaten

In plaats van $\mathcal{P}(A)$, de machtsverzameling van A, gebruikt de semantiek nu $\mathcal{A}$, de waardeverzameling van een bepaald aggregaatstype.

Net zoals $\mathcal{P}(A)$ in Hoofdstuk 4 moet $(\mathcal{A}, \sqsubseteq, \sqcup)$ een volledige tralie (eng. complete lattice) zijn:

Definitie 6. *Een partieel geordende verzameling* $(\mathcal{A}, \sqsubseteq)$[1] *is een volledige tralie* $(\mathcal{A}, \sqsubseteq, \sqcup)$ *als voor alle* $X \subseteq \mathcal{A}$ *de l.u.b.* $\sqcup X$ *gedefinieerd is, zodat:*

$$\forall z \in \mathcal{A} : \sqcup X \sqsubseteq z \Leftrightarrow \forall x \in X : x \sqsubseteq z.$$

De l.u.b. van $\emptyset$ *duiden we aan met een het symbool* $\bot$ *("bottom"), namelijk:*

$$\sqcup \emptyset = \bot.$$

De typeclass `Lattice` *stelt dit voor in Haskell:*

```
class Monoid l ⇒ Lattice l where
  (⊑) :: l → l → Bool
  ⊔   :: [l] → l
  ⊔   = foldr (◇) ε
  ⊥   :: l
  ⊥   = ⊔ []
```

De operators $\diamond$ *en* ε *komen uit de* `Monoid`*-typeclass.*

In Sectie 2.5.1 wordt dieper ingegaan op volledige tralies, als de lezer nog niet bekend is met tralies is het aangeraden om deze sectie te bekijken.

Iedere volledige tralie $(\mathcal{A}, \sqsubseteq, \sqcup)$ vormt dus ook een monoid $(\mathcal{A}, \bot, \sqcup)$. [2] Bovendien geldt dat $a \sqcup b = b \sqcup a$ en $a \sqcup a = a$, dus de monoid is *commutatief* en *idempotent*.

Merk op dat waar de wiskundige l.u.b. een verzameling verwacht, de Haskell functie $\sqcup$ gebruik maakt van gelinkte lijsten. Als de *Lattice* inderdaad idempotent en commutatief is, dan maakt dit geen verschil.

Daarnaast moet er een manier zijn om van A naar $\mathcal{A}$ te converteren en omgekeerd. Dit laat toe om effect handlers te schrijven die onafhankelijk zijn van een specifieke $\mathcal{A}$ Op die manier kunnen meerdere tralies één programma op verschillende manieren interpreteren.

Definitie 7. *We beschouwen twee functies* $(\uparrow) : A \to \mathcal{A}$ *en* $(\downarrow) : \mathcal{A} \to \mathcal{P}(A)$ *(uitgesproken als "coerce" en "uncoerce"). In typeclassvorm wordt dit:*

```
class Coerce i a where
  ↑ :: i → a
  ↓ :: a → [i]
```

Deze twee functies moeten onderling aan de volgende vereisten voldoen:

- $\downarrow$ *na* $\uparrow$ *geeft het oorspronkelijke element:*

$$\downarrow(\uparrow s) = \{s\},$$

[1] $\sqsubseteq$ is een orderelatie (reflexief, transitief en antisymmetrisch).

[2] De notatie $\sqcup$ wordt hier misbruikt voor zowel de binaire operatie in de monoid, als de unaire variant voor *Lattice*, die op verzamelingen van tralies werkt.

- $\uparrow$ *na* $\downarrow$ *geeft de oorspronkelijke tralie (als* $\mathcal{A}$ *een instance is van* `Lattice` *en* `Coerce`*):*

$$\sqcup\{\uparrow s \mid s \in \downarrow l\} = l.$$

De functie $\uparrow$ zet een element om in zijn bijbehorende tralie. Omgekeerd zet $\downarrow$ een tralie om in de verzameling van alle elementen waaruit de tralie bestaat.

Monoid homomorfismes Zowel $\mathcal{A}$ als $\mathcal{P}(A)$ vormen een volledige tralie en dus een monoid. Het is in deze context dus relevant om te bekijken hoe deze twee domeinen aan elkaar gerelateerd zijn door middel van een *homomorfismes* over monoids.

Definitie 8. *Zij* $(A, \varepsilon_A, \diamond_A)$ *en* $(B, \varepsilon_B, \diamond_B)$ *monoids, dan is de functie* $h : A \to B$ *een monoid homomorfisme van* A *naar* B *als*

$$h(\varepsilon_A) = \varepsilon_B$$
$$\forall a_1, a_2 \in A : h(a_1 \diamond_A a_2) = h(a_1) \diamond_B h(a_2)$$

Theorema 3. $h(A) = \sqcup\{\uparrow c \mid c \in A\}$ *is een monoid-homomorfisme van de monoid* $(\mathcal{P}(A), \emptyset, \cup)$ *naar de monoid* $(\mathcal{A}, \bot, \sqcup)$.

Voor het bewijs, zie Appendix B.

Tenslotte, als $\mathcal{A}$ zowel *Lattice* als *Coerce* instantiëert, dan *kan* $\downarrow$ een monoid-homomorfisme van $(\mathcal{A}, \bot, \sqcup)$ naar $(\mathcal{P}(A), \emptyset, \cup)$ zijn

$$\downarrow \bot = \emptyset,$$
$$\downarrow (a \sqcup b) = \downarrow a \cup \downarrow b,$$

Samen met Theorema 3 wil dit zeggen dat $\mathcal{A}$ isomorf is aan $\mathcal{P}(A)$.

Dit is zeker niet voor alle $\mathcal{A}$ gegarandeerd. Bijvoorbeeld, als er een s bestaat zodat $\uparrow s = \bot$, dan geldt: $\downarrow \bot = \downarrow (\uparrow s) = \{s\} \neq \emptyset$. Dus dan is $\sqcup$ geen monoid-homomorfisme (dit is eenvoudig op te lossen door een speciaal $\bot$ element toe te voegen). Een eenvoudige tralie waar dit het geval is $(\mathbb{N}, \leqslant, \max)$, met $A = \mathbb{N}$ en $\forall i \in \mathbb{N} : \uparrow i = i$, $\downarrow i = \{i\}$. Dan is $\bot = 0 = \uparrow 0$ en dus is $\downarrow \bot = \downarrow 0 = \{0\} \neq \emptyset$

Anderzijds impliceert $\downarrow (a \sqcup b) = \downarrow a \cup \downarrow b$ dat

$$a \sqsubseteq b \Rightarrow a \sqcup b = b \Rightarrow \downarrow a \subseteq \downarrow (a \sqcup b) = \downarrow b.$$

Dan volgt dat $table_{\mathcal{A}}$ (5.6) monotoon is:

$$s_i \sqsubseteq s_j \Rightarrow table_{\mathcal{A}}(s_i) \sqsubseteq table_{\mathcal{A}}(s_j)$$

Voor veel nuttige tralies is $table_{\mathcal{A}}$ niet monotoon, bijvoorbeeld voor de tralie $(\mathbb{N}, \leqslant, \max)$.

Aggregaten die geen volledige tralie vormen. Sommige aggregaten vormen wel een monoid maar geen volledige tralie, bijvoorbeeld de som, het aantal oplossingen of het gemiddelde. Veronderstel dat het aggregaat t_{count} het aantal oplossingen berekent:

$$
\begin{aligned}
&t_{count} :: \mathit{Table}\ p\ b\ c\ c \to \mathit{Count}\\
&t_{count} = ...\\
&\textbf{newtype}\ \mathit{Count} = \mathit{Count}\ \mathit{Int}\\
&\textbf{instance}\ \mathit{Monoid}\ \mathit{Count}\ \textbf{where}\\
&\quad \varepsilon = \mathit{Count}\ 0\\
&\quad \mathit{Count}\ s1 \diamond \mathit{Count}\ s2 = \mathit{Count}\ (s1 + s2)
\end{aligned}
$$

De operator $\diamond$ is niet idempotent. Bijgevolg kan $(\mathit{Count}, \varepsilon, \mathit{foldr}\ (\diamond)\ \varepsilon)$ geen volledige tralie zijn. Een ander voorbeeld is de lijst-monoid, waar $\diamond$ niet commutatief en niet idempotent is en dus zeker geen volledige tralie vormt.

$$
\begin{aligned}
&t_{all} :: \mathit{Table}\ p\ b\ c\ c \to [c]\\
&t_{all} = ...\\
&\textbf{instance}\ \mathit{Monoid}\ [\,]\ \textbf{where}\\
&\quad \varepsilon \quad\ \ = [\,]\\
&\quad \mathit{mappend} = (+\!\!+)
\end{aligned}
$$

De lijst-monoid is dus niet de juiste monoid om alle oplossingen te vinden (het is de set-monoid die hier gewenst is).

5.2 De semantische functie

Om getablede programma's met hun aggregaat te laten corresponderen is een nieuwe denotationele semantiek nodig. Deze nieuwe semantiek gelijkt sterk op de de semantiek $\mathcal{T}[\![\,.\,]\!]$ uit Hoofdstuk 4. Het is in die zin nuttig om in het achterhoofd te houden dat verzamelingen ook een volledige tralie $(\mathcal{P}(A), \emptyset, \cup)$ vormen en dat

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Coerce}\ A\ \mathcal{P}(A)\ \textbf{where}\\
&\quad \uparrow c = \mathit{singleton}\ c\\
&\quad \downarrow s = \mathit{toList}\ s
\end{aligned}
$$

een geldige instantie is voor *Coerce*.

De nieuwe semantische functie is $\mathcal{T}_{\mathcal{A}}[\![\,.\,]\!] : A \to \mathcal{A}$ (5.1).

$$
\begin{aligned}
&\mathcal{T}_{\mathcal{A}}[\![\,.\,]\!] : \mathit{Table}\ A \to \mathcal{A} && (5.1)\\
&\mathcal{T}_{\mathcal{A}}[\![\,\mathit{Pure}\ a\,]\!] = \uparrow a && (5.2)\\
&\mathcal{T}_{\mathcal{A}}[\![\,\mathit{Fail}\ \ \,]\!] = \bot && (5.3)\\
&\mathcal{T}_{\mathcal{A}}[\![\,\mathit{Or}\ l\ r\,]\!] = \mathcal{T}_{\mathcal{A}}[\![\,l\,]\!] \sqcup \mathcal{T}_{\mathcal{A}}[\![\,r\,]\!] && (5.4)\\
&\mathcal{T}_{\mathcal{A}}[\![\,\mathit{Call}\ p\ b\ \mathit{cont}\,]\!] = \sqcup\{\mathcal{T}_{\mathcal{A}}[\![\,\mathit{cont}(c)\,]\!] \mid c \in \downarrow \mathit{table}_{\mathcal{A}}(p, b)\} && (5.5)
\end{aligned}
$$

$\mathcal{T}_{\mathcal{A}}[\![\,.\,]\!]$ beeldt een succesvolle berekening *Pure a* af op $\uparrow a$, een mislukte berekening *Fail* op $\bot$, een niet-deterministische keuze *Or l r* op de l.u.b. van de resultaten van zijn keuzes en een *Call p b cont* op de l.u.b. van de continuaties van de call (voor elk element van $\downarrow table_{\mathcal{A}}(p, b)$ is er een continuatie).

De hulp-functie $table_{\mathcal{A}}(p, b)$ (5.6) berekend het resultaat van een getablede call.

$$table_{\mathcal{A}} : Predicate\ A\ A \times A \to \mathcal{A} \tag{5.6}$$
$$table_{\mathcal{A}}(p, b) = \sqcup \mathcal{KC}\left(\lambda s.\lambda(q, k).\Big(\underbrace{s(q, k)}_{\text{oud}} \sqcup \underbrace{\mathcal{C}_{\mathcal{A}}[\![\, q(k)\,]\!](s)}_{\text{nieuw}}\Big)\right)(p, b)$$

De functie $table_{\mathcal{A}}$ (5.6) gebruikt de semantische functie $\mathcal{C}_{\mathcal{A}}[\![\,.\,]\!]$ (5.7) voor het iteratief oplossen van recursieve calls (analoog aan $\mathcal{C}[\![\,.\,]\!]$ in Hoofdstuk 4). Een anonieme λ-functie wikkelt rond $\mathcal{C}_{\mathcal{A}}[\![\,.\,]\!]$ en voegt het oude resultaat $s(q, k)$ en nieuwe resultaat $\mathcal{C}_{\mathcal{A}}[\![\, q(k)\,]\!](s)$ samen.

Er bestaat niet altijd het kleinste vaste punt voor deze λ-functie. In tegenstelling tot *table* uit Sectie 4.1 kan *lfp*(.) dus niet gebruikt worden. Maar we zien wel dat een argument kleiner moet zijn dan zijn afbeelding onder de λ-functie. Opeenvolgende oproepen $\bot, f(\bot), f(f(\bot)), \ldots$ vormen dus een ketting $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots$. Dit is een opwaartse kleene ketting ($\mathcal{KC}(.)$, zie Definitie 4).

De l.u.b. van deze ketting is dus wel goed gedefiniëerd. Onder bepaalde voorwaarden (monotoniciteit en continuiteit, zie Theorema 2) is deze l.u.b. eveneens het kleinste vaste punt. Deze voorwaarden zijn wel of niet vervuld afhankelijk van de gebruikte tralie en het programma dat door de semantiek wordt geëvalueerd. In tegenstelling tot Hoofdstuk 4 is er dus geen algemeen geldende monotoniciteit en continuiteit.

Tenslotte is er nog $\mathcal{C}_{\mathcal{A}}[\![\,.\,]\!]$, de semantische functie voor recursieve calls.

$$\mathcal{C}_{\mathcal{A}}[\![\,.\,]\!] : Table\ A \to (Predicate\ A\ A \times A \to \mathcal{A}) \to A \tag{5.7}$$
$$\mathcal{C}_{\mathcal{A}}[\![\ Pure\ a\ ]\!](s) = \uparrow a \tag{5.8}$$
$$\mathcal{C}_{\mathcal{A}}[\![\ Fail\ \ ]\!](s) = \bot \tag{5.9}$$
$$\mathcal{C}_{\mathcal{A}}[\![\ Or\ l\ r\ ]\!](s) = \mathcal{C}_{\mathcal{A}}[\![\, l\,]\!](s) \sqcup \mathcal{C}_{\mathcal{A}}[\![\, r\,]\!](s) \tag{5.10}$$
$$\mathcal{C}_{\mathcal{A}}[\![\ Call\ p\ b\ cont\ ]\!](s) = \sqcup\{\underbrace{\mathcal{C}_{\mathcal{A}}[\![\, cont(c)\,]\!](s)}_{\text{continuaties}} \mid \underbrace{c \in \downarrow s(p, b)}_{\text{oude resultaten}}\} \tag{5.11}$$

Het resultaat van een succesvolle berekening is opnieuw $\uparrow a$, het resultaat van een mislukte berekening is $\bot$. Het resultaat van een niet-deterministische keuze is de l.u.b. van de resultaten van de keuze. Het resultaat van een call *Call b cont* is gelijk aan de l.u.b. van de continuatie van alle gekende resultaten $\downarrow s(p, b)$.

De eerder gesuggereerde vergelijking tussen $\mathcal{T}[\![\,.\,]\!]$ en $\mathcal{T}_{\mathcal{A}}[\![\,.\,]\!]$ kan nu concreet gemaakt worden door $\mathcal{P}(A)$ te substitueren voor $\mathcal{A}$ in $\mathcal{T}_{\mathcal{A}}[\![\,.\,]\!]$. Dan is $\mathcal{C}_{\mathcal{P}(\mathcal{A})}[\![\,.\,]\!] = \mathcal{C}[\![\,.\,]\!]$ en uit Theorema 2 volgt dat $table_{\mathcal{P}(A)}(b) = table(b)$. Dan volgt ook dat $\mathcal{T}_{\mathcal{P}(\mathcal{A})}[\![\,.\,]\!] = \mathcal{T}[\![\,.\,]\!]$.

5.3 Vergelijking met Prolog

Standaard gedraagt Tabled Prolog zich zoals de semantische functie $\mathcal{T}[\![\,.\,]\!]$ uit Hoofdstuk 4. Een resultaat wordt aan de oplossingstabel toegevoegd als het nog niet in de tabel aanwezig is (meer precies: als er nog geen *variant* van de oplossing aanwezig is).

XSB-Prolog en Lattice Answer Subsumption XSB-Prolog laat toe dat de programmeur een ander gedrag kiest wanneer een element aan de tabel wordt toegevoegd.. Met het `table`-directive kan de programmeur een predicaat aanduiden. Dit predicaat wordt dan gebruikt om bestaande en nieuwe oplossingen te combineren:

```
:- table sp(_,_,lattice(min/3)).
```

In dit Prolog fragment wordt een predicaat `sp` getabled. Wanneer een oplossing $sp(A1, B1, C1)$ wordt gevonden en een oplossing $sp(A2, B2, C2)$ met $A1 = A2$, $B1 = B2$ staat al in de tabel, dan zal de l.u.b. van $C1$ en $C2$ onthouden worden. Deze techniek heet *Lattice Answer Subsumption* [20].

XSB ondersteund ook een andere vorm van answer subsumption, *Partial Order Subsumption*. Dit wordt gespecificeerd met het `table`-directive.

```
:- table sp(_,_,po(</2)).
```

Een oplossing $sp(A, B, C)$ wordt alleen aan de table toegevoegd als er geen oplossing $sp(A, B, C')$ in de tabel is zodat $C < C'$. Merk op dat `</2` een *paritiële orde* kan zijn.

Met de juiste tralie *Min a*, kan een effect dat identiek is aan *Lattice Answer Subsumption* bereikt worden met $\mathcal{T}_{Min\ a}[\![\,.\,]\!]$.

data *Min a* = *Minimum* | *Min a* **deriving** (*Eq*, *Ord*)
instance *Coerce a* (*Min a*) **where**
  $\uparrow$ *a* = *Min a*
  $\downarrow$ *Minimum* = []
  $\downarrow$ (*Min a*) = [*a*]
instance *Ord a* $\Rightarrow$ *Monoid* (*Min a*) **where**
  ε = *Minimum*
  mappend = *min*
instance *Ord a* $\Rightarrow$ *Lattice* (*Min a*) **where**
  Minimum $\sqsubseteq$ *a* = *True*
  a $\sqsubseteq$ *Minimum* = *False*
  Min a $\sqsubseteq$ *Min b* = $b \leqslant a$
  $\sqcup$ *l* = *foldr min Minimum l*

De semantische functie $\mathcal{T}_{\mathcal{A}}[\![\,.\,]\!]$ komt dus overeen met Tabled Prolog met Lattice Answer Subsumption (modulo unificatie, negatie of cut).

Mode-directed tabling Andere Prolog systemen zoals YapTab (Yap Prolog met tabulatie)[18], B-Prolog[25] en ALS-Prolog[6] breiden Tabled Prolog op een andere manier uit. Deze systemen bieden *tabling-modes* aan. Dit wil zeggen dat men voor bepaalde argumenten van een getablede predicaat kan selecteren hoe nieuwe oplossingen aan de tabel worden toegevoegd. De flexibiliteit van deze methode is dus sterk afhankelijk van het aantal aangeboden modes.

In alle systemen worden de gebruikte modes aangeduid met het `table`-directive, bijvoorbeeld voor een predicaat `sp/3`, in YapTab:

```
:- table sp(index, index, min).
```

Dit geeft aan dat de eerste twee argumenten de mode *index* gebruiken en het laatste argument de mode *min*.

YapTab heeft het grootste aanbod aan tabling modes: *index*, *first*, *last*, *min*, *max*, *sum* en *all*.

B-Prolog ondersteunt 5 modes: + (invoer), −(uitvoer), *min* (minimum), *max* (maximum) en *nt* (niet-getabled). Input en output hebben hier betrekking op hoe B-Prolog controleerd of een variante van een oplossing al in de tabel is. Hierbij worden dan enkel invoer in rekening gebracht (min en max worden verondersteld uitvoer te zijn). Daarnaast voorziet B-Prolog in een kardinaliteitsconstraint, die het aantal oplossingen in de tabel beperkt.

ALS-Prolog voorziet in 3 tabling modes: + (geïndexeerd), − (niet-geïndexeerd, alleen de eerste oplossing wordt onthouden) en @ (niet-geïndexeerd, alle oplossingen worden onthouden).

YapTab biedt dus strikt meer opties dan de andere twee systemen. XSB-Prolog kan met *Lattice Answer Subsumption* alle zeven table modes van YapTab nabootsen, op functioneel vlak. De functionaliteit van de Haskell semantiek is dus gelijkaardig aan XSB en YapTab.

Wat efficiëntie betreft, blijkt uit benchmarks[18] dat YapTab de snelste is voor de problemen die van de aangeboden modes gebruik maken (vergeleken met B-Prolog en XSB).

Omdat *sum* niet idempotent is, kan deze mode niet correct geïmplementeerd worden met tralies. Een implementatie met $\sqcup = \mathit{foldr}\ (+)\ 0$ en $\bot = 0$ overtreedt de wetten die aan *Lattice* zijn opgelegd.[3] Het resultaat van $\mathcal{T}_{\mathcal{SUM}}[\![\,.\,]\!]$ is dan soms juist en soms niet. *Lattice Answer Subsumption* stoot op een gelijkaardig probleem: het resultaat is afhankelijk van de gebruikte scheduling strategie.

5.4 Uitvoerbaarheid

De semantische functie $\mathcal{T}_{\mathcal{A}}[\![\,.\,]\!]$ laat zich makkelijk vertalen in Haskell:[4]

$$\begin{aligned} &\mathit{tl} :: \forall\ b\ c\ l\,.\,(\mathit{Lattice}\ l, \mathit{Coerce}\ c\ l) \Rightarrow \mathit{Table}\ (\mathit{Predicate}\ b\ c)\ b\ c\ c \to l \\ &\mathit{tl}\ (\mathit{Pure}\ c) \qquad\quad = \uparrow c \end{aligned}$$

[3] Bijvoorbeeld $1 \leqslant 2 \leqslant 2$, dus $\sqcup\{1,2\} \leqslant 2$, dus $1 + 2 = 3 \leqslant 2$↯.

[4] Om deze code te compileren is wel de `ScopedTypeVariables`-uitbreiding van de Glasgow Haskell Compiler nodig [1].

$$
\begin{aligned}
&tl\ \ Fail &&= \bot \\
&tl\ (Or\ l\ r) &&= \sqcup\,[tl\ l, tl\ r] \\
&tl\ (Call\ p\ b\ cont) &&= \sqcup\,[tl\ (cont\ c) \mid c \leftarrow\!\downarrow\ (tableA\ p\ b :: l)] \\
&tableA :: (Lattice\ l, Coerce\ c\ l) \Rightarrow Predicate\ b\ c \to b \to l \\
&tableA = lfp\ (\lambda s\ q\ k \to \sqcup\,[s\ q\ k, cl\ (pBody\ q\ k)\ s])\ (\lambda q\ k \to \bot)
\end{aligned}
$$

De semantische functie $\mathcal{C}_{\mathcal{A}}[\![\,.\,]\!]$ kent eveneens een vrij letterlijke Haskell variant *cl*:

$$
\begin{aligned}
&cl :: (\ Lattice\ l, Coerce\ c\ l) \\
&\quad \Rightarrow Table\ (Predicate\ b\ c)\ b\ c\ c \to (Predicate\ b\ c \to b \to l) \to l \\
&cl\ (Pure\ c) &&s = \uparrow c \\
&cl\ \ Fail &&s = \bot \\
&cl\ (Or\ l\ r) &&s = \sqcup\,[cl\ l\ s, cl\ r\ s] \\
&cl\ (Call\ p\ b\ cont) &&s = \sqcup\,([cl\ (cont\ c)\ s \mid c \leftarrow\!\downarrow\ (s\ p\ b)])
\end{aligned}
$$

De functie *lfp* berekent het kleinste vaste punt. De implementatie hiervan laten we bewust leeg. Net zoals in Sectie 4.3.1 kan een implementatie met *afhankelijkheidsanalyse*[5] gebruikt worden om dit kleinste vaste punt te berekenen.

$$lfp :: (a \to a) \to a \to a$$

5.5 Voorbeelden

De volgende voorbeelden demonstreren het nut van aggregaten voor het oplossen van bepaalde problemen. Aggregaten zijn elementair voor het efficiënt oplossen van problemen die gebruik maken van *Dynamisch Programmeren*. Voor deze strategie is het van groot belang dat alleen de relevante oplossingen worden bijgehouden.

5.5.1 Subset Sum

> Gegeven een lijst van gehele getallen, en een bepaald geheel getal s, vind dan de *kortste* niet-lege sublijst waarvan de som gelijk is aan s.

Deze sectie geeft een oplossing voor het subset sum probleem uit de inleiding aan de hand van aggregaten. Het predicaat *sss* werd reeds in de inleiding besproken:

$$
\begin{aligned}
&\textbf{type}\ Pos = Int \\
&sss :: Nondet\ m \Rightarrow [Int] \to Open\ ((Int, Pos) \to m\ [Int]) \\
&sss\ ss\ osss\ (s, i) = when\ (i \geqslant 0)\ \$\ \textbf{do} \\
&\quad osss\ (s, i-1) \vee \textbf{do} \\
&\qquad x \leftarrow ss\ !?\ i \\
&\qquad when\ (s \equiv x)\ (return\ [x]) \vee \textbf{do}\ xs \leftarrow osss\ (s-x, i-1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad return\ (x : xs)
\end{aligned}
$$

[5] De code hiervoor staat in de online repository https://bitbucket.org/AlexanderV/thesis

$$sssProgram :: [Int] \to Int \to TabledProgram\ (Int, Pos)\ [Int]\ [Int]$$
$$sssProgram\ ss\ s = TP\ \$\ declare\ (sss\ ss)\ (\lambda p \to expression\ (p\ (s, length\ ss)))$$

De kortste lijst is hier het gewenste resultaat. Daarom definiëren we een *Lattice*, *Shortest* bovenop lijsten van *Int*s, zodat $\sqsubseteq$ steeds de kortste lijst selecteert, dan is $\bot$ een oneindig lange lijst. Dit wordt voorgesteld door de constructor *InfList*.

$$\textbf{data}\ Shortest = InfList \mid Shortest\ [Int]$$
$$\textbf{instance}\ Eq\ Shortest\ \textbf{where}$$
$$\begin{aligned} InfList &\equiv InfList &&= True \\ Shortest\ a &\equiv Shortest\ b &&= length\ a \equiv length\ b \\ a &\equiv b &&= False \end{aligned}$$
$$\textbf{instance}\ Monoid\ Shortest\ \textbf{where}$$
$$\begin{aligned} \varepsilon &= InfList \\ mappend\ a\ b &= \textbf{if}\ a \sqsubseteq b\ \textbf{then}\ b\ \textbf{else}\ a \end{aligned}$$
$$\textbf{instance}\ Lattice\ Shortest\ \textbf{where}$$
$$\begin{aligned} InfList &\sqsubseteq a &&= True \\ a &\sqsubseteq InfList &&= False \\ Shortest\ a &\sqsubseteq Shortest\ b &&= length\ a \geqslant length\ b \\ \sqcup &= foldr\ (\diamond)\ \varepsilon \\ \bot &= \varepsilon \end{aligned}$$
$$\textbf{instance}\ Coerce\ [Int]\ Shortest\ \textbf{where}$$
$$\begin{aligned} \uparrow s &= Shortest\ s \\ \downarrow InfList &= [\,] \\ \downarrow (Shortest\ s) &= [s] \end{aligned}$$

Bijvoorbeeld, voor een lijst $[-1, 2, 3]$ geeft dit

$$\ggg tl\ \$\ tpToProgram\ \$\ sssProgram\ [-1, 2, 3]\ 2 :: Shortest\ Shortest\ [2]$$
$$\ggg tl\ \$\ tpToProgram\ \$\ sssProgram\ [-1, 2, 3]\ 4 :: Shortest\ Shortest\ [3, 2, -1]$$
$$\ggg tl\ \$\ tpToProgram\ \$\ sssProgram\ [-1, 2, 3]\ 6 :: Shortest\ InfList$$

5.5.2 Knapzak (Knapsack)

> Gegeven een rugzak met een maximale capaciteit (in de vorm van een maximaal gewicht) en een lijst van items met een gewicht en een waarde, welke selectie van items leidt tot een maximale waarde, zonder het maximale gewicht te overschrijden?

Het *onbegrensde (eng. unbounded) knapzak probleem*, met maximum gewicht W en strikt positieve gewichten w_i en strikt positieve waarden v_i is gedefiniëerd als

$$\max_{x_1,\ldots,x_n} \sum_{i=1}^{n} x_i.v_i \text{ zodat } \sum_{i=1}^{n} x_i.w_i \leqslant W$$
$$x_i \geqslant 0 \text{ voor } i = 1,\ldots,n$$
$$v_i > 0 \text{ voor } i = 1,\ldots,n$$
$$w_i > 0 \text{ voor } i = 1,\ldots,n$$

In het onbegrensde knapzak probleem wordt een gebruikt item opnieuw op de lijst van beschikbare items geplaatst, zodat het meerdere keren gebruikt kan worden.

Veronderstel dat reeds een item gekozen is. Hoe moeten we de rugzak verder aanvullen zodat de overgebleven capaciteit optimaal benut wordt? Met andere woorden: hoe vullen we een rugzak met het overgebleven gewicht zo optimaal mogelijk? Dit is een voorbeeld van een probleem dat met *Dynamisch Programmeren* wordt opgelost: een probleem aanpakken door kleinere instanties van het probleem op te lossen.

We demonstreren dit idee in XSB-Prolog met *Lattice Answer Subsumption*:

```
% item(W,V) where W is weight, V is value
item(1, 1).
item(1, 2).
item(5, 20).

:- table knapsack(lattice(kncompare/3),_).

% kncompare(A,B,C) where C is the largest of A and B
kncompare(K1, K2, K1) :-
  knwv(K1, _, V1),
  knwv(K2, _, V2),
  V1 > V2, !.
kncompare(_, K2, K2).

% knwv(K, W, V) where W is the weight of K, V
% is the value of K
knwv([], 0, 0).
knwv([(WI,VI)|IS], W, V) :-
  knwv(IS, WIS, VIS),
  W is WI + WIS,
  V is VI + VIS.

% knapsack(KS, W) where KS is a knapsack s.t.
% the weight of KS doesn't exceed W.
knapsack([], _).
knapsack([(WI, VI)|Items], W) :-
  item(WI,VI),
```

```
    WItems is W - WI,
    WItems >= 0,
    knapsack(Items, WItems).
```

Dit levert dan bijvoorbeeld het volgende resultaat:

```
?- knapsack(K, 7), knwv(K, W, V).

K = [(5,20), (1,2), (1,2)]
W = 7
V = 24
```

Nu modelleren we deze oplossing in Haskell. Een *Item* is een datatype me een gewicht en een waarde:

```
data Item = Item {weight :: Int, value :: Int}
```

Een knapzak is een lijst van items.

```
newtype Knapsack = Knapsack {ksItems :: [Item]}
```

Van een knapzak kunnen we ook het totale gewicht en de totale waarde berekenen.

```
ksWeight :: Knapsack → Int
ksWeight (Knapsack ks) = sum (map weight ks)
ksValue :: Knapsack → Int
ksValue (Knapsack ks) = sum (map value ks)
```

Twee *Knapsack*-en kunnen we vergelijken voor gelijkheid $\equiv$ en orde $\leqslant$:

```
instance Eq Knapsack where
  k1 ≡ k2 = ksWeight k1 ≡ ksWeight k2 ∧ ksValue k1 ≡ ksValue k2
instance Ord Knapsack where
  k1 ⩽ k2 = v1 < v2 ∨ (v1 ≡ v2 ∧ w1 ⩽ w2) where
    v1 = ksValue k1
    v2 = ksValue k2
    w1 = ksWeight k1
    w2 = ksWeight k2
```

De volgende functie lost het knapsack probleem op:

```
unboundedKnapsack :: Nondet m
                  ⇒ m Item
                  → Open (Int → m Knapsack)
unboundedKnapsack is super w
  | w ⩽ 0      = return (Knapsack [])
  | otherwise = return (Knapsack []) ∨ do
                  item ← is
```

waarde(v_i)	gewicht (w_i)
1	2
2	2
5	20

Tabel 5.1: Drie voorbeelditems voor het knapzakprobleem.

```
  let wI = weight item
  if wI > w then fail else do
    Knapsack ks ← super (w − wI)
    return (Knapsack (item : ks))
```

De functie *unboundedKnapsack* verwacht een keuze aan items (*Table p b c Item*) en een maximum gewicht. De lege knapsack (*Knapsack* [ ]) is een geldige knapsack voor ieder gewicht. Als het gewicht positief is, dan probeert het programma elk item en vult recursief het overgebleven gewicht met de beste mix van items.

De *Lattice* is in dit geval *Max*, waarbij $\sqsubseteq$ steeds het grootste element selecteert. Het kleinste element in de tralie is *Bottom*.

```
data Max a = Bottom | Max a deriving (Eq, Ord)
unMax :: Max a → Maybe a
unMax Bottom  = Nothing
unMax (Max a) = Just a

instance Ord a ⇒ Monoid (Max a) where
  ε = Bottom
  mappend = max

instance Ord a ⇒ Lattice (Max a) where
  a ⊑ b = a ⩽ b
  ⊔     = foldr (⋄) ε
  ⊥     = ε

instance Coerce a (Max a) where
  ↑ a          = Max a
  ↓ Bottom  = [ ]
  ↓ (Max a) = [a]
```

De functie *knapsackProgram* sluit de recursie.

```
knapsackProgram :: [Item] → Int → TabledProgram Int Knapsack Knapsack
knapsackProgram is w = TP $
  let isT = foldr (∨) fail $ map return $ is
  in declare (unboundedKnapsack isT) (λp → expression (p w))
```

Enkele voorbeelden, met 3 items (opgelijst in Tabel 5.1):

```
items :: [Item]
items = [Item 1 2, Item 2 2, Item 5 20]
```

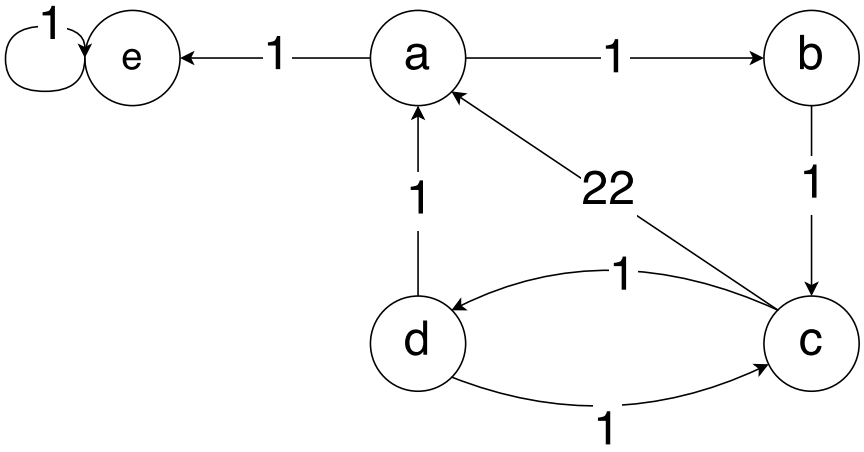
Dit geeft de volgende uitvoer:

```
⋙ unMax $ tl $ tpToTable $ knapsackProgram items 5
Just (Knapsack [Item 5 20])

⋙ unMax $ tl $ tpToTable $ knapsackProgram items 7
Just (Knapsack [Item 5 20, Item 2 2])
```

5.5.3 Kortste Pad in een cyclische grafe

Gegeven een gerichte grafe $G = (V, E)$ met vertices V en edges E, waar vertices voorzien zijn van labels. Wat is de lengte van het kortste pad tussen een gegeven vertex en een andere vertex gelabeld met een geven label?



Figuur 5.1: Een grafe met 5 nodes a, b, c, d, e.

In Prolog stellen we de grafe in Figuur 5.1 voor met het predicaat `edge/3`. Het predicaat `sp/3` heeft dezelfde signatuur en berekend het kortste pad tussen twee vertices.

```
% a directed weighted graph: edge(Node1,Node2,Cost).
edge(a,b,1).
edge(a,e,1).
edge(b,c,1).
edge(c,a,22).
edge(c,d,1).
edge(d,a,1).
edge(d,c,1).
edge(e,e,1).
```

```
min(X,Y,Z) :- Z is min(X,Y).

% shortest path predicate
:- table sp(_,_,lattice(min/3)).
sp(N,N,0).
sp(N1,N2,C) :- edge(N1,N2,C).
sp(N1,N2,Cost) :- sp(N1,N3,C1), edge(N3,N2,C2), Cost is C1 + C2.
```

Dan kunnen we de volgende queries stellen:

```
?-sp(a, a, C).
C = 0;
no

?-sp(b, a, C).
C = 3;
no

?-sp(c, a, C).
C = 2;
no

?-sp(d, a, C).
C = 1;
no

?-sp(e, a, C).
no
```

In Haskell kunnen we vertices voorstellen met het datatype *Node*, dat bestaat uit een label (van type *Char*), en de structuur van de grafe, voorgesteld door een adjacency list (type $[(Node, Distance)]$).

$$\textbf{data}\ Node = Node\ \{label :: Char, neighbours :: [(Node, Distance)]\}$$
$$neighbour :: Nondet\ m \Rightarrow Node \rightarrow m\ (Node, Distance)$$
$$neighbour\ n = foldr\ (\vee)\ fail\ \$\ map\ return\ \$\ neighbours\ n$$

Omdat Haskell argumenten van een functie niet-strikt evalueert, kunnen we hiermee cyclische grafen definiëren, zoals de grafe in Figuur 5.1:

$$\begin{aligned} &a, b, c, d, e :: Node \\ &[a, b, c, d, e] = [&&Node\ \texttt{'a'}\ [(b, 1), (e, 1)], \\ &&&Node\ \texttt{'b'}\ [(c, 1)], \\ &&&Node\ \texttt{'c'}\ [(a, 22), (d, 1)], \\ &&&Node\ \texttt{'d'}\ [(c, 1), (a, 1)], \\ &&&Node\ \texttt{'e'}\ [(e, 1)]] \end{aligned}$$

Het predicaat `sp/2` wordt vervangen door *distanceTo*:

```
distanceTo :: Nondet m ⇒ Char → Open (Node → m Distance)
distanceTo l super n | label n ≡ l = return 0
                     | otherwise   = do (n0, d0) ← neighbour n
                                        dn       ← super n0
                                        return (d0 + dn)
```

distanceTo l berekend de afstand tot een node met label *l* op een recursieve manier: een pad van een node *a* naar een node *b* moet eerst via een edge naar een naburige node gaan. Vervolgens wordt een pad van de naburige node naar node *b* gevolgd.

De code hieronder definieerd *Distance*, een *Lattice* die het kortste pad berekend.

```
data Distance = Distance Int | InfDistance deriving (Eq, Ord)

instance Num Distance where
  InfDistance + b            = InfDistance
  a           + InfDistance  = InfDistance
  Distance d1 + Distance d2  = Distance (d1 + d2)
  fromInteger                = Distance . fromInteger

instance Monoid Distance where
  ε       = InfDistance
  mappend = min

instance Lattice Distance where
  d1 ⊑ d2 = d2 ⩽ d1
  ⊔       = foldr (⋄) ε
  ⊥       = ε

instance Coerce Distance Distance where
  ↑ a =  a
  ↓ a = [a]
```

Merk op hoe $\sqsubseteq$ gelijk is aan $\leqslant$ met de plaats van de argumenten verwisseld. Dit wil zeggen dat hoe kleiner de afstand, hoe *beter* de oplossing.

De volgende voorbeelden berekenen de afstand tot de vertex *a* van de vertices *a*, *b*, *c*, *d* en *e* in de duidelijk cyclische grafe in Figuur 5.1.

```
distanceToProg :: Char → Node → TabledProgram Node Distance Distance
distanceToProg l n = TP $ declare (distanceTo l) (λp → expression (p n))


⋙ tl $ tpToTable $ dijkstraProg 'a' a :: Distance
Distance 0


⋙ tl $ tpToTable $ dijkstraProg 'a' b :: Distance
Distance 3
```

$$\ggg\ tl\ \$\ tpToTable\ \$\ dijkstraProg\ \texttt{'a'}\ c :: Distance$$
$$Distance\ 2$$

$$\ggg\ tl\ \$\ tpToTable\ \$\ dijkstraProg\ \texttt{'a'}\ d :: Distance$$
$$Distance\ 1$$

$$\ggg\ tl\ \$\ tpToTable\ \$\ dijkstraProg\ \texttt{'a'}\ e :: Distance$$
$$InfDistance$$

Voor deze formulering van *distanceTo* is het gebruik van tralies strikt noodzakelijk! Omdat de grafe cyclisch is bestaan er tussen bijvoorbeeld d en c oneindig veel paden met een verschillend gewicht (achtereenvolgens met gewicht $1, 3, 5, 7, \ldots$). De oplossingsverzameling voor een dergelijk programma is dus zelfs met tabulatie oneindig groot, dus is het onmogelijk om hiervan in een eindige hoeveelheid tijd een aggregaat te berekenen. Met de *tl* effect handler kan de oplossing wel gevonden worden.

Als het *distanceTo* anders geformuleerd wordt, is een oplossing met (gewone) tabulatie wel mogelijk: *distanceToDP* herformuleert het probleem met behulp van dynamisch programmeren:

$$\begin{aligned}
&distanceToDP :: Nondet\ m \Rightarrow Char \to Open\ ((Node, Int) \to m\ Distance)\\
&distanceToDP\ l\ super\ (n, i) \mid label\ n \equiv l = return\ 0\\
&\quad \mid i \equiv 0 \qquad\qquad\qquad\quad = fail\\
&\quad \mid otherwise \qquad\qquad\qquad\quad = \mathbf{do}\ (n0, d0) \leftarrow neighbour\ n\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ dn \qquad\ \leftarrow super\ (n0, i - 1)\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ return\ (d0 + dn)
\end{aligned}$$

Het idee is hier dat *distanceToDP* `'a'` voor parameter (d, i) maar paden zoekt die hoogstens i edges passeren. Als er N vertices in de grafe zitten, dan moet *distanceToDP* enkel maar voor $i \leqslant N - 1$ uitgevoerd worden (als $N - 1$ is de lengte van het langste simpel[6] pad dat kan voorkomen in eender welke grafe met N vertices).

5.6 Conclusie

Aggregaten laten toe om bepaalde effecten te bekomen op een efficiëntere manier. Met name wanneer we geïntreseerd zijn in een beperkt aantal elementen uit de hele oplossingsverzameling. In sommige gevallen (zie Sectie 5.5.3) zijn zelfs oplossingen mogelijk die met gewone tabulatie niet konden worden berekend.

Sommige aggregaten vormen een volledige tralie. In dit geval kunnen we de semantiek $\mathcal{T}_{\mathcal{A}}[\![\,.\,]\!]$ gebruiken om het gewenste effect te bekomen, zonder dat de interface *Nondet* of de syntax *Table* verandert.

Hetzelfde gebeurt in Prolog: de structuur van het programma blijft dezelfde, maar de `table`-directive wordt aangepast met *Lattice Answer Subsumption* of extra *table-modes*.

[6] Een simpel pad is een pad dat geen cycle bevat. Omdat de gewichten strikt positief zijn, heeft een simpel pad een kleiner gewicht dan een gelijkaardig pad dat wel een cycle bevat.

Hoofdstuk 6

Benchmarks

In dit hoofdstuk worden de verschillende Haskell implementaties met elkaar vergeleken op het gebied van uitvoeringstijd. De verschillende implementaties moeten met elkaar wedijveren in drie benchmarks: het knapzakprobleem, het berekenen van strongly connected components (SCCs) in een cyclische grafe en het berekenen van de kortste paden in een grafe.

Het doel van deze selectie is om verschillende toepassingsgebieden te belichten: het knapzakprobleem is een voorbeeld van dynamisch programmeren, opgelost met aggregaten; het berekenen van SCCs in een cyclische grafe is iets waarvoor tabulatie een natuurlijke oplossing is; en het berekenen van de kortste paden is een voorbeeld van een niet-dynamisch-programmeren probleem dat ook met aggregaten kan worden opgelost.

De resultaten bewijzen het nut van de afhankelijkheidsanalyse en aggregaten. Het is *niet* de bedoeling om een idee te krijgen van de absolute performantie (i.e. in vergelijking met implementaties van dezelfde problemen in Haskell, Prolog, of andere systemen).

Alle benchmarks werden uitgevoerd op een Linux (3.19) Laptop met een Core i5-2410M 2.3-2.9 GHz, 4 GB RAM, gecompileerd met `-O2` en GHC 7.6.3. De relevante broncode bestanden zijn te vinden in de bench/-map van het online repository.[1]

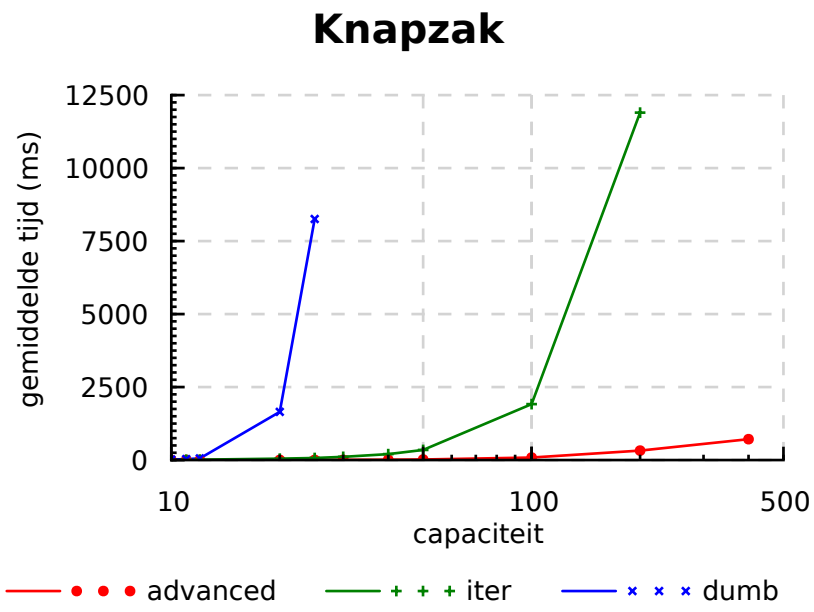
6.1 Knapzak

Deze benchmark omvat het oplossen van het knapzak probleem, op een vaste set van 35 pseudo-random items (met een gewicht en waarde tussen 1 en 100). Het probleem wordt opgelost voor een maximaal gewicht ("capaciteit") van 10, 11, 12, 20, 25, 30, 40, 50, 100, 200 en 400.

Aan deze benchmark nemen 3 implementaties deel:

- **iter** Een naïeve implementatie die gebruik maakt van aggregaten, maar het vaste punt berekent zonder afhankelijkheidsanalyse.

[1] https://bitbucket.org/AlexanderV/thesis



Figuur 6.1: Het verloop van de Knapzak benchmark.

$$knapsackIter :: [Item] \to Int \to Knapsack$$
$$knapsackIter\ is\ w = unMax\ \$\ iterTl\ \$\ tpToTable\ \$\ knapsackProgram\ is\ w$$

waarbij *iterTl* de effect handler is en *unMax*, *knapsackProgram* en *tpToTable* zoals in Hoofdstuk 5.

- **advanced** Een implementatie die gebruik maakt van aggregaten en het vaste punt berekent met afhankelijkheidsanalyse.

  $$knapsackAdv :: [Item] \to Int \to Knapsack$$
  $$knapsackAdv\ is\ w = unMax\ \$\ tl\ \$\ tpToTable\ \$\ knapsackProgram\ is\ w$$

  waarbij *tl* de effect handler is die tralies en de afhankelijkheidsanalyse ondersteunt.

- **dumb** Een implementatie die geen gebruik maakt van aggregaten. In plaats daarvan wordt de gewone semantiek met afhankelijheidsanalyse gebruikt. Deze implementatie berekent eerst alle mogelijke knapzakken van een bepaalde capaciteit. Uit alle knapzakken wordt de beste knapzak geselecteerd.

  $$knapsackDumb :: [Item] \to Int \to Knapsack$$
  $$knapsackDumb\ is\ w = maximum\ \$\ ts\ \$\ tpToTable\ \$\ knapsackProgram\ is\ w$$

  waarbij *ts* de effect handler is die met de normale semantiek overeenkomt en *maximum* het maximale element van een verzameling berekent.

  Deze situatie komt overeen met het geval waarin aggregaten niet beschikbaar zouden zijn.

De *dumb* en *iter* implementaties worden niet voor alle capaciteiten uitgevoerd omdat zijn anders de grafiek (Figuur 6.1) te veel zouden vertekenen.

Uit de resultaten (Figuur 6.1 en Tabel 6.1) kunnen we het volgende afleiden:

1. De *dumb* implementatie (de blauwe lijn) presteert veel slechter dan de andere twee implementaties. Het gebruik van aggregaten heeft dus een reëel voordeel, zelfs zonder afhankelijkheidsanalyse.

2. De implementatie met afhankelijkheidsanalyse (rode lijn) presteert veel beter dan de *iter* (groene lijn) implementatie.

6.2 Strongly Connected Components

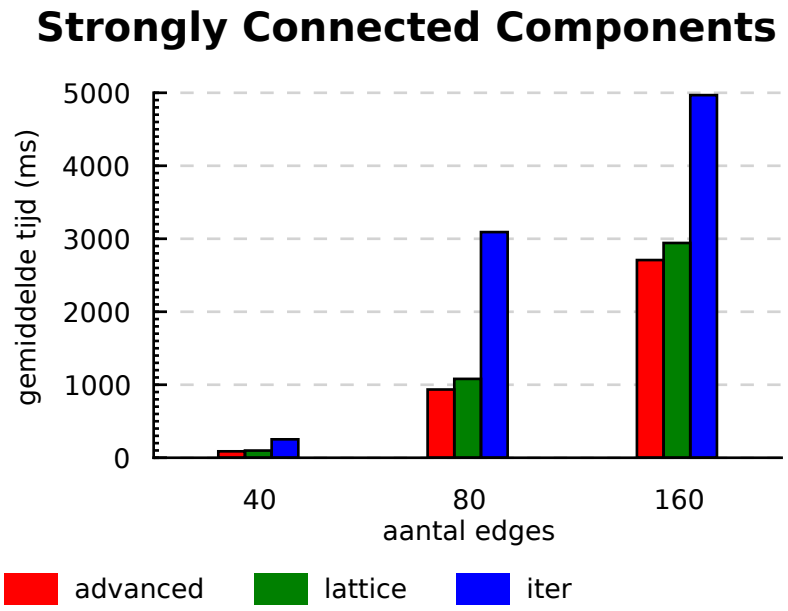
In deze benchmark is het de bedoeling om alle *Strongly Connected Components(SCC)* in een gerichte grafe te vinden.

$G = (V, E)$ is een grafe met V de verzameling vertices en $E \subseteq V \times V$ de verzameling van edges tussen deze knopen. Een vertex w zit in de Strongly Connected Component van een vertex v als er een pad bestaat van v naar w en van w naar v.

In het bijzonder gaat het over een grafen met slechts 20 knopen. De benchmarks worden uitgevoerd voor een grafe met 40, 80 en 160 edges.

Aan deze benchmark nemen 3 implementaties deel:

- **iter** Een implementatie die gebaseerd is op de semantiek $\mathcal{T}[\![\,.\,]\!]$ en het vaste punt berekent zonder gebruik te maken van afhankelijkheidsanalyse.



Figuur 6.2: Het verloop van de Strongly Connected Components benchmark.

Capaciteit	Benchmark	Tijd	
		gemiddelde	standaard afwijking
10	iter	6.71 ms	354 μs
	advanced	1.40 ms	82.3 μs
	dumb	16.9 ms	654 μs
11	iter	10.8 ms	1.20 ms
	advanced	1.66 ms	263 μs
	dumb	32.2 ms	3.01 ms
12	iter	14.0 ms	1.29 ms
	advanced	1.83 ms	181 μs
	dumb	56.3 ms	8.93 ms
20	iter	48.3 ms	3.82 ms
	advanced	4.79 ms	799 μs
	dumb	1.65 s	3.75 ms
25	iter	70.7 ms	3.93 ms
	advanced	5.67 ms	219 μs
	dumb	8.26 s	20.7 ms
30	iter	110 ms	354 μs
	advanced	7.33 ms	47.2 μs
40	iter	203 ms	807 μs
	advanced	11.8 ms	107 μs
50	iter	346 ms	916 μs
	advanced	18.9 ms	880 μs
100	iter	1.92 s	2.74 ms
	advanced	86.1 ms	474 μs
200	iter	11.9 s	57.8 ms
	advanced	324 ms	24.7 ms
400	advanced	715 ms	1.41 ms

Tabel 6.1: De resultaten van de *knapsack* benchmark.

$$sccIter :: Graph \rightarrow Vertex \rightarrow S.Set\ Vertex$$
$$sccIter\ g\ x = iterTableSemantics\ \$\ tpToTable\ \$\ scc\ g\ x$$

waarbij *iterTableSemantics* de effect handler is. De definitie van *scc* is te vinden in het online repository.

- **advanced** Een implementatie gebaseerd op de semantiek $\mathcal{T}[\![\,.\,]\!]$ en het vaste punt berekent met afhankelijkheidsanalyse.

$$sccAdv :: Graph \rightarrow Vertex \rightarrow S.Set\ Vertex$$
$$sccAdv\ g\ x = ts\ \$\ tpToTable\ \$\ scc\ g\ x$$

waarbij *ts* de effect handler is.

- **lattice** Een implementatie die gebaseerd is op aggregaten (hier wordt de set-tralie gebruikt) en het vaste punt berekend met afhankelijkheidsanalyse.

$$sccAdvLattice :: Graph \rightarrow Vertex \rightarrow S.Set\ Vertex$$
$$sccAdvLattice\ g\ x = tl\ \$\ runProgram\ \$\ scc\ g\ x$$

waarbij *tl* de effect handler is.

Uit de resultaten (Figuur 6.2 en Tabel 6.2) kunnen we het volgende opmerken:

1. Het nut van de afhankelijkheids analyse is opnieuw duidelijk. Zowel *advanced* (rode balk) en *lattice* (groene balk) zijn duidelijk sneller dan de naïeve implementatie (blauwe balk).

2. Er bestaat een klein verschil tussen de geavanceerde implementatie (rode balk) en de implementatie met tralies (groene balk). De compiler genereert dus optimalere code voor de specifieke instantie voor verzamelingen dan voor de meer generieke instantie voor tralies.

Aantal edges	Benchmark	Tijd	
		gemiddelde	standaard afwijking
40	advanced	88.1 ms	1.38 ms
	lattice	98.5 ms	3.92 ms
	iter	252 ms	2.54 ms
80	advanced	935 ms	16.6 ms
	lattice	1.08 s	59.3 ms
	iter	3.09 s	1.52 ms
160	advanced	2.71 s	15.6 ms
	lattice	2.94 s	6.64 ms
	iter	4.97 s	52.7 ms

Tabel 6.2: De resultaten van de *Strongly Connected Component* benchmark, met 20 vertices.

6.3 Kortste pad

Deze benchmark berekent de kortste paden tussen alle nodes in een gewogen gerichte grafe. Een gewogen gerichte grafe is een gerichte grafe $G = (V, E)$, met daarbij een functie $w : E \times E \to \mathbb{R}_0^+ \cup \{+\infty\}$ die aan elke edge (u, v) een bepaald gewicht toewijst. Als $(u, v) \notin E$ dan $w(u, v) = +\infty$.

Het doel is dan om een functie sp te vinden zodat:

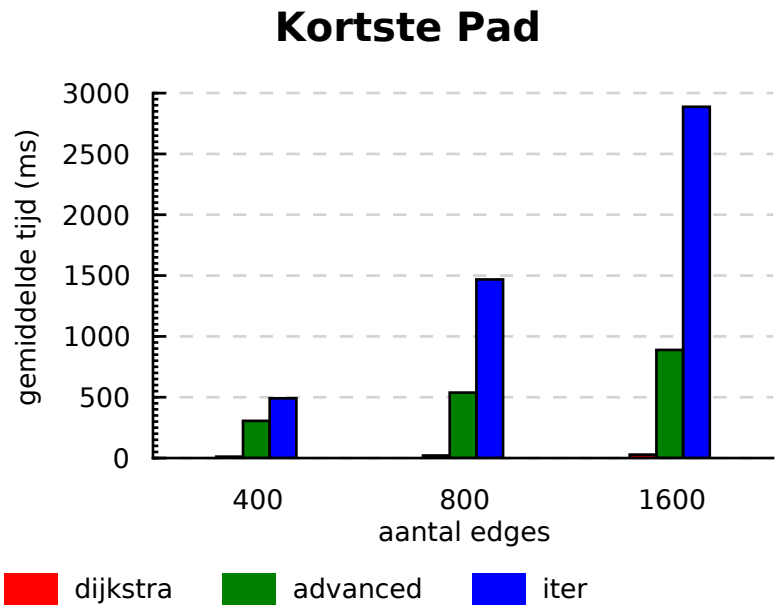
$$sp : E \times E \to \mathbb{R}_0^+ : (u, v) \mapsto sp(u, v)$$
$$sp(u, v) = \min(\{ \sum_{(x,y)\in p} w(x, y) \mid p \text{ is een pad van } u \text{ naar } v\} \cup \{+\infty\})$$

In het bijzonder gaat het om een grafe met 200 knopen. De benchmarks worden uitgevoerd op grafen met 400, 800 en 1600 edges.

Aan deze benchmark nemen drie implementaties deel:

- **dijkstra** Dit is een implementatie van Dijkstra's kortste pad algoritme. Deze implementatie is de basis waartegen de andere implementaties worden vergeleken.
- **iter** Een implementatie met aggregaten die het vaste punt berekent zonder afhankelijkheidsanalyse.

```
tabledSPIter :: Graph → Vertex → Vector Weight
tabledSPIter g a = unWAdj $ iterTl $ tpToTable $ shortestPathProg g a
```



Figuur 6.3: Het verloop van de Kortste Pad benchmark.

waarbij *iterTl* de effect handler is. De defintie van *shortestPathProg* en *unWAdj* zijn te vinden in het online repository. De tralie *WAdj* die hier gebruikt wordt is een pure functionele map met met $O\ (log\ n)$ amortized lookup en insert. Het resultaat is een vector (een array) van gewichten.

- **advanced** Een implemenatie met aggregaten die het vaste punt berekend met afhankelijkheidsanalyse.

  $$tabledSPAdv :: Graph \rightarrow Vertex \rightarrow Vector\ Weight$$
  $$tabledSPAdv\ g\ a = unWAdj\ \$\ tl\ \$\ tpToTable\ \$\ shortestPathProg\ g\ a$$

  waarbij *tl* de effect handler is.

Uit de resultaten (Figuur 6.3 en Tabel 6.3) kunnen we de volgende besluiten trekken:

1. Het verschil in performantie tussen Dijkstra's algoritme (rode balk) en de andere implementaties is zo groot dat het resultaat nog het gemakkelijkste op de tabel kan worden afgelezen. Realistisch gezien is Tabulatie, zoals ze nu in Haskell is geïmplementeerd dus geen alternatief voor een specialistisch algoritme.

2. De progressie van de *iter* en *advanced* implementaties toont het effect van iedere optimalisatie. Het *iter* algoritme houdt tussenliggende resultaten bij en itereert hierover tot een vast punt is gevonden. Het *advanced* algoritme itereert

Capaciteit	Benchmark	Tijd	
		gemiddelde	standaard afwijking
400	dijkstra	11.5 ms	1.38 ms
	iter	491 ms	24.7 ms
	advanced	306 ms	1.24 ms
800	dijkstra	20.8 ms	782 μs
	iter	1.47 s	65.4 ms
	advanced	538 ms	13.1 ms
1600	dijkstra	28.3 ms	414 μs
	iter	2.89 s	302 ms
	advanced	889 ms	8.91 ms

Tabel 6.3: De resultaten van de *Kortste pad* benchmark. Met 200 vertices.

eveneens, maar doet hierbij aan afhankelijkheidsanalyse, wat zorgt voor een kortere uitvoeringstijd.

6.4 Conclusie

Enerzijds is het voordeel van de afhankelijkheidsanalyse en aggregaten duidelijk te zien in alle benchmarks. In dit opzicht zijn de benchmarks dus een success. Anderzijds is de kortste pad benchmark ontnuchterend. De performantie van de Tabulatie Monad ligt nog ver onder elk realistisch niveau.

Hoofdstuk 7

Verder Werk en Besluit

7.1 Besluit

Deze thesis bestudeert de Tabulatie Monad in Haskell. Dit is een manier om programma's waarin niet-determinisme en recursie samen voorkomen toch een nuttig resultaat te laten berekenen. Deze methode is geïnspireerd op de Tabulatie-techniek uit logisch programmeren en Prolog.

De Tabulatie Monad werd opgebouwd met behulp van de *Effect Handlers*-aanpak. Bij deze aanpak wordt eerst een abstracte syntax opgesteld. Vervolgens worden concrete effecten aan deze abstracte syntax toegewezen door effect handlers. De abstracte syntax is in Haskell gespecificeerd als een algebraïsch datatype. De programmeur kan een gebruiksvriendelijke interface gebruiken, in de vorm van de *Nondet*-typeclass en open recursie.

In deze thesis hebben effect handlers twee verschijningsgestalten. Eerst worden ze formeel wiskundig gedefiniëerd. In deze vorm zijn ze een abstract model voor tabulatie. Na een vertaling in Haskell worden ze gebruikt om effectief getablede programma's uit te voeren. De effect handler is dus ook een uitvoerbare specificatie van de Tabulatie Monad.

De abstracte syntax en een naiëve effect handler samen vormen een Tabulatie Monad. De effect handler breiden we op twee manieren uit: met een afhankelijkheidsanalyse en met aggregaten. Deze twee uitbreidingen zijn orthogonaal. Een effect handler kan dus beide ondersteunen (maar hoeft dit niet te doen). Deze twee pistes worden uitgebreid geanalyseerd, zowel theoretisch als empirisch. Afhankelijkheidsanalyse verbetert de performantie van de effect handler sterk. De bron van de performantiewinst is een intelligentere berekening van het vaste punt van een recursieve oproep. Afhankelijkheidsanalyse herberekent namelijk alleen het resultaat wanneer dit noodzakelijk is omdat andere getablede functies veranderen.

Een aggregaat berekent een optimale oplossing uit een verzameling oplossingen. We ondersteunen dit door *enkel* deze optimale oplossing te berekenen. Omdat niet alle oplossingen worden berekend vermijdt deze effect handler onnodig werk. Bovendien kunnen met aggregaten bepaalde problemen (zoals het kortste pad) compacter worden geformuleerd. De besproken effect handler ondersteunt niet alle mogelijke aggregaten.

Enkel die aggregaten die een volledige tralie vormen kunnen worden gemodelleerd. Het berekenen van het aantal oplossingen of de som van de oplossing is bijvoorbeeld niet mogelijk.

We vergelijken deze uitbreidingen met de naïve effect handler op drie benchmark problemen: het knapzak probleem, het berekenen van Strongly Connected Components en het berekenen van het kortste pad in een grafe.

De eindconclusie luidt dus als volgt: we hebben op een begrijpelijke manier een Tabulatie Monad geconstrueerd. Daarbij hebben we aandacht gehad voor de performantie en de gebruiksvriendelijkheid.

7.2 Verder werk

Performantie Er is nog veel verbetering mogelijk aan de implementatie van onze effect handlers. Enerzijds moet een parallelle implementatie zeker onderzocht worden. Yap Prolog [17] ondersteunt ook parallelle berekening van Tabled Prolog programma's. Anderzijds moet het mogelijk zijn om bepaalde constante factoren sterk te verkleinen door beter datastructuren te gebruiken, bijvoorbeeld door gebruik te maken van generische programmeertechnieken. [7] Deze technieken kan automatisch een datastructuur worden afgeleid die optimaal is voor een gegeven type.

Probabilistische berekeningen Probabilistische berekeningen wijzen een bepaalde kans toe aan elke niet-deterministische keuze. Het eindresultaat van een dergelijke berekening is dan een kansverdeling over de oplossingsverzameling. Hier is eveneens een verband met probabilistische Prolog (ProbLog) [13]. Om dit soort berekeningen te ondersteunen is een aanpassing van zowel de syntax als de effect handler vereist. Nieuwe syntax is vereist om kansen te kunnen specificeren. Vanzelfsprekend moet dan ook de effect handler worden aangepast.

Flexibelere programmastructuur Tenslotte kunnen we opmerken dat onze structuur voor getablede programma's op twee manieren restrictief is. Ten eerste moeten alle getablede functies in een programma hetzelfde argument - en resultaatstype hebben. Ten tweede moeten alle functies gedeclareerd zijn alvorens het programma wordt uitgevoerd. Er is geen mogelijkheid om functies *dynamisch* te definiëren. In toekomstig werk kan bekeken worden in hoeverre en op welke manier deze beperkingen kunnen worden opgeheven.

Bijlagen

Bijlage A

Bewijzen van monotoniciteit voor denotationele semantiek

– Beware! Here be proofs.

A.1 Monotoniciteit

$\lambda s.\lambda(q,k).\mathcal{C}[\![\, q(k) \,]\!](s)$ *is monotoon.* Eerst bewijzen we dat $\lambda s.\mathcal{C}[\![\, t \,]\!](s)$ monotoon is voor alle t, door middel van inductie op de hoogte van de expressieboom t:

basis Als de hoogte van t één is dan is t ofwel een *Fail*, *Pure* x of *Call p b cont*.

1. Als $t = Fail$, dan is $\mathcal{C}[\![\, t \,]\!](s) = \phi$ ongeacht welke s, dus

$$
\begin{aligned}
&\lambda s.\mathcal{C}[\![\, t \,]\!](s) \text{ is monotoon} \\
&\iff \text{[Definitie monotoniciteit]} \\
&s_i \sqsubseteq s_j \Rightarrow \mathcal{C}[\![\, t \,]\!](s_i) \subseteq \mathcal{C}[\![\, t \,]\!](s_j) \\
&\iff \\
&s_i \sqsubseteq s_j \Rightarrow \phi \subseteq \phi \\
&\iff \\
&\mathit{true}.
\end{aligned}
$$

2. Als $t = Pure\, x$, dan is $\mathcal{C}[\![\, t \,]\!](s) = \{x\}$ ongeacht welke s, dus

$$\begin{aligned}
&\lambda s.\mathcal{C}[\![\, t \,]\!](s) \text{ is monotoon} \\
&\iff \text{[Definitie monotoniciteit]} \\
&s_i \sqsubseteq s_j \Rightarrow \mathcal{C}[\![\, t \,]\!](s_i) \subseteq \mathcal{C}[\![\, t \,]\!](s_j) \\
&\iff \\
&s_i \sqsubseteq s_j \Rightarrow \{x\} \subseteq \{x\} \\
&\iff \\
&true.
\end{aligned}$$

3. Als $t = Call\, b$ voor willekeurige b, dan $\mathcal{C}[\![\, t \,]\!](s) = \mathcal{C}[\![\, Call\, p\, b\, cont \,]\!](s) = \bigcup_{c \in s(p,b)} cont(c)$ voor alle s, dus

$$\begin{aligned}
&\lambda s.\mathcal{C}[\![\, t \,]\!](s) \text{ is monotoon} \\
&\iff \text{[Definitie monotoniciteit]} \\
&s_i \sqsubseteq s_j \Rightarrow \mathcal{C}[\![\, t \,]\!](s_i) \sqsubseteq \mathcal{C}[\![\, t \,]\!](s_j) \\
&\iff \\
&s_i \sqsubseteq s_j \Rightarrow \bigcup_{c \in s_i(p,b)} \mathcal{C}[\![\, cont(c) \,]\!](s_i) \subseteq \bigcup_{c \in s_j(p,b)} \mathcal{C}[\![\, cont(c) \,]\!](s_j) \\
&\impliedby \\
&s_i \sqsubseteq s_j \Rightarrow \bigcup_{c \in s_i(p,b)} c \subseteq \bigcup_{c \in s_j(p,b)} c \\
&\iff \\
&s_i \sqsubseteq s_j \Rightarrow s_i(p,b) \subseteq s_j(p,b) \\
&\iff \text{[Definitie } \sqsubseteq\text{: } s_i \sqsubseteq s_j \Rightarrow \forall p, b : s_i(p,b) \subseteq s_j(p,b)] \\
&true.
\end{aligned}$$

inductie Als de hoogte van $t > 1$ dan is $t = Or\, a\, b$ met a en b deelbomen met een hoogte strikt kleiner dan de hoogte van t. De inductiehypothese zegt dan dat $\lambda s.\mathcal{C}[\![\, a \,]\!](s)$ en $\lambda s.\mathcal{C}[\![\, b \,]\!](s)$ monotoon zijn. Dan:

$$\begin{aligned}
&\lambda s.\mathcal{C}[\![\, t \,]\!](s) \text{ is monotoon} \\
&\iff \text{[Definitie monotoniciteit]} \\
&s_i \sqsubseteq s_j \Rightarrow \mathcal{C}[\![\, Or\ a\ b \,]\!](s_i) \subseteq \mathcal{C}[\![\, Or\ a\ b \,]\!](s_i) \\
&\iff \text{[Definitie } \mathcal{C}[\![\, . \,]\!] \text{]} \\
&s_i \sqsubseteq s_j \Rightarrow \Big(\mathcal{C}[\![\, a \,]\!](s_i) \cup \mathcal{C}[\![\, b \,]\!](s_i)\Big) \subseteq \Big(\mathcal{C}[\![\, a \,]\!](s_j) \cup \mathcal{C}[\![\, b \,]\!](s_j)\Big) \\
&\impliedby \text{[Definitie } \cup\text{]} \\
&s_i \sqsubseteq s_j \Rightarrow \Big(\mathcal{C}[\![\, a \,]\!](s_i) \subseteq \mathcal{C}[\![\, a \,]\!](s_j)\Big) \wedge \Big(\mathcal{C}[\![\, b \,]\!](s_i) \subseteq \mathcal{C}[\![\, b \,]\!](s_j)\Big) \\
&\iff \text{[Inductie Hypothese: } \lambda s.\mathcal{C}[\![\, a \,]\!](s) \text{ en } \lambda s.\mathcal{C}[\![\, b \,]\!](s) \text{ zijn monotoon]} \\
&true
\end{aligned}$$

Nu volgt:

$\lambda s.\lambda(q,k).\mathcal{C}[\![\, q(k) \,]\!](s)$ is monotoon.
$\iff$ [Definitie monotoniciteit]
$s_i \sqsubseteq s_j \Rightarrow \lambda(q,k).\mathcal{C}[\![\, q(k) \,]\!](s_i) \sqsubseteq \lambda(q,k).\mathcal{C}[\![\, q(k) \,]\!](s_j)$
$\iff$ [Equivalentie tussen universele quantificatie en λ-abstractie]
$s_i \sqsubseteq s_j \Rightarrow \forall q,k : \mathcal{C}[\![\, q(k) \,]\!](s_i) \subseteq \mathcal{C}[\![\, q(k) \,]\!](s_j)$
$\iff$ [$\forall t : \lambda s.\mathcal{C}[\![\, t \,]\!](s)$ is monotoon.]
true

□

A.2 Continuiteit

Lemma 1 (Continuiteit van $\mathcal{C}[\![\, . \,]\!]$)**.** *Voor elke eindige abstracte syntaxboom t van type Table C geldt:*

$$\forall s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq \cdots \in [\mathit{Predicate}\ B\ C{\times}B \to \mathcal{P}(C)] : \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, t \,]\!](s_i) = \mathcal{C}[\![\, t \,]\!](\bigsqcup_{i=0}^{\infty} s_i)$$

Bewijs. We bewijzen de implicatie in beide richtingen apart.

- $\bigcup_{i=0}^{\infty} \mathcal{C}[\![\, t \,]\!](s_i) \subseteq \mathcal{C}[\![\, t \,]\!](\bigsqcup_{i=0}^{\infty} s_i)$:

  $$\begin{aligned} \bigsqcup_{i=0}^{\infty} s_i \sqsubseteq \bigsqcup_{i=0}^{\infty} s_i \iff & \forall s_i : s_i \sqsubseteq \bigsqcup_{i=0}^{\infty} s_i && [\text{ definitie l.u.b. }] \\ \Longrightarrow & \forall s_i : \mathcal{C}[\![\, t \,]\!](s_i) \subseteq \mathcal{C}[\![\, t \,]\!](\bigsqcup_{i=0}^{\infty} s_i) && [\ \mathcal{C}[\![\, . \,]\!] \text{ is monotoon }] \\ \Longrightarrow & \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, t \,]\!](s_i) \subseteq \mathcal{C}[\![\, t \,]\!](\bigsqcup_{i=0}^{\infty} s_i) \end{aligned}$$

- $\bigcup_{i=0}^{\infty} \mathcal{C}[\![\, t \,]\!](s_i) \supseteq \mathcal{C}[\![\, t \,]\!](\bigsqcup_{i=0}^{\infty} s_i)$:
  Veronderstel dat voor een x behorende tot C geldt dat $x \notin \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, t \,]\!](s_i)$.
  Als $t = \mathit{Pure}\ x$ of $t = \mathit{Fail}$ dan weten we dat $x \notin \mathcal{C}[\![\, t \,]\!](\sqcup_{i=0}^{\infty} s_i)$.
  Als $t = \mathit{Or}\ l\ r$ dan volgt:

  $$x \notin \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, t \,]\!](s_i) \Rightarrow x \notin \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, a \,]\!](s_i) \text{ en } x \notin \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, b \,]\!](s_i)$$

  Dus kunnen we inductief verdergaan met $\bigcup_{i=0}^{\infty} \mathcal{C}[\![\, a \,]\!](s_i)$ en $\bigcup_{i=0}^{\infty} \mathcal{C}[\![\, b \,]\!](s_i)$.
  Omdat t eindig moet zijn, weten we dat deze inductie effectief stopt.

Als $t = \mathit{Call}\ p\ b\ \mathit{cont}$ dan geldt:

$$
\begin{aligned}
& x \notin \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, t \,]\!](s_i) \\
& \implies \text{[definitie } \mathcal{C}[\![\, . \,]\!] \text{]} \\
& x \notin \bigcup_{i=0}^{\infty} \left(\bigcup_{c \in s_i(p,b)} \mathcal{C}[\![\, \mathit{cont}(c) \,]\!](s_i) \right) \\
& \implies \text{[monotoon: } s_i \sqsubseteq \bigsqcup_{i=0}^{\infty} s_i \Rightarrow \forall u \in \mathit{Table}\ C : \mathcal{C}[\![\, u \,]\!](s_i) \sqsubseteq \mathcal{C}[\![\, u \,]\!](\bigsqcup_{j=0}^{\infty} s_j) \text{]} \\
& x \notin \bigcup_{i=0}^{\infty} \left(\bigcup_{c \in s_i(p,b)} \mathcal{C}[\![\, \mathit{cont}(c) \,]\!](\bigsqcup_{j=0}^{\infty} s_j) \right) \\
& \implies \\
& \forall i \in \mathbb{N}, \forall c \in s_i(p,b) : x \notin \mathcal{C}[\![\, \mathit{cont}(c) \,]\!](\bigsqcup_{j=0}^{\infty} s_j) \\
& \implies \text{[definitie } \sqcup \text{ voor partiële functies]} \\
& \forall c \in (\bigsqcup_{i=0}^{\infty} s_i)(p,b) : x \notin \mathcal{C}[\![\, \mathit{cont}(c) \,]\!](\bigsqcup_{i=0}^{\infty} s_i) \\
& \implies \text{[definitie } \mathcal{C}[\![\, . \,]\!] \text{]} \\
& x \notin \mathcal{C}[\![\, t \,]\!](\bigsqcup_{i=0}^{\infty} s_i)
\end{aligned}
$$

Nu volgt:

$$
\begin{aligned}
& \forall x \in C : x \notin \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, t \,]\!](s_i) \Rightarrow x \notin \mathcal{C}[\![\, t \,]\!](\bigsqcup_{i=0}^{\infty} s_i) \\
& \iff \\
& \forall x \in C : x \in \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, t \,]\!](s_i) \Leftarrow x \in \mathcal{C}[\![\, t \,]\!](\bigsqcup_{i=0}^{\infty} s_i) \\
& \iff \\
& \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, t \,]\!](s_i) \supseteq \mathcal{C}[\![\, t \,]\!](\bigsqcup_{i=0}^{\infty} s_i)
\end{aligned}
$$

□

Theorema. *continuiteit van* $\lambda s.\lambda(p,k).\mathcal{C}[\![\, p(k) \,]\!](s)$

$$
\begin{aligned}
& \forall s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq \cdots \in [\mathit{Predicate}\ B\ C \times B \to \mathcal{P}(C)] : \\
& \quad \bigsqcup_{i=0}^{\infty} \lambda(p,k).\mathcal{C}[\![\, p(k) \,]\!](s_i) = \lambda(p,k).\mathcal{C}[\![\, p(k) \,]\!](\bigsqcup_{i=0}^{\infty} s_i)
\end{aligned}
$$

Bewijs.

$$\bigsqcup_{i=0}^{\infty} \lambda(p,k).\mathcal{C}[\![\, p(k) \,]\!](s_i) = \lambda(p,k).\mathcal{C}[\![\, p(k) \,]\!](\bigsqcup_{i=0}^{\infty} s_i)$$

$\iff$ [equivalentie tussen lambda-abstractie en universele quantificatie]

$$\forall p,k : \left(\bigsqcup_{i=0}^{\infty} \lambda(p,k).\mathcal{C}[\![\, p(k) \,]\!](s_i)\right)(p,k) = \mathcal{C}[\![\, p(k) \,]\!](\bigsqcup_{i=0}^{\infty} s_i)$$

$\iff$ [definitie $\sqcup$ voor partiële functies]

$$\forall p,k : \bigcup_{i=0}^{\infty} \mathcal{C}[\![\, p(k) \,]\!](s_i) = \mathcal{C}[\![\, p(k) \,]\!](\bigsqcup_{i=0}^{\infty} s_i)$$

$\iff$ [Lemma 1]

true

□

Bijlage B

Bewijs Theorema 3

Lemma 2. $\forall A, B \in \mathcal{P}(\mathcal{A}) : \sqcup(A \cup B) = (\sqcup A) \sqcup (\sqcup B)$

Bewijs.

$$\begin{aligned}
&\forall z \in \mathcal{A} : \sqcup(A \cup B) \sqsubseteq z \\
&\qquad \iff [\text{ def. } \sqcup] \\
&\forall x \in A \cup B : x \sqsubseteq z \\
&\qquad \iff [\text{ def. } \cup] \\
&\forall x \in A : x \sqsubseteq z \wedge \forall x \in B : x \sqsubseteq z \\
&\qquad \iff [\text{ def. } \sqcup] \\
&\sqcup A \sqsubseteq z \wedge \sqcup B \sqsubseteq z \\
&\qquad \iff [\text{ def. } \sqcup] \\
&(\sqcup A) \sqcup (\sqcup B) \sqsubseteq z
\end{aligned}$$

Dan volgt dankzij de reflexiviteit van $\sqsubseteq$:

$$\sqcup(A \cup B) \sqsubseteq \sqcup(A \cup B) \Rightarrow \sqcup(A \cup B) \sqsubseteq (\sqcup A) \sqcup (\sqcup B) \tag{B.1}$$
$$(\sqcup A) \sqcup (\sqcup B) \sqsubseteq (\sqcup A) \sqcup (\sqcup B) \Rightarrow (\sqcup A) \sqcup (\sqcup B) \sqsubseteq \sqcup(A \cup B). \tag{B.2}$$

Tenslotte volgt door de anti-symmetrie van $\sqsubseteq$:

$$(\text{B.1}) \wedge (\text{B.2}) \Rightarrow \sqcup(A \cup B) = (\sqcup A) \sqcup (\sqcup B).$$

□

Theorema. *$h(A) = \sqcup\{\uparrow c \mid c \in A\}$ is een monoid-homomorfisme van de monoid $(\mathcal{P}(A), \phi, \cup)$ naar de monoid $(\mathcal{A}, \bot, \sqcup)$.*

Bewijs. Opdat $h : \mathcal{P}(A) \to \mathcal{A}$ een monoid-homomorfisme zou zijn, moet gelden dat $h(\phi) = \bot$ en $h(A \cup B) = h(A) \sqcup h(B)$.

- $h(A)$ bewaart het lege element:

$$h(\phi) = \sqcup\{\uparrow c \mid c \in \phi\} = \sqcup \phi = \bot$$

- $h(A)$ bewaart samenvoeging:

$$\begin{aligned} h(A \cup b) &= \sqcup\{\uparrow c \mid c \in A \cup B\} \\ &= \sqcup(\{\uparrow c \mid c \in A\} \cup \{\uparrow c \mid c \in B\}) \\ &= (\sqcup\{\uparrow c \mid c \in A\}) \sqcup (\sqcup\{\uparrow c \mid c \in B\}) \text{ [Lemma 2]} \\ &= h(A) \sqcup h(B) \end{aligned}$$

□

Bijlage C

Wetenschappelijke Paper

The Table Monad in Haskell

Alexander Vandenbroucke
alexander.vandenbroucke@student.kuleuven.be

Abstract

Non-deterministic functions are functions that can return multiple answers. When such a function calls itself recursively it can produce an infinite number of answers where only finitely many distinct results exist. Tabling is a technique that allows the combination of recursion and non-determinism in Prolog.

In this paper we construct the Table Monad, introducing tabling for non-deterministic functions in Haskell. This monad is constructed using the "Effect-Handlers"-approach. A variation on tabling, tabling with aggregates is also discussed. Benchmarks show that for some cases this kind of tabling is more efficient.

1. Introduction

How long is the list of solutions computed by the following Haskell program?

$$p :: [(Int, Int)]$$
$$p = (1,2) : [(y,x) \mid (x,y) \leftarrow p]$$
$$\ggg \mathit{print}\ p$$
$$[(1,2),(2,1),(1,2),(2,1),\ldots]$$

How many solutions does the following Tabled Prolog program compute?

```
:- table p/2.
p(1,2).
p(Y,X) :- p(X,Y).

?- p(A,B).
A=1, B=2;
A=2, B=1;
no.
```

The Haskell program runs forever, producing an infinite list of answers, although only two distinct answers exist ((1,2) and (2,1)). In contrast, the Prolog program produces two distinct answers only once. The `table` directive is responsible for this behaviour. It instructs Prolog to *table* the predicate `p/2`.

Tabling [9] is a technique that originated in Logic Programming. It allows programs like the one above to terminate or be computed more efficiently.

Can we informally explain why the Haskell program does not terminate? Intuitively the behaviour of the program is caused by the co-occurrence of two concepts: recursion and non-determinism. Clearly, a program that does not contain recursion cannot run forever. Conversely, a program that can run forever must contain recursion. In this instance, it is the function p that recursively calls itself.

By non-determinism [7] we mean that a function that returns zero, one or more results (as opposed to just one): the function p does not return a single result, but returns a list of all its results (which are infinitely many). The first answer that p produces is $(1,2)$. Then p can recursively derive a new answer $(2,1)$. From $(2,1)$ it derives $(1,2)$ anew, and so on. If we allow unbridled recursion in conjunction with non-determinism, infinite recursive loops can be our reward.

Prolog has inherent non-determinism: resolution of a Prolog goal (such as `?-p(A,B)` in the example) can return multiple answers. Had the Prolog example program not been augmented with the `table` directive, it would have computed the infinite sequence `A=1,B=2;A=2,B=1;A=1;B=2;...` It would suffer from exactly the same problem as the Haskell program.

If tabling solves these issues for Prolog, then it should solve them for non-determinism in Haskell as well.
The main contributions of this paper are:

- An interface for writing recursive non-deterministic programs. It consists of the *Nondet*-typeclass and open recursion. This interface allows enough freedom to implement tabling. In particular, it allows the implementation to prevent infinite recursion (Section 3).

- An implementation of the *Table Monad* that instantiates the *Nondet*-typeclass and deals with open recursion. This implementation is based on

the "Effect-Handlers"-approach [10], which factors the discovery of a monad in two tasks:

1. The abstract syntax, describing all operations that must be supported (Section 4).
2. The effect handler that maps an abstract syntax tree to an effect, where Tabling is such an effect (Section 5).

- An effect handler that implements a slightly different form of tabling based on aggregates. This allows a much more efficient computation in certain cases (Section 5.3)
- Benchmarks that demonstrate the benefits of aggregates and an effect handler that uses dependency analysis (Section 6).

2. Overview

To prevent non-deterministic recursion from going haywire, we look to Prolog for inspiration. Emulating Prolog was what sparked interest in non-determinism in the first place [2]. By analogy, emulating Tabled Prolog with Tabled Non-determinism should be equally viable.

Non-determinism, as provided by lists, admits a Monad instance:

instance *Monad* [ ] **where**
  $return\ a = [a]$
  $[\,] \gg\!= f = [\,]$
  $(x : xs) \gg\!= f = (f\ x) +\!\!+ (xs \gg\!= f)$

This is the standard *Monad* instance for lists [7]. We want to model tabling with a monad as well. This gives us composable effects. When we have multiple tabled computations, they can be easily composed with monadic bind ($\gg\!=$) and we can layer multiple effects using monad transformers.

Discovering a new monadic structure that also implements tabling as well as providing non-determinism and recursion is a difficult task that requires a lot of creativity and ingenuity. Instead we will use the "Effect Handlers"-approach to create the monad, and "open recursion" to rein in recursion.

Effect Handlers The idea is to decompose the creation of the monad in two phases.

First, an abstract syntax is created. This syntax is in essence an embedded domain specific language (DSL) that allows us to describe all the desired operations. In particular, it should support non-determinism and recursion explicitly. The abstract syntax tree (AST) for such a language always admits an easily defined monad instance.

Second, we define an effect handler (or several handlers) that maps the AST to the desired sequence of effects. In doing so, we have effectively separated the act of coming up with the monad into interface (abstract syntax supporting the operations and monad instance) and implementation (the effect handler).

This process can be done abstractly first: we can invent a grammar that precisely specifies the abstract syntax of the DSL, and we can define a semantic function that gives this syntax a precise meaning. Only then must we deal with the gritty details of implementing this abstract syntax and semantics in Haskell (reasoning about abstract statements where such clutter is absent is often easier).

In the abstract, we are chiefly concerned with correctness: has the desired effect been achieved? This correctness property then carries over into Haskell, where we concern ourselves operational issues such as efficiency. The process is cyclic though: performance concerns that are raised in Haskell are met with formally verified measures, before they are implemented (unless they are trivial).

Programming languages have a concrete syntax and an abstract syntax. Usually the concrete syntax is used to write programs, and abstract syntax is used to interpret or compile the program. Both the concrete and abstract syntax can be written in Haskell. The DSL then becomes an Embedded DSL: written and interpreted entirely within Haskell, as opposed to writing the DSL program in a separate source language.

Different effects can be assigned to the same syntax tree. Consider a semantic function that associates a program with the set of all its distinct solutions. This semantics gives the same solution as Tabled Prolog (modulo unification, negation as failure).

Instead of always putting all solutions in the set, we can be more selective. For instance, we can retain only the optimal solution with respect to some ordering, allowing more efficient implementations of aggregates such as minimum or maximum.

This idea is formalized by generalizing the semantics for sets to any complete lattice. Intuitively, a complete lattice is a partially ordered set where for any subset we can find a smallest element of the set that is greater or equal than all the members of the subset. In particular, sets form a complete lattice under union, and any totally ordered set (for example the integers) can be extended to form a lattice under maximum or minimum.

Open Recursion To gain control over recursion, the program must be rewritten to open up recursion to external influences. The recursion in the example

$$p = (1,2) : [(y,x) \mid (x,y) \leftarrow p]$$

is closed: code outside p cannot influence the recursion. Contrast this with p_{open}:

$$p_{open} :: [(Int, Int)] \to [(Int, Int)]$$
$$p_{open}\ r = (1,2) : [(y,x) \mid (x,y) \leftarrow r]$$

The function p_{open} takes the recursive call as an explicit argument. Writing $p = fix\ p_{open}$ where *fix* is the standard Haskell fixed point operator, recovers the original p. By substituting a different fixed point operator we can decide when the recursion stops.

3. *Open* + *Nondet* Programming Interface

Programmers do not usually write in abstract syntax directly. In the same vein, a more convenient interface to open recursion and non-determinism is provided. The intent of open recursion can be more clearly stated using a type synonym:

$$\textbf{type}\ Open\ s = s \to s$$

The example function p then becomes:

$$p_{open} :: Open\ [(Int, Int)]$$
$$p_{open}\ r = (1,2) : [(y,x) \mid (x,y) \leftarrow r]$$

Non-determinism can be captured with the following typeclass:

$$\begin{aligned}&\textbf{class}\ Monad\ m \Rightarrow Nondet\ m\ \textbf{where}\\ &\quad fail :: m\ a\\ &\quad (\vee) :: m\ a \to m\ a \to m\ a\end{aligned}$$

The first function *fail* indicates a failed computation, i.e. a computation without any results. The $\vee$-operator represents non-deterministic choice between two possible non-deterministic computations (which can make choices in turn). Notice that *Nondet* is a subclass of *Monad*. As an immediate consequence, programmers may use *return a* to indicate a successful computation witnessed by some result a and $\gg\!=$ to sequence computations. With this notation, the example is:

$$\begin{aligned}&p_{nd} :: Nondet\ m \Rightarrow Open\ (m\ (Int, Int))\\ &p_{nd}\ r = return\ (1,2) \vee \textbf{do}\ (x,y) \leftarrow r\\ &\qquad\qquad\qquad\qquad\qquad return\ (y,x)\end{aligned}$$

To verify, we can instantiate *Nondet* for lists:

$$\begin{aligned}&\textbf{instance}\ Nondet\ [\,]\ \textbf{where}\\ &\quad fail = [\,]\\ &\quad (\vee) = (+\!+)\end{aligned}$$

then

$$\begin{aligned}&\ggg fix\ p_{nd} :: [(Int, Int)]\\ &[(1,2),(2,1),(1,2),(2,1),\ldots]\end{aligned}$$

exactly like before.

4. Abstract syntax

The data type *Table* defines the abstract syntax tree of a tabled computation:

$$\begin{aligned}&\textbf{data}\ Table\ p\ b\ c\ a\\ &\quad = Pure\ a\\ &\quad \mid Fail\\ &\quad \mid Or\ (Table\ p\ b\ c\ a)\ (Table\ p\ b\ c\ a)\\ &\quad \mid Call\ p\ b\ (c \to Table\ p\ b\ c\ a)\end{aligned}$$

The *Table* type has 4 arguments: p is the type of the tabled function, b is the argument to a tabled function, c is the type of its result, and a is the type of the result of the entire computation.

Each of the constructors corresponds to a specific operation:

- `Pure a` is a successful computation witnessed by a, and
- `Fail` is a failed computation (a computation without any results),
- `Or x y` is a non-deterministic choice between computations x and y,
- `Call f b cont` is a recursive call to f with argument b, and *cont* is a continuation. A continuation is a function that resumes the computation when it is given the result of the call.

Thinking of *Table* as syntax tree, the following *Monad* instance simply substitutes parts of the tree for $\gg\!=$, and wraps a value in a *Pure* leaf for *return*.

$$\begin{aligned}&\textbf{instance}\ Monad\ (Table\ f\ b\ c)\ \textbf{where}\\ &\quad return = Pure\\ &\quad (Pure\ a) \quad \gg\!= f = f\ a\\ &\quad Fail \quad \gg\!= f = Fail\\ &\quad a\ `Or`\ b \quad \gg\!= f = (a \gg\!= f)\ `Or`\ (b \gg\!= f)\\ &\quad Call\ p\ b\ cont \gg\!= f = Call\ p\ b\ (\lambda c \to cont\ c \gg\!= f)\end{aligned}$$

The *Nondet* instance simply uses the corresponding *Table* constructors:

$$\begin{aligned}&\textbf{instance}\ Nondet\ (Table\ f\ b\ c)\ \textbf{where}\\ &\quad fail = Fail\\ &\quad (\vee) = Or\end{aligned}$$

The expression $fix\ p_{nd} :: Table\ p\ b\ c\ (Int, Int)$ creates an infinite AST (see Figure 1).

Such an infinite AST is not suitable as input to an effect handler. The infinity is caused by the *fix*-operator, the this operator substitutes a new infinite tree for the right child of any *Or* node.

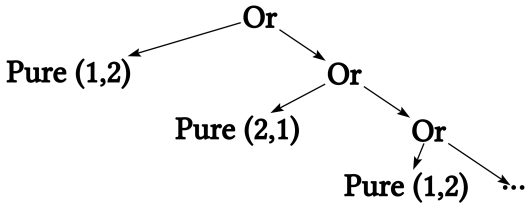


Figure 1. The infinite tree *fix* p_{nd} :: *Table p b c* (Int, Int)**.**

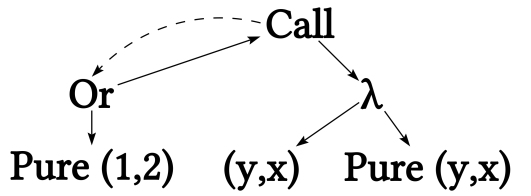


Figure 2. The finite tree *close* p_{close} ()**.**

A smarter fixed point operator *close* can produce a finite AST for open recursive functions:

$$\begin{aligned}
&\textbf{newtype } P\ b\ c = P\ \{unP :: (b \to Table\ (P\ b\ c)\ b\ c\ c)\}\\
&close\ ::\ Open\ (b \to Table\ (P\ b\ c)\ b\ c\ c)\\
&\qquad \to b \to Table\ (P\ b\ c)\ b\ c\ c\\
&close\ o = \textbf{let } t\ b = Call\ (P\ (o\ t))\ b\ Pure\ \textbf{in } t
\end{aligned}$$

Instead of simply substituting the function itself, a *Call* node with a predicate P is substituted. This closes of the recursion, making the tree finite. However, $p_{nd} :: Nondet\ m \Rightarrow Open\ (m\ (Int, Int))$ does not quite match the required type signature $Open\ (b \to Table\ (P\ b\ c)\ b\ c\ c)$. A type $Open\ (() \to s)$ is isomorphic to *Open s*, so p_{nd} is equivalent p_{close}:

$$\begin{aligned}
&p_{close} :: Nondet\ m \Rightarrow Open\ (() \to m\ (Int, Int))\\
&p_{close}\ c\ () = return\ (1,2) \vee \textbf{do } (y,x) \leftarrow c\ ()\\
&\qquad\qquad\qquad\qquad return\ (x,y)
\end{aligned}$$

The expression *close* p_{close} () creates a finite tree, displayed in Figure 2. The dashed arrow is used to denote that Call contains reference to the *Or*-node in its predicate *P b c*.

5. Effect Handler

Given an abstract syntax tree, a compiler generates code corresponding to this syntax. The code that the compiler generates can be thought of as the *meaning or semantics* of the abstract syntax. An effect handler associates a formal semantics with an AST in a similar fashion, but the meaning is represented more abstractly to simplify reasoning about the semantics.

For convenience, denote *Table* $(P\ b\ c)\ b\ c\ a$ by *Table A*, and use $p(b)$ to denote *unP p b*. The effect handler is a function $\mathbb{T}[\![\,.\,]\!] : Table\ A \to \mathscr{P}(A)$, mapping syntax to the set of all its solutions (a subset of A). Owing to the recursive nature of the AST, $\mathbb{T}[\![\,.\,]\!]$ is defined inductively.

base cases For *Pure a* and *Fail* their set of solutions is the singleton set $\{a\}$ and the empty set $\emptyset$ respectively:

$$\begin{aligned}
\mathbb{T}[\![Pure\ a]\!] &= \{a\}\\
\mathbb{T}[\![Fail]\!] &= \emptyset
\end{aligned}$$

inductive cases The solution *Or a b* of is the union of the solutions of its two choices:

$$\mathbb{T}[\![Or\ a\ b]\!] = \mathbb{T}[\![a]\!] \cup \mathbb{T}[\![b]\!]$$

How do we compute the set of solutions for *Call p b cont*? Assume that we have a function s that knows all the solutions for all predicates and arguments arguments that are used in $p(b)$, then we could also compute the results for $p(b)$ with the function $\mathbb{C}[\![\,.\,]\!]$. The function $\mathbb{C}[\![\,.\,]\!]$ is like $\mathbb{T}[\![\,.\,]\!]$ except that it computes the result of *Call q k cont* by applying *cont* on every solution in $s(q,k)$ and taking the union of the results.

$$\begin{aligned}
&\mathbb{C}[\![\,.\,]\!] : Table\ C \to (P\ B\ C \times B \to \mathscr{P}(C)) \to \mathscr{P}(C)\\
&\mathbb{C}[\![Pure\ c]\!](s) = \{c\}\\
&\mathbb{C}[\![Fail]\!](s) = \emptyset\\
&\mathbb{C}[\![Or\ a\ b]\!](s) = \mathbb{C}[\![a]\!](s) \cup \mathbb{C}[\![b]\!](s)\\
&\mathbb{C}[\![Call\ q\ k\ cont]\!](s) = \bigcup_{c \in s(q,k)} \mathbb{C}[\![cont(c)]\!](s)
\end{aligned}$$

If s contains not just the results for all predicates used in $p(b)$, but *all* predicates, then the result that $\mathbb{C}[\![\,.\,]\!]$ computes must be known to s, i.e. we have $\mathbb{C}[\![p(b)]\!](s) = s(p,b)$ for all predicates p and arguments b. Then s must be a fixed point of the function $\lambda s.\lambda(q,k).\mathbb{C}[\![q(k)]\!](s)$. In fact, we can prove[1] that this function has a least fixed point.

$$\begin{aligned}
&\text{Let } table : P\ B\ C \times B \to \mathscr{P}(C)\\
&\qquad table(p,b) = lfp(\lambda s.\lambda(q,k).\mathbb{C}[\![q(k)]\!](s))(p,b)
\end{aligned}$$

where $lfp(.)$ denotes the least fixed point operator. With the *table* function we can now define the semantics of a *Call p b cont*:

$$\mathbb{T}[\![Call\ p\ b\ cont]\!] = \bigcup_{c \in table(p,b)} \mathbb{T}[\![cont(c)]\!]$$

[1] Proof is given in my master thesis.

To demonstrate, consider that the following function is the least fixed point,

$$s(q,k) = \begin{cases} \{(1,2),(2,1)\} & \text{if q = p and } k = () \\ \emptyset & \text{otherwise} \end{cases}$$

where p is the predicate constructed.
Now we have for $\mathbb{T}[\![close\ p_{close}\ ()]\!]$:

$$\begin{aligned} \mathbb{T}[\![close\ p_{close}\ ()]\!] &= \mathbb{T}[\![Call\ p\ ()\ (\lambda(y,x) \to Pure\ (y,x))]\!] \\ &= \bigcup_{(y,x) \in table(p,())} \mathbb{T}[\![Pure(x,y)]\!] \\ &= \bigcup_{(y,x) \in s(p,())} \mathbb{T}[\![Pure(x,y)]\!] \\ &= \mathbb{T}[\![Pure(2,1)]\!] \cup \mathbb{T}[\![Pure(1,2)]\!] \\ &= \{(2,1),(1,2)\} \end{aligned}$$

this is exactly the solution found by Tabled Prolog.

5.1. Effect Handler in Haskell

Because $\mathbb{T}[\![.]\!]$ is essentially a function, and Haskell is a functional programming language, we can easily translate it into Haskell (using *Set* from `Data.Set` and *foldMap* from `Data.Foldable`).

```
𝕋 :: (Ord a, Ord c) ⇒ Table (P b c) b c a → Set a
𝕋 (Pure a)          = singleton a
𝕋  Fail             = empty
𝕋 (Or a b)          = 𝕋 a `union` 𝕋 b
𝕋 (Call p b cont)   = foldMap (𝕋 ∘ cont) (table p b)
table :: Ord c ⇒ P b c → b → Set c
table = lfp (λs → λq k → ℂ (unP q k) s)
            (λq k → empty)
ℂ :: Ord c
  ⇒ Table (P b c) b c c
  → (P b c → b → Set c)
  → Set c
ℂ (Pure a)          s = singleton a
ℂ  Fail             s = empty
ℂ (Or a b)          s = ℂ a s `union` ℂ b s
ℂ (Call p b cont)   s = foldMap (flip ℂ s ∘ cont) (s p b)
lfp :: (a → a) → a → a
```

5.2. Dependency analysis

In the previous section, the definition of the least fixed point operator *lfp* was intentionally left undefined. Computing the least fixed point is where different implementations of the same semantics have the most leeway.

The most performant solution (as shown by benchmarks, see later) performs dependency analysis between the predicates. Briefly, the idea is that when the dependencies of a predicate don't change, the predicate doesn't change either. When we iteratively compute the least fixed point, we only run $\mathbb{C}$ on those predicates that have changed dependencies. When there are no predicates with changed dependencies, we have reached a fixed point. This idea has been implemented in Haskell, and the results can be observed in the Section 6. Due to space constraints the implementation itself is elided.

5.3. Aggregates

The semantics $\mathbb{T}[\![.]\!]$ described in Section 5 associates an abstract syntax tree with the full set of all its solutions. Frequently, we are not interested in *all* solutions but only a select few, e.g. the optimum solution(s).

For some problems, where the solution space is really large (e.g. the Knapsack Problem, the Subset Sum problem), it is worthwile to compute the optimum with a different approach that avoids generating all intermediate results.

Selecting particular elements from a set is exactly the operation that an aggregate (such as `MIN` or `MAX`) accomplishes. The notion of an aggregate as "the worst thing that is is better than all elements in a subset of A" formalized with *Complete Lattices*.

For a more in-depth explantation, see Appendix A

6. Benchmarks

To substantiate the claims in Section 5.2 and Section 5.3, we compare implementations with and without dependency analysis or aggregates for a number of problems. The detailed test results are listed in Appendix B

The first problem is the knapsack problem. This problem is a natural fit for aggregates (see Appendix A). There are three implementations, displayed in Figure 3: *dumb*: dependency analysis, no aggregates; *iter*: aggregates, no dependency analysis; *advanced*: aggregates and dependency analysis. Aggregates and dependency analysis are clearly hugely beneficial.

The second benchmark computes all strongly connected components in a small directed graph (only 20 vertices, with 40, 80 and 160 edges). There are three implementations, displayed in Figure 4: *iter*: no dependency analysis, no aggregates; *advanced*: dependency analyis, no aggregates; *lattice*: dependency analysis, uses the aggregate of all solutions (this aggregate is semantically identical to $\mathbb{T}$. Dependency analysis outperforms *iter*. It seems that the compiler can

generate slightly faster code for the specific *advanced*-implementation, compared to the more general *lattice* code.

In the final problem we compute the shortest path between any two nodes in a weighted directed graph. In this case there are four implementation, displayed in Figure 5: *dijkstra*: a straight-forward implementation of Dijkstra's shortest path algorithm, without any tabling; *iter* and *advanced*: both use aggregates, only the latter uses dependency analysis. Dijkstra's algorithm significantly outperforms the other implementations, but the (beneficial) impact of dependency analysis is still clear.

7. Related Work

Pareja-Flores, and Peña and Velázquez-Iturbide [3] discuss a Tabling approach in Haskell that requires manual program transformations, as opposed to our DSL approach.

Wadler[7] introduced "lists of successes" as a way of modelling non-determinism in functional programming, which enspired the simple example used throughout this paper. A later publication [8] by the same author suggests using monads as a way of structuring functional programs. Indeed, the monad instance for lists is already mentioned in this paper. Several more advanced monad-transformers for non-determinism are discussed by Kiselyov, Shan, Friedman and Sabry [2].

A particularily accessible explanation of Effect Handlers is given by Schrijvers and Hinze[10].

Functional logic programming languages, such as Curry[1] exhibit properties of both functional and logical programming languages. Curry, for instance is heavily based on Haskell, supporting almost all Haskell constructs. In addition, it also provides syntactical constructs to return non-deterministic results, or have the system search for non-deterministic solutions for so called "free" variables. Non-deterministic functions in Curry run into the same problem with recursion that is discussed in this paper.

Warren [9] gives a good introduction to Tabled Prolog using the XSB-Prolog system. Several Prolog systems such as XSB-Prolog[6] and YapTab[5] also support aggregates.

8. Conclusion

We demonstrated that there exists a problem when we combine non-determinism with open recursion. We present a solution in the form of Tabling, inspired by Tabled Prolog. Tabling is implemented as the Table monad. This monad is created using the "Effect-Handlers"-approach, where we split this task in abstract syntax and effect handlers.

After defining the abstract syntax, we can achieve different effects with different effect handlers. In particular, we discuss two optimizations of the basic effect handler: dependency analysis and aggregates. The benefit of these optimizations have been demonstrated with a set of benchmarks.

9. Future Work

The implementation of effect handlers still has much room for improvement. The YapTab Prolog system, for instance, supports parallel exectution of Tabling [4]. More fundamental changes are also possible. Probabilistic Tabled Programs are such a possibility. In this case the result of a tabled computation is no longer a set of solutions but a probability distribution on this set.

References

[1] S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, Apr. 2010.

[2] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). In *ACM SIGPLAN Notices*, volume 40, pages 192–203. ACM, 2005.

[3] C. Pareja-Flores, R. Peña, and J. A. Velázquez-Iturbide. A tabulation transformation tactic using haskell arrays. In *Fuji International Workshop on Functional and Logic Programming. World Scientific, Singapur*, pages 211–223, 1995.

[4] R. Rocha, F. Silva, and V. Santos Costa. Yaptab: A tabling engine designed to support parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[5] J. Santos and R. Rocha. Efficient support for mode-directed tabling in the yaptab tabling system. *arXiv preprint arXiv:1301.7680*, 2013.

[6] T. Swift and D. S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In *Logics in Artificial Intelligence*, pages 300–312. Springer, 2010.

[7] P. Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

[8] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

[9] D. S. Warren. *Programming in Tabled Prolog*, volume 1. `http://www3.cs.stonybrook.edu/~warren/xsbbook/`, 1999.

[10] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In *Haskell 2014*, 2014.

A. Aggregates

We use complete lattices to capture the notion of an aggregate.

Definition 1 (Complete Lattice) *A partially ordered set $(S, \sqsubseteq)$ is a complete lattice $(S, \sqsubseteq, \cup)$ if for every $X \subseteq S$ the least upper bound (l.u.b.) $\sqcup X$ is defined, such that:*

$$\forall z \in S : \sqcup X \sqsubseteq z \iff \forall x \in X : x \sqsubseteq z$$

The l.u.b. of $\emptyset$ is denoted by the symbol $\bot$ ("bottom"), i.e

$$\sqcup \emptyset = \bot$$

In Haskell lattices are captured by the following type-class:

```
class Lattice l where
  (⊑) :: l  → l → Bool
  ⊔   :: [l] → l
  ⊥   :: l
  ⊥ = ⊔ [ ]
```

The program only concerns itself with values of type A, but the end result should be an aggregate (a lattice) of type $L(A)$. Hence we need a way to convert between these two types.

Definition 2 *Consider two functions $\uparrow : A \to L(A)$ and $\downarrow : L(A) \to \mathscr{P}(A)$:*

```
class Coerce i a where
  ↑ :: i →  a
  ↓ :: a → [i]
```

These functions must satisfy the following laws:

$$\downarrow (\uparrow s) = \{s\}$$
$$\sqcup \{\uparrow s \mid s \in \downarrow l\} = l$$

Now we can define a new semantics $\mathbb{T}_{\mathbb{L}}$[2]:

```
𝕋𝕃 :: forall l a b c ∘ (Lattice l, Coerce a l, Coerce c l)
    ⇒ Table (P b c) b c a → l
𝕋𝕃 (Pure a)        =↑ a
𝕋𝕃  Fail           = ⊥
𝕋𝕃 (Or a b)        = ⊔ [𝕋𝕃 a, 𝕋𝕃 b]
𝕋𝕃 (Call p b cont) =
  ⊔ [𝕋𝕃 (cont c) | c ←↓ (tableL p b :: l)]

tableL :: (Lattice l, Coerce c l) ⇒ P b c → b → l
tableL = lfp (λs → λq k → ⊔ [s q k, ℂ𝕃 (unP q k) s])
             (λq k → ⊥)
```

[2]The `ScopedTypeVariables` extension is required.

```
ℂ𝕃 :: (Lattice l, Coerce c l)
    ⇒ Table (P b c) b c c → (P b c → b → l) → l
ℂ𝕃 (Pure c)        s =↑ c
ℂ𝕃  Fail           s = ⊥
ℂ𝕃 (Or a b)        s = ⊔ [ℂ𝕃 a s, ℂ𝕃 b s]
ℂ𝕃 (Call p b cont) s = ⊔ [ℂ𝕃 (cont c) s | c ←↓ (s p b)]
```

Let's apply this to the knapsack problem:
An item has a *weight* and a *value*.

```
type Knapsack = [Item]
data Item = Item {weight :: Int, value :: Int}
          deriving (Show, Eq, Ord)
```

In this particular problem we only consider three items: with weight 2 and value 1, with weight 2 and value 2 and with weight 5 and value 20.

```
item :: Nondet m ⇒ m Item
item = return (Item 2 1)
     ∨ return (Item 2 2)
     ∨ return (Item 5 20)
```

The *knapsack* function is the tabled program that solves the problem: The empty knapsack ([ ]) is always a solution. Otherwise, for every item, we try to fill a knapsack with the capacity that remains after putting the item in the knapsack.

```
knapsack :: Nondet m ⇒ Open (Int → m Knapsack)
knapsack ks w | w ⩽ 0      = return [ ]
              | otherwise = return [ ] ∨ do
                i ← item
                if weight i > w then
                  fail
                else do
                  is ← ks (w − weight i)
                  return (i : is)
```

A *Knapsack* is a *Lattice*, and items can be coerced into a knapsack and vice versa.

```
instance Lattice Knapsack where
  k1 ⊑ k2 = v1 < v2 ∨ (v1 ≡ v2 ∧ w1 < w2) where
    v1 = sum (map value  k1)
    w1 = sum (map weight k1)
    v2 = sum (map value  k2)
    w2 = sum (map weight k2)
  ⊔ = foldr lmax ⊥ where
    lmax a b = if a ⊑ b then b else a
  ⊥ = [ ]
instance Coerce [Item] [Item] where
  ↑ = id
  ↓ ks = [ks]
```

The old effect handler $\mathbb{T}$ generates all solutions (and all intermediate solutions), $\mathbb{T}_\mathbb{L}$ generates only the optimal ones.

$$
\begin{aligned}
&\ggg \mathbb{T}_\mathbb{L}\ (\mathit{close\ knapsack}\ 5) :: \mathit{Knapsack}\\
&[\mathit{Item}\ 2\ 2, \mathit{Item}\ 5\ 20]\\
&\ggg \mathbb{T}\ (\mathit{close\ knapsack}\ 5)\\
&\mathit{fromList}\ [[\,], [\mathit{Item}\ 2\ 1], [\mathit{Item}\ 2\ 1, \mathit{Item}\ 2\ 1],\\
&[\mathit{Item}\ 2\ 1, \mathit{Item}\ 2\ 2], [\mathit{Item}\ 2\ 2], [\mathit{Item}\ 2\ 2, \mathit{Item}\ 2\ 1],\\
&[\mathit{Item}\ 2\ 2, \mathit{Item}\ 2\ 2], [\mathit{Item}\ 5\ 20]]
\end{aligned}
$$

‘ Every complete lattice forms a commutative idempotent monoid[3]. Some aggregates do not form such a monoid. For instance, the Sum monoid $\mathit{mappend}\ (\mathit{Sum}\ 1)\ (\mathit{Sum}\ 1) = (\mathit{Sum}\ 2)$ is clearly not idempotent. The List monoid is neither commutative nor idempotent. Hence, we cannot capture every useful aggregate with a complete lattice. Consequently, such aggregates cannot be solved with this method.

B. Benchmark Data

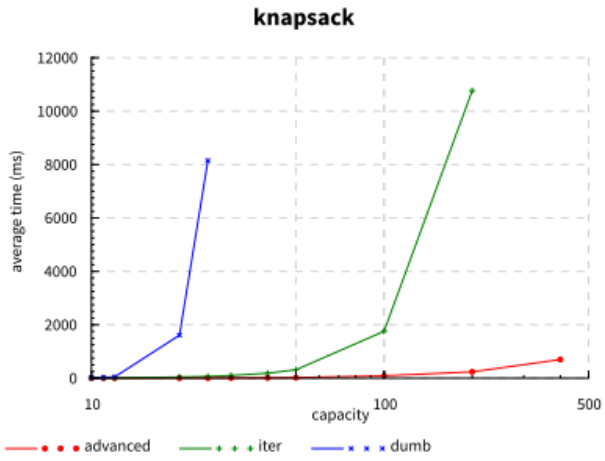


Figure 3. The results of the knapsack benchmark.

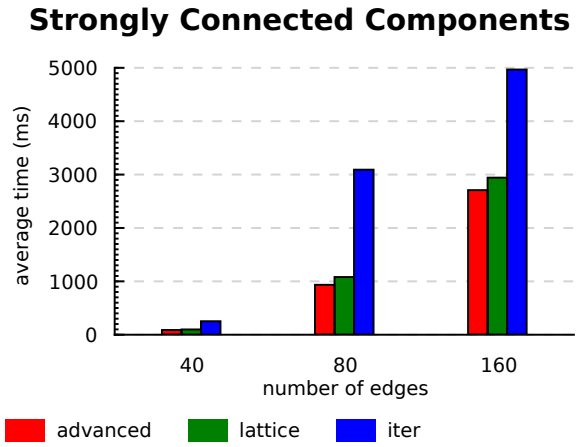


Figure 4. The results of the Strongly Connected Component benchmark.

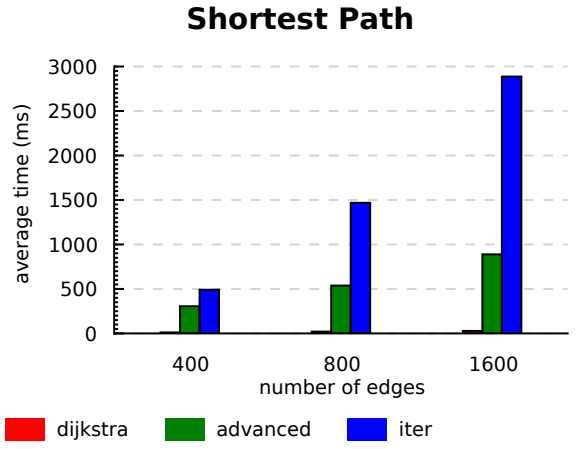


Figure 5. The results of the Shortest Path benchmark.

[3]With the following defintion: $\mathit{mempty} = \bot$, $\mathit{mappend}\ a\ b = \sqcup\,[a,b]$. Then $\sqcup\,[a,a] = a$ (idempotent) and $\sqcup\,[a,b] = \sqcup\,[b,a]$ (commutative)

Capacity	Benchmark	Time	
		avg.	std. dev.
10	iter	5.86 ms	656 μs
	advanced	1.46 ms	13.4 μs
	dumb	15.1 ms	257 μs
11	iter	9.24 ms	1.39 ms
	advanced	1.67 ms	18.2 μs
	dumb	26.6 ms	205 μs
12	iter	10.8 ms	127 μs
	advanced	1.79 ms	27.5 μs
	dumb	45.1 ms	138 μs
20	iter	39.5 ms	187 μs
	advanced	4.51 ms	33.8 μs
	dumb	1.61 s	3.80 ms
25	iter	65.1 ms	610 μs
	advanced	5.79 ms	39.8 μs
	dumb	8.15 s	5.62 ms
30	iter	100 ms	832 μs
	advanced	8.05 ms	43.6 μs
40	iter	187 ms	3.35 ms
	advanced	12.6 ms	61.2 μs
50	iter	316 ms	939 μs
	advanced	19.8 ms	242 μs
100	iter	1.76 s	2.54 ms
	advanced	89.7 ms	1.43 ms
200	iter	10.8 s	25.1 ms
	advanced	238 ms	2.99 ms
400	advanced	699 ms	15.1 ms

Table 1. The results of the knapsack benchmark.

# Edges	Benchmark	Time	
		avg.	std. dev.
40	advanced	88.1 ms	1.38 ms
	lattice	98.5 ms	3.92 ms
	iter	252 ms	2.54 ms
80	advanced	935 ms	16.6 ms
	lattice	1.08 s	59.3 ms
	iter	3.09 s	1.52 ms
160	advanced	2.71 s	15.6 ms
	lattice	2.94 s	6.64 ms
	iter	4.97 s	52.7 ms

Table 2. The results of the Strongly Connected Component benchmark.

# Edges	Benchmark	Time	
		avg.	std. dev.
400	dijkstra	11.6 ms	265 μs
	iter	505 ms	3.80 ms
	advanced	335 ms	2.68 ms
800	dijkstra	21.1 ms	275 μs
	iter	1.53 s	13.6 ms
	advanced	565 ms	3.69 ms
1600	dijkstra	29.5 ms	908 μs
	iter	2.57 s	22.2 ms
	advanced	969 ms	5.07 ms

Table 3. The results fo the Shortest-Path benchmark.

Bijlage D

Poster

The Table Monad in Haskell

Alexander Vandenbroucke

supervisor/advisor:
prof. dr. ir. Tom Schrijvers

Problem Statement

A recursive predicate that runs **forever**. Yet only a finite number of distinct solutions exists.

Prolog

Standard Prolog:

```
p(1,2).
p(X,Y) :- p(Y,X).
```

→ `?- p(A,B).`
`A = 1 B = 2;`
`B = 2 A = 1;`
`A = 1 B = 2;`
`...`

Haskell

Non-determinism (with lists):

```
p :: [(Int, Int)]
p = (1,2):[(y,x) | (x,y) <- p]
```

→ p = [(1,2),(2,1), (1,2), …]

The same predicate, now running in a Prolog interpreter with tabling, terminates in a **finite** amount of time.

Tabled Prolog:

```
:- table p/2.
p(1,2).
p(X,Y) :- p(Y,X).
```

→ `?- p(A, B).`
`A = 1 B = 2;`
`B = 2 A = 1;`
`no.`

Table Monad: **?** → p = [(1,2),(2,1)]

Method

1) Interface

Non-determinism

```
class Monad m => Nondet where
  fail :: m a
  or   :: m a -> m a -> m a
```

Open Recursion:

```
type Open s = s -> s
```

Effect Handlers

- *Creating a Monad is **hard***
- *made easier by splitting the work:*
  - *abstract syntax supporting **Nondeterminism + Open Recusion***
  - *effect handlers mapping the abstract syntax tree to **effects** (e.g Tabling)*

2) Abstract Implementation

a. abstract syntax

abstract *Table* grammar has 4 productions:

- ***Pure a***: a succesful computation with result *a*
- ***Fail***: a failed computation
- ***Or a b:*** a non-deterministic choice between *a* or *b*
- ***Call p b cont:*** a recursive call to predicate *p* with argument *b* and continuation *cont*

b. effect handler

A function *ts : Table A→ 𝒫(A)*: *t ↦ ts(t)* where ts(t) is:

- if t is ***Pure a:*** the singleton set *{a}*
- if t is ***Fail***: the empty set ∅
- if t is ***Or a b:*** the union *f(a) ∪ f(b)*
- if t is ***Call p b cont:*** the union:
  ∪{ f(cont(c))| c ∈ lfp{ λs. λ(q,k). cs(q(k),s)}(p,b)}
  where lfp{.}: computes the least fixpoint

A function *cs: Table A × (Predicate A x A →𝒫(A))→𝒫(A)*
: t, s ↦ cs(t,s)
where cs(t,s) is exactly like ts(t) except it uses *s* instead of *lfp* when resolving ***Call***

3) Haskell Implementation

a. abstract syntax

```
data Table p b c a
 = Pure a
 | Fail
 | Or (Table p b c a) (Table p b c a)
 | Call p b (c -> Table p b c a)
instance Monad (Table p b c)
instance Nondet (Table p b c)
```

Easy instances

b. effect handler

```
ts  :: Table p b c a -> Set a
lfp :: (a -> a) -> a
cs  :: Table p b c a
    -> (Predicate b c -> b -> Set a)
     -> Set a
toTable :: Open (b -> Table (P b c) b c c)
        -> b -> Table (P b c) b c c
toTable o = let t b = Call (P (o t)) b Pure
            in t
```

Results I

```
p :: Nondet m
  => Open (() -> m (Int, Int))
p r () = return (1,2) `or` do
  (y,x) <- r ()
  return (x,y)
```

① Turn *p* into a *Table*.
② Use *ts* to solve *Table*.

```
②   ①
ts (toTable p ())
= [(1,2), (2,1)]
        ③
```

③ The program **terminates**.

Conclusions

- The **Table Monad** is the Haskell analogue to Tabling in Prolog.
- The abstract interface consists of **Non-determinism** and **Open Recursion**.
- It can be implemented using the **Effect Handlers** approach.
- The implementation behaves similar to Tabled Prolog (esp. **termination**).
- **Aggregates** can be more efficient when not all solutions are needed.

Results II

```
range :: Nondet m
      => Open ((Int,Int) -> m Int)
range r (a,b)
 | b < a = fail
 | otherwise = return a `or` r (a+1,b)
```

```
②   ①          ③
tMax (toTable p (1,10)) = Max 10
```

Aggregates
① Turn *p* into a Table
② Use *tMax* to solve *Table*
③ the program is more **efficient** because we only compute the maximum (Dynamic Programming)

References

- O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). In ACM SIGPLAN Notices, volume 40, pages 192–203. ACM, 2005.
- D. S. Warren. Programming in Tabled Prolog, volume 1. http://www3.cs.stonybrook.edu/~warren/xsbbook/, 1999.
- N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In Haskell 2014, 2014.

Bibliografie

[1] *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.8.4*, 7.8.4 edition, 2015. https://downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/other-type-extensions.html#scoped-type-variables.

[2] S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, Apr. 2010.

[3] R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys (CSUR)*, 12(4):403–417, 1980.

[4] B. C.d.S. Oliveira, T. Schrijvers, and W. R. Cook. MRI: Modular Reasoning about Interference in Incremental Programming. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 22(6):797–852, 2012.

[5] N. Francez, C. Hoare, and W. P. de Roever. *Semantics of nondeterminism, concurrency and communication*. Springer, 1978.

[6] H.-F. Guo and G. Gupta. Simplifying dynamic programming via mode-directed tabling. *Software: Practice and Experience*, 38(1):75–94, 2008.

[7] F. Henglein and R. Hinze. Sorting and searching by distribution: From generic discrimination to generic tries. In *Programming Languages and Systems*, pages 315–332. Springer, 2013.

[8] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[9] P. Hudak, J. Peterson, and J. Fasel. A gentle introduction to haskell. https://www.haskell.org/tutorial/, June 2000.

[10] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pages 97–136. Springer, 1995.

[11] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). In *ACM SIGPLAN Notices*, volume 40, pages 192–203. ACM, 2005.

[12] M. Lipovača. *Learn You a Haskell for Great Good*. no starch press, 2011.

[13] T. Mantadelis and G. Janssens. Dedicated tabling for a probabilistic setting. In *Technical Communications of the 26th International Conference on Logic Programming*, volume 7, pages 124–133, 2010.

[14] E. Moggi. *Computational lambda-calculus and monads*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1988.

[15] C. Pareja-Flores, R. Peña, and J. A. Velázquez-Iturbide. A tabulation transformation tactic using haskell arrays. In *Fuji International Workshop on Functional and Logic Programming. World Scientific, Singapur*, pages 211–223, 1995.

[16] G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems*, pages 80–94. Springer, 2009.

[17] R. Rocha, F. Silva, and V. Santos Costa. Yaptab: A tabling engine designed to support parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[18] J. Santos and R. Rocha. Efficient support for mode-directed tabling in the yaptab tabling system. *arXiv preprint arXiv:1301.7680*, 2013.

[19] D. Scott and C. Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group, 1971.

[20] T. Swift and D. S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In *Logics in Artificial Intelligence*, pages 300–312. Springer, 2010.

[21] P. Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

[22] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

[23] D. S. Warren. *Programming in Tabled Prolog*, volume 1. http://www3.cs.stonybrook.edu/~warren/xsbbook/, 1999.

[24] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In *Haskell 2014*, 2014.

[25] N.-F. Zhou, Y. Kameya, and T. Sato. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *ICTAI (2)*, pages 213–218, 2010.

Fiche masterproef

Student: Alexander Vandenbroucke

Titel: De Tabulatie Monad in Haskell

Engelse titel: The Tabulation Monad in Haskell

UDC: 621.3

Korte inhoud:
Niet-determinisme geeft functies de mogelijkheid om meerdere resultaten terug te geven. In combinatie met recursie kan dit echter tot problemen leiden, waarbij de functie oneindig veel oplossing produceert terwijl er maar eindig veel verschillende oplossingen bestaan. Tabulatie is een techniek die het gelijktijdig gebruik van niet-determinisme en recursie zonder dit probleem mogelijk maakt in Prolog. In deze thesis construeren we de Tabulatie Monad, die tabulatie introduceert voor niet-deterministische functies in Haskell. Deze monad wordt opgesteld met de "Effect-Handlers"-aanpak. Naast de "gewone" tabulatie wordt ook een variant besproken die gebruikt maakt van aggregaten. Benchmarks tonen aan dat dit sommige problemen veel efficiënter kan berekenen.

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

Promotor: Prof. dr. ir. Tom Schrijvers

Assessor: Prof. dr. ir. Maurice Bruynooghe, Prof. dr. Bart Demoen

Begeleider: Prof. dr. ir. Tom Schrijvers