Een Lawine van Ontwortelde Bomen

Liber Amicorum voor Doaitse Swierstra

Eindredactie
Jurriaan Hage
Atze Dijkstra

Omslagontwerp
Atze Dijkstra
Tekeningen: Clara Löh

Publicatiedatum
30 Mei 2013

Drukwerk
Het Grafisch Huis
Kostersgang 32-34
9711 CX Groningen
`http://hetgrafischhuis.nl/`

Uitgever
Departement Informatica
Universiteit Utrecht
Postbus 80.089
3508 TB Utrecht
Nederland

Voorwoord

Sedert vele jaren kennen wij Doaitse: als student, als promovendus, en als medewerker. Lang voor zijn komst naar Utrecht had Doaitse zijn magnum opus – het proefschrift *Lawine, an experiment in language and machine design* – al geschreven. Tot lang daarna mochten wij, en anderen die in zijn omgeving vertoefden, van hem vernemen dat de "oplossing voor alle problemen" al in dit proefschrift waren terug te vinden. We hebben gemeend met de keuze van de titel van het Liber Amicorum recht te doen aan dit sneeuwbaleffect dat een enkel individu met zijn werk en gedachten op zijn omgeving kan hebben. De ontworteling die dientengevolge kan zijn ontstaan bij anderen in zijn omgeving wordt hopelijk ruimschoots goedgemaakt door de hierdoor verkregen nieuwe inzichten in leven, maatschappij en wetenschap.

De bijdragen in deze bundel laten een enorme variatie zien in onderwerp en stijl. Zo zijn er puur technische verhandelingen, persoonlijke anekdotes en gedragsanalyses, en alle denkbare mengvormen hiervan; zelfs zwaar wiskundig geschut wordt ingezet om uitspraken over Doaitse te onderbouwen. Collegae uit binnen- en buitenland, familie, en vrienden, zij hebben allen hun verhaal gedaan ter gelegenheid van het emeritaat van Doaitse, en wij willen bij deze allen bedanken voor hun bijdrage en gedane moeite. Behalve de auteurs hebben ook Clara Löh, Edith Stap, en Albert-Jan Swierstra en Anita Kroder van Het Grafisch Huis uit Groningen op essentiële wijze aan dit boekje bijgedragen, waarvoor alweer onze grote dank. Hun noeste arbeid heeft er uiteindelijk toe geleid dat voor het drukken van dit Liber Amicorum de nodige bomen ontworteld dienden te worden, maar dat terzijde.

Samengepakt in een bundel wordt de diversiteit van het geschrevene pas goed zichtbaar. We hebben geprobeerd deze rijkdom aan variatie te benadrukken door voor een presentatie van de verhalen te kiezen waarbij kort en lang, technisch en persoonlijk elkaar afwisselen. We hopen dat U, en in het bijzonder Doaitse, van het resultaat zult genieten, en dat het U inzicht zal geven in zowel de persoon Doaitse Swierstra, alsmede in het onderzoek dat hem al die jaren heeft beziggehouden.

Atze Dijkstra en Jurriaan Hage
Utrecht, 26 April 2013.

Table of Contents

Everything you always wanted to know about Doaitse **9**
Andres Löh and José Pedro Magalhães . . . . .

Men hoeft nimmer te twijfelen aan waar hij staat **12**
Carroll Morgan . . . . .

An in-depth memory of SDS **16**
Maarten Fokkinga . . . . .

A Short Example of the Calculational Method **22**
Roland Backhouse . . . . .

Subkwadratische SDS Verificatie **25**
Gerard Tel . . . . .

The history of finding palindromes **7**
Johan Jeuring . . . . .

Tempora mutantur, nos et mutamur in illis **34**
Jan van Leeuwen . . . . .

How do I Type Data? For Doaitse, Emeriti Rule **39**
Arno Siebes . . . . .

Haskell in the Large **47**
Jurriaan Hage . . . . .

Verstandige Zaken of Het Ontleden van Haskell **55**
Hans L. Bodlaender . . . . .

Incremental Evaluation, Again **59**
Matthijs Kuiper . . . . .

Traversals with Class **62**
Bastiaan Heeren . . . . .

Testing Proven Algorithms **76**
Thomas Arts . . . . .

Accumulating Attributes **87**
Jeremy Gibbons . . . . .

Metaclasses in object oriented programming languages **103**
Piet van Oostrum . . . . .

Doaitse, parsers, theorie en praktijk **117**
Harald Vogt . . . . .

Embedded Attribute Grammars with C++ Templates **118**
Arie Middelkoop . . . . .

A note on indirection **128**
Eelco Dijkstra . . . .

Derivation of an imperative unification algorithm **132**
Lex Bijlsma . . . .

The Mismatch between Computing Science and Corporate/Government IT **145**
Nico Verwer . . . .

Anecdotarium **148**
Alexey Rodriguez Yakushev . . . .

Mechanized Metatheory for a λ-Calculus with Trust Types **150**
Lucilia Figueiredo, Rodrigo Ribeiro and Carlos Camarão . . . .

Herinneringen aan SDS **163**
John-Jules Meyer . . . .

Type Checking by Domain Analysis in Ampersand **164**
Stef Joosten . . . .

$\forall$UNITY for Everyone **176**
Wishnu Prasetya and Tanja Vos . . . .

Squiggoling with Bialgebras - Recursion Schemes from Comonads Revisited **189**
Ralf Hinze and Nicolas Wu . . . .

Een Veelkleurige Universiteit **202**
Marinus Veldhorst . . . .

Doing a Doaitse: Simple Recursive Aggregates in Datalog **207**
Oege de Moor . . . .

Doaitse – Mijn Mentor in de Nieuwe Wereld **217**
Maarten Pennings . . . .

Circularity and Lambda Abstraction **219**
Olivier Danvy, Peter Thiemann and Ian Zerny . . . .

Doaitse heeft altijd gelijk **232**
Wilke Schram . . . .

Janboerenfluitjes | Zekerheidzoeken **233**
Sietse van der Meulen . . . .

Beste Doaitse **236**
Lidwien van de Wijngaert . . . .

Formation in the public debate about nanotechnology **237**
Tsjalling Swierstra en Lidwien van de Wijngaert . . . .

Frater Familias **253**
Tsjalling Swierstra . . . . .

Thanks to Doaitse I Became a Nerd **255**
Karina Olmos . . . . .

A Man with a Mission **256**
Fritz Henglein . . . . .

Challenging Doaitse by Bayesianisms **258**
Linda C. van der Gaag . . . . .

Grootschaligheid als uitdaging **264**
Jan Grijpink . . . . .

Aan Doaitse **274**
Corine de Gee . . . . .

The Era of Doaitse **275**
Jeroen Fokker . . . . .

Herenleed **278**
Remco Veltkamp . . . . .

The Workroom **279**
Atze Dijkstra . . . . .

Fun With Robots, CoCoCo and Mate **283**
Alberto Pardo and Marcos Viera . . . . .

Template-Based Document Generation Using Attribute Grammars **295**
Pablo Ramón Azero Alcocer . . . . .

Methodische plaatjes vullen geen gaatjes **301**
Matthijs Maat and Gert Florijn . . . . .

There Is Just Information at the Bottom **305**
Lex Augusteijn . . . . .

Fast, Applicative Combinator Lexers **307**
Stefan Holdermans . . . . .

AMEN **317**
Wouter Swierstra . . . . .

Everything you always wanted to know about Doaitse

Andres Löh[1] and José Pedro Magalhães[2]

[1] andres@well-typed.com
Well-Typed LLP
[2] jpm@cs.ox.ac.uk
Department of Computer Science, University of Oxford

Abstract. Doaitse is a very special person. His humor, wit, brightness, and character never fail to impress any audience, independently of context or location. He has continuously inspired and motivated us, and we will always remember our time in Utrecht with fond memories of Doaitse. We collect an (obviously non-exhaustive) list of such memories here. Whether he be in Utrecht, Tynaarlo, or anywhere else, we will always remember Doaitse's unique personality.

Notation

The use of syntax such as <$>, <*>, and <|> when programming with applicative functors, as popularised by McBride and Paterson [1], is now commonplace. In a private communication with Doaitse, however, we were able to ascertain that this notation was, in fact, invented by him. Indeed, it can already be seen in one of his papers dating back to 1996 [2]. As is common knowledge, the syntax of a programming language is a contentious and dangerous subject, easily sparking irate online discussions and flame wars. In fact, some language feature proposals end up never being implemented due to disagreement on syntax alone. As such, we will appropriately credit Doaitse for the invention of this beautiful notation by adapting the following naming convention in our future papers:

$<\$>$ *Swierstra florin* $<*>$ *Swierstra star* $<|>$ *Swierstra dike*

Circularity

Doaitse once came upon the following fragment of code, which diagonalises a series of (potentially infinite) lists into a single list:

$$
\begin{aligned}
&\mathit{diag} :: [[\alpha]] \to [\alpha] \\
&\mathit{diag} = \mathit{concat} \circ \mathit{foldr}\ \mathit{skew}\ [\,] \circ \mathit{map}\ (\mathit{map}\ (:[\,])) \\
&\mathit{skew} :: [[\alpha]] \to [[\alpha]] \to [[\alpha]] \\
&\mathit{skew}\ [\,]\ \ l = l \\
&\mathit{skew}\ (h:t)\ l = h : \mathit{combine}\ (+\!\!+)\ t\ l \\
&\mathit{combine} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \to [\alpha] \\
&\mathit{combine}\ \_\ \mathit{xs}\ \ [\,]\ \ = \mathit{xs} \\
&\mathit{combine}\ \_\ [\,]\ \ \mathit{ys}\ \ = \mathit{ys} \\
&\mathit{combine}\ f\ \ (x:\mathit{xs})\ (y:\mathit{ys}) = f\ x\ y : \mathit{combine}\ f\ \mathit{xs}\ \mathit{ys}
\end{aligned}
$$

Having found it verbose and hard to analyse in terms of computational complexity, Doaitse came up with the following code instead:

```
diag :: [[α]] → [α]
diag xs = diag' xs [] [] where
  diag' []           []  []     = []
  diag' []           ll  (r:rr) =   diag' (r:ll) []     rr
  diag' []           ll  []     =   diag' ll     []     []
  diag' ((v:vv):ww)  ll  rr     = v:diag' ww     (vv:ll) rr
  diag' ([]:ww)      ll  rr     =   diag' ww     ll     rr
```

Although it diagonalises in a different way, it is as valid as the original program, and more clearly linear on its input.

But Doaitse was not happy yet, and one day, while biking, he came up with a simple two-line solution, which he wrote on the whiteboard when he arrived at the office. Given the ephemeral nature of writings on a whiteboard, we cannot recall the complete details of that program. We leave it here with a few gaps, which Doaitse can certainly fill in quickly:

```
let xs = ...   -- hint: use xs here
in ...         -- hint: use xs here too
```

The most remarkable property of this implementation was not its size, nor its clear circularity, however; it was the fact that it only worked correctly when given an infinite input, diverging otherwise.

Languages

In the invitation to prepare this submission, we were told we could use "any language that you know Doaitse can read". Dado que o Doaitse consegue ler basicamente qualquer linguagem, decidimos escrever este parágrafo em diversas linguagens. Doaitse venait souvent dans le bureau et commençait une conversation dans la langue maternelle de l'auditeur. In tal fan Doaitse's sinnen wiene simpel mar soad oare wiene bysûnder slim. Se dice que el ha aprendido muchos idiomas solo por escuchar una vieja serie de lecciones grabadas. Среди его словарного запаса готовых фраз были такие как "Здравствуйте, дети!". Weitere Lieblingssätze waren "Jetzt geht's los, die Löcher aus dem Käse!", "Heute bin ich ein richtiger Krawallmacher!", "Das wird schmecken!" sowie generell jeder deutsche Satz, in dem Passiv verwendet wird. Hoewel een buitenstaander zulke zinnen alleen in specifieke omstandigheden zou gebruiken, lukt het Doaitse altijd om de actuele situatie aan zijn zinnen aan te passen.

Christmas

On the 5th of December 2008, a day that is traditionally celebrated in the Netherlands with an exchange of presents during the "Sinterklaasavond", Doaitse released his `ChristmasTree` package for the (Haskell) world to download and unpack.[3] This package's name is, however, a misnomer; the correct rendering would be `CHRISTMASTREE`, as it is, in fact, an acronym, standing for "Changing Haskell's Read Implementation Such That by Manipulating ASTs it Reads Expressions Efficiently".

Furthermore, it is not only brilliantly named; it also does what its name promises, reading data in linear time, and avoiding the standard Haskell *read* exponential behavior in some cases of data types with infix operators.

[3] `http://hackage.haskell.org/package/ChristmasTree`

A rather amazing program

Profiling, in GHC, adds annotations to a program, so that information can be collected at runtime to determine its space and time behaviour. This invariably makes programs slower, so profiling is only used as a debugging tool. Except Doaitse found a program which ran *faster* (nearly twice as fast) with profiling.[4] We reproduce (most of) this program below:

```
main           = parseIO uulibP inputExample >>= print
uulibP         = length <$> (pList $ pChoice $ map (pSym) ['a'..'z'])
pChoice ps     = foldr (<|>) pFail ps
inputExample   = foldr (++) "" $ replicate 1000 difficultString
difficultString = "abcdefadsjkhdasjkdasjhkdsakjdsajkdsafklfddsfajklyrrtttryytuuyttyuuy
```

(Notice Doaitse's use of his own *florin* and *dike* operators.) Simon Peyton Jones said this was "clearly a rather amazing program". Simon Marlow considered it "fascinating", and said he would have to sleep on it to see if a fix would occur to him. Some time later, a good dream (or perhaps a nightmare) told Simon Marlow that the way to fix this behaviour was by "avoiding generating chains of indirections in stack squeezing".

Postscript

Thank you, Doaitse, for being such an amazing person, and having helped us both countless times in our days in Utrecht. You have shaped our way of thinking, and made us better persons, both personally and academically.

References

1. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1) (January 2008) 1–13
2. Swierstra, S.D., Duponcheel, L.: Deterministic, error-correcting combinator parsers. In: Advanced Functional Programming, Second International School-Tutorial Text, London, UK, UK, Springer-Verlag (1996) 184–207

[4] `http://hackage.haskell.org/trac/ghc/ticket/5505`

Men hoeft nimmer te twijfelen aan waar hij staat

Carroll Morgan

carrollm@cse.unsw.edu.au
School of Computer Science and Engineering
University of New South Wales

Abstract. Op 30 mei 2013 doet Prof. S. Doaitse Swierstra, na dertig jaar, afstand van zijn functie bij de Rijksuniversiteit Utrecht[1]; op 30 april doet Hare Majesteit Koningin Beatrix, na drieëndertig jaar, afstand van haar troon bij het Koninkrijk der Nederlanden. Toevallig werden deze plannen (allebei) op 29 januari 2013 bij ons onder de aandacht gebracht; ze hebben (allebei) met "rijken" te maken; en ze gaan (allebei) over hele bijzondere mensen.

Kan dit echt op louter toeval berusten? Zijn er nog meer overeenkomsten te ontdekken? Voor zo'n uitdaging moeten we niet wijken: nieuw, uitgebreid onderzoek wordt geëist.

De eerste aanleiding tot dit nieuwe onderzoek werd in de pers gevonden [1], waar men het volgende las:

> Ik hoefde nimmer te twijfelelen
> aan waar de koningin stond.
>
> *Wim Kok*
> *Voormalige minister-president van Nederland*

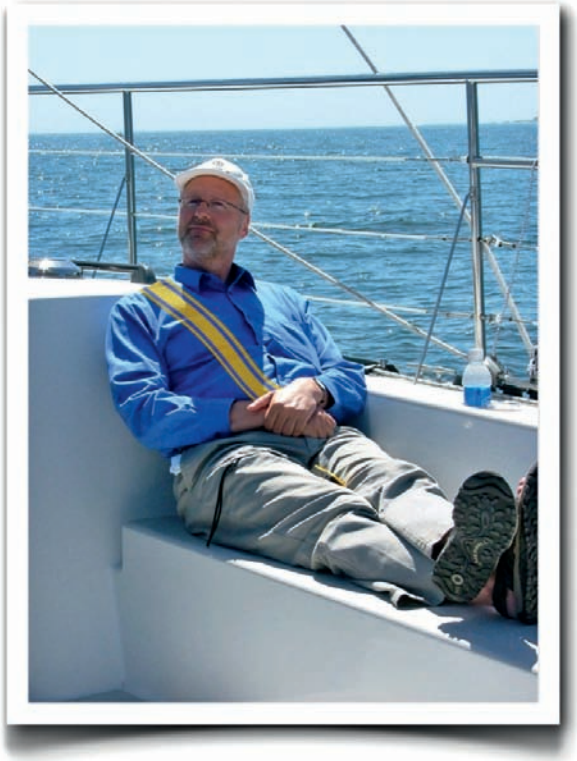
Wat opvallend is, is dat je deze eigenschap ook heel snel bij je eerste kennismaking met Doaitse Swierstra meemaakt. Maar er is ook een zekere verwarring bij, iets dat je een "Heeft-ie dat *echt* gezegd?" -gevoel zou kunnen noemen. Voor sommige van ons is dat in het gezelschap van de *IFIP* Working Group 2.1 gebeurd, bij onze allereerste bijeenkomst, wat de verwarring alleen groter maakte: dat zijn hele rare mensen, en je werd er echt duizelig van. Maar tegelijkertijd hielp het dat je bij die 2.1-mensen was: als je wat schichtig rondkeek, zag je dat zij, blijkbaar, allemaal aan Doaitse gewend waren — en dus had je er niets van te vrezen, wat hij dan ook heeft gezegd. De wereld draaide gewoon door; er kwam geen bliksem uit de hemel naar beneden te schieten.

Dus het zou makkelijk zijn de indruk te krijgen dat, als je bij Doaitse bent, er een constante spanning in de lucht hangt, een dreigend waas van onzekerheid. Niets zou verder van de waarheid kunnen zijn: als je met Doaitse in gezelschap bent is het heel rustig, ben je volledig op je gemak, hoef je eigenlijk helemaal niks meer voor jezelf te beslissen. Dat doet hij, natuurlijk.

Waar ga je slapen? *Geen probleem*: Doaitse weet het al, en het spreekt vanzelf dat je een kamer met hem mee zou moeten delen [2]. Waarheen ga je op je volgend sabbatical? *Geen probleem*: Doaitse heeft al een sabbatical bij de *RUU* geregeld, en *verder* heeft-ie een mooi, groot huis voor jou –en je familie van 6 mensen– in Utrechts Tuindorp gevonden [3]. In het kort: bij Doaitse wordt het *warme* niet-determinisme van jou tot een *koud* niet-determinisme

[1] Zo heette het eerder.

van hem omgetoverd:[2] dat wil zeggen voor jou nog wel verrassend, maar voor hem helemaal vooraf beslist. Het leven wordt dus voor iedereen wat simpeler, een verfrissend verlicht absolutisme, door Doaitse geïncarneerd, in dit modern tijdperk van saaie constitutionele monarchieën.



Doaitse houdt z'n eilandje zorgvuldig in de gaten.

En er zijn nog meer overeenkomsten tussen Doaitse en Hare Majesteit. Zo Koningin Beatrix gezag over verschillende eilanden in de Caraïbische Zee uitvoert, zo voert Doaitse gezag uit over een stukje van de Noordzee. Lekker warm is het daar niet — maar er zijn andere voordelen.

Ga eerst maar naar Texel, en daarna linksaf: dan kom je uiteindelijk op de *ABC* eilanden van Beatrix terecht. Maar sla je bij Texel naar rechts, dan kom je onmiddellijk bij één eiland *A* aan: en daar is het het domein van Doaitse.

Veel van ons zijn bij Ameland geweest, door Doaitse voor officieël staatsbezoek uitgenodigd — en daar hebben we uiterst geheime dossiërs bekeken, de meesten ervan in een cryptische taal overgedragen die speciaal voor die taak ontwikkeld werd. Ik zou zeggen dat je meer van zulke dingen zou komen te weten door één week op dat frisse *A* van Doaitse te verblijven dan bij meerdere maanden op die zwoele *ABC*-tjes van de koningin.

Wat Zuid Amerika betreft, zijn er nog verdere verbanden van alle twee Hoogheden te vinden — waar we Doaitse die ereteken zou kunnen veroorloven wegens zijn Cochabamba-Connectie die in Utrecht begint, een kleine meter boven de zeespiegel, maar in de Andes op 2.558 meters verrassend uitsteekt. Zo vindt een Hoogheid zich toch volledig op Z'n gemak... meestal. Helaas heeft Doaitse daar een bijeenkomst geörganiseerd die berucht werd vanwege wateroverlast [5]. In het geval van Hare Majesteit lopen de belangen elders, naar Argentinië toe maar wel in de buurt, en volgens sommige mensen zijn die ook een beetje berucht.

In die streken moeten Hoogheden blijkbaar altijd op Hun hoede blijven.

[2] *Warm* niet-determinisme wordt opnieuw uitgevoerd wanneer de behoefte zich voordoet — en het mag elke keer anders uitkomen; *koud* niet-determinisme wordt vooraf beslist, maar blijft tot het laatste moment bedekt — en daarna is het elke keer dezelfde. Eerder werden ze als *rood-* en *blauw* niet-determinisme bekend [4].

Koningin Beatrix werd ook als volgt beschreven:

Ze weet veel, daagt je uit
en praat je niet naar de mond.

Jan Pronk
oud-minister

Dat zou je ook over Doaitse kunnen zeggen.

Er zijn veel mensen die, uit beleefdheid, altijd met een compliment zouden beginnen — "Wat heb ik je toesprak interessant gevonden!" En dan weet je zeker wat er nog te verwachten staat...

Zo is het met Doaitse helemaal niet, omdat bij hem hoef je zelden te wachten. Zíjn echte mening hoor je meteen, eigenlijk voortijdig zou je kunnen zeggen: vaak krijg je een oordeel van Doaitse nog voordat je begonnen bent. Maar uiteindelijk, hoor je misschien ook iets leuks... uiteraard alleen als je het verdiend hebt. En als dat gebeurt, weet je dan ook dat het in alle ernst bedoeld wordt.

Het blijkt dat Koningin Beatrix van plan is naar Kasteel Drakenstyn te verhuizen. Een kijkje op het internet voldoet om te zien dat het kasteel –op de voorkant tenminste– bijna geen muren heeft en dat het door water omsingeld wordt. Het is voornamelijk uit ramen gemaakt, met een stukje lijm hier en daar geplakt om ze overeind te houden, en het staat achter een sloot. Zo is het bijna ook met het nieuwe kasteel van Doaitse: er blijken meer ruiten dan bakstenen te zijn, en een mooi meertje ligt maar een paar meter verderop.

Voor de meesten van ons is het moeilijker iets te zeggen over de persoonlijke kant van Hare Majesteit. Maar nu ze met pensioen gaat, zal ze haar vriendjeskring zeker een beetje uitbreiden: ik blijf bij de telefoon. In het geval van Doaitse is er echter meer duidelijkheid beschikbaar: *the cure for pessimism is optimism; there is no cure for optimism* [6]. Daar lijdt Doaitse zeker aan — en ik laat dit knipsel, uit een recent berichtje, even zien als bewijs ervan [7]:

I just woke up here as a retired man,
sun shining, ice on the lake,
and a lot of work to do.

Doaitse Swierstra

Leden van de Koninklijke Orde van Amici Doedenii [8] zullen deze aanpak van het leven zeker herkennen. Inderdaad, het resultaat van ons onderzoek heeft ons tot de onontkoombare conclusie geleid dat

---

Doaitse is a unique personality, that it is a privilege to count him as a friend and colleague, and that

He need never doubt where he stands

in the esteem of those who know him.

Acknowledgements

Thanks to *MM* and *RvG* for proofreading.

References

1. NRC Handelsblad. *Special Beatrix*. Jaargang 43 no.10. 29–30 January 2013.
2. Hendrik Boom. Host for WG 2.1 meeting 37 in Montreal. May 1987.
3. Doaitse Swierstra. Host for 6-month sabbatical at *RUU*. Thank you!
   Utrecht, second-half 1994.
4. Alberto Pettorossi. Host for WG 2.1 meeting 38 in Rome. March 1988.
5. Pablo Azero. Local organiser for WG 2.1 meeting 55 in Cochabamba.
   January 2001.
6. Dorothy Parker (attrib.)
   "The cure for boredom is curiosity. There is no cure for curiosity."
   `http://en.wikiquote.org/wiki/Dorothy_Parker`
7. Prof. S.D. Swierstra. Rijksuniversiteit Zandlust, Tynaarlo.
   Private communication. March 2013.
8. NRC Geboorteregister.
   `http://geboorteregister.nrc.nl/zoeken/voornaam/Doaitse/`

An In-Depth Memory of SDS

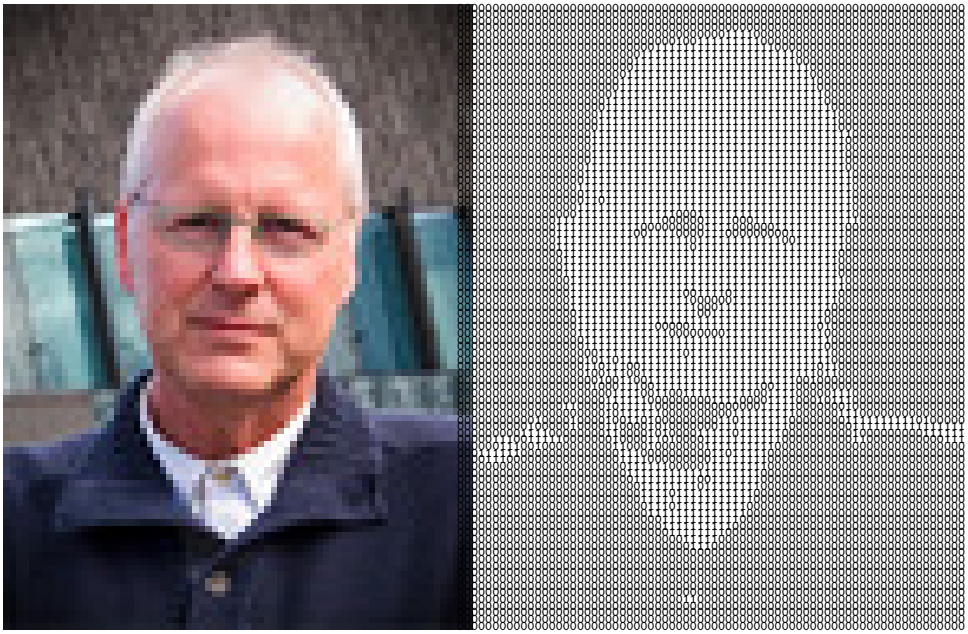
Maarten Fokkinga

m.m.fokkinga@utwente.nl
Dept. EWI, University of Twente

In 1976 I met Doaitse for the first time; he began his work at the *Technische Hogeschool Twente* as *promotiemedewerker*, where I had a job as *wetenschappelijk medewerker*. Time, names, locations, and positions have changed since then a lot, but one thing has been invariant (for which I'm very happy): we've stayed in contact and kept to have the same interests, more or less.

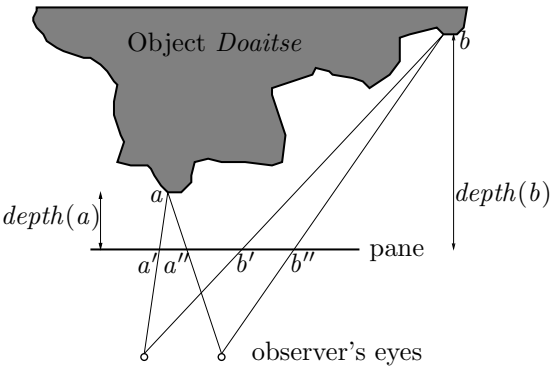
In this note I would like to recall some memories, but I realize that whatever I will write, it will not do justice to Doaitse's contributions to my academic life. Therefore, I'll present the memories figuratively rather than literally; actually, one figure only: a picture of the face view of Doaitse. Since Doaitse deserves more than the two dimensions that I have at my disposal here in this booklet, I will create a third dimension for him: depth. I'll do this with a poor man's program that I've constructed in 1995, exactly in the middle of the years 1976–2013 that we know each other.

The basis is a well-known photograph together with its translation to bits 0,1:



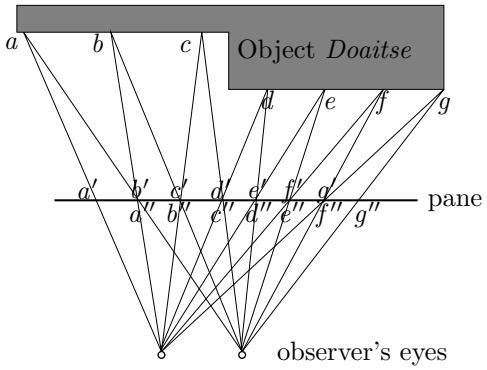
Doaitse in colors and in bits.
(By suitable scaling the right picture gets the proper height-width ratio.)

3D pictures. Before presenting the program code, we very briefly sketch the principle upon which our 3D picture construction is based. Consider this top view of the eyes of an observer, a pane of glass (on which we are going to paint a picture), and an object:



Points a', a'' are the image of a on the pane of glass, as viewed by the left and right eye of the observer. If at points a', a'' two suitable pixels are painted on the pane, then the observer doesn't experience any difference in his view of point a of the object. Similarly for b and b', b'', and any other points $c, d, e, \ldots$.

Now suppose points a and b are chosen on the object in such a way that a'' and b' coincide and therefore carry identical pixels. The observer experiences the same color (the *color of the pixel*) for both points of the object, but he does not experience a difference in the depth (= distance to the pane) of the points. In the same way, there might be a series of points $c, d, e, \ldots$ on the object such that b'' and c' coincide, as well as c'' and d', and d'', e', and so on; the pixels on the pane at all these points are equal and the observer experiences the same color (the color of the pixel) for all those points of the object, but he does not experience a difference in the depth of these points of the object:



It follows that a line on the surface of the object that is parallel to the line through the observer's eyes (and thus parallel to the pane too), is pictured on the pane as a repeating pattern of pixels. A jump in the line of the object to a lesser distance to the pane, makes the pattern shorter. Working with disjoint pixels of finite size, a shorter pattern means that some pixels of the pattern disappear.

On this principle, our algorithm is based: on the pane we construct a picture, and each line in the picture is filled with a repeating pattern, representing a line of the object's surface, parallel to the line through the observer's eyes, as a whole appearing as a kind of *wall-paper*. When there is a transition in the distance of the object's line segment to the pane, called *depth*, the pattern grows or shrinks by one (or more) pixel. We take characters `abc...` as pixels (so that what we have called the *color* of a pixel is now its *form*).

The program code. We start with a desired depth encoding:

$depth$ = the list displayed on the next page in Fig. 1, containing strings, each of which consists of digits `0`,`1`,...,`9`

So, the eventual 3D picture will show a flat background (depth is zero) with Doaitse's head in the foreground (depth is one); his eyes and mouth and some other small parts, however, are displayed as holes: they are again in the background (because the depth is zero). We choose the following pattern:

$pattern$ = `"sdswierstra"`

On every line we unroll the pattern, and at each transition from 0 to 1 (from left to right) one character is dropped from the pattern, and at each transition from 1 to 0 a character is added to the pattern. More generally, at each transition from m to n the pattern is extended with $m-n$ characters (shortened, if $m-n$ is negative). First, we translate every line (one string) in *depths* to a list of numbers:

$$depths' = map\ (map\ f)\ depths \qquad \text{where } f\ c = code\ c - code\ \texttt{'0'}$$

Quite arbitrarily we start every next line with a different rotation of *pattern*:

$$\begin{array}{lcl} cycle & = & concat\ \cdot\ repeat \\ randomseq & = & cycle\ [0,2,7,3,11,5,9,4,8,1,6,10] \\ rotation\ xs\ i & = & drop\ i\ xs +\!\!+ take\ i\ xs \\ patterns & = & map\ (rotation\ pattern)\ randomseq \end{array}$$

So, $patterns$ = [`"sdswierstra"`, `"swierstrasd"`, `"strasdswier"`, `"wierstrasds"`, ...].
We join every line from $depths'$ with one of the patterns, and transform each pair to one line of the output picture:

$$picture = map\ mkline\ (zip2\ patterns\ depths')$$

It only remains to define *mkline*:

$$mkline\ (pat, ds) = \ldots$$

Parameter *pat* gives the pattern that we unroll along the list *ds* (*ds* is one line of $depths'$, and consists of *d*epth numbers). The unrolled pattern is *cycle pat*. We construct each line character by character, on the basis of the depth numbers *ds*.
So, schematically we have:

$$mkline\ (pat, ds) = mkln\ \ldots\ (cycle\ pat)\ ds$$

$$\begin{array}{lcl} mkln\ \ldots\ (p:ps)\ [\,] & = & [\,] \\ mkln\ \ldots\ (p:ps)\ (d:[\,]) & = & p:[\,] \\ mkln\ \ldots\ (p:ps)\ (d:d':ds) & & \\ = \quad p:\ mkln\ \ldots\ (cycle\ patH)\ (d':ds), & & \text{if } d < d' \quad \text{—— go Higher} \\ = \quad p:\ mkln\ \ldots\ ps\ \ (d':ds), & & \text{if } d = d' \quad \text{—— stay flat} \\ = \quad p:\ mkln\ \ldots\ (cycle\ patL)\ (d':ds), & & \text{if } d > d' \quad \text{—— go Lower} \end{array}$$

where ...

At a depth transition in $d : d' : ds$ we need some extra information, namely characters that need be added to the current pattern at a high-low transition, and the current pattern itself, so that the new pattern can be constructed. We store the characters in a list *stock* that is manipulated in a *last in first out* regime. The initial pattern in a line is taken to be *cycle pat*, which is named $(p : ps)$ in function *mkln*. The current pattern is *take n ps*, where n is the length of the pattern.

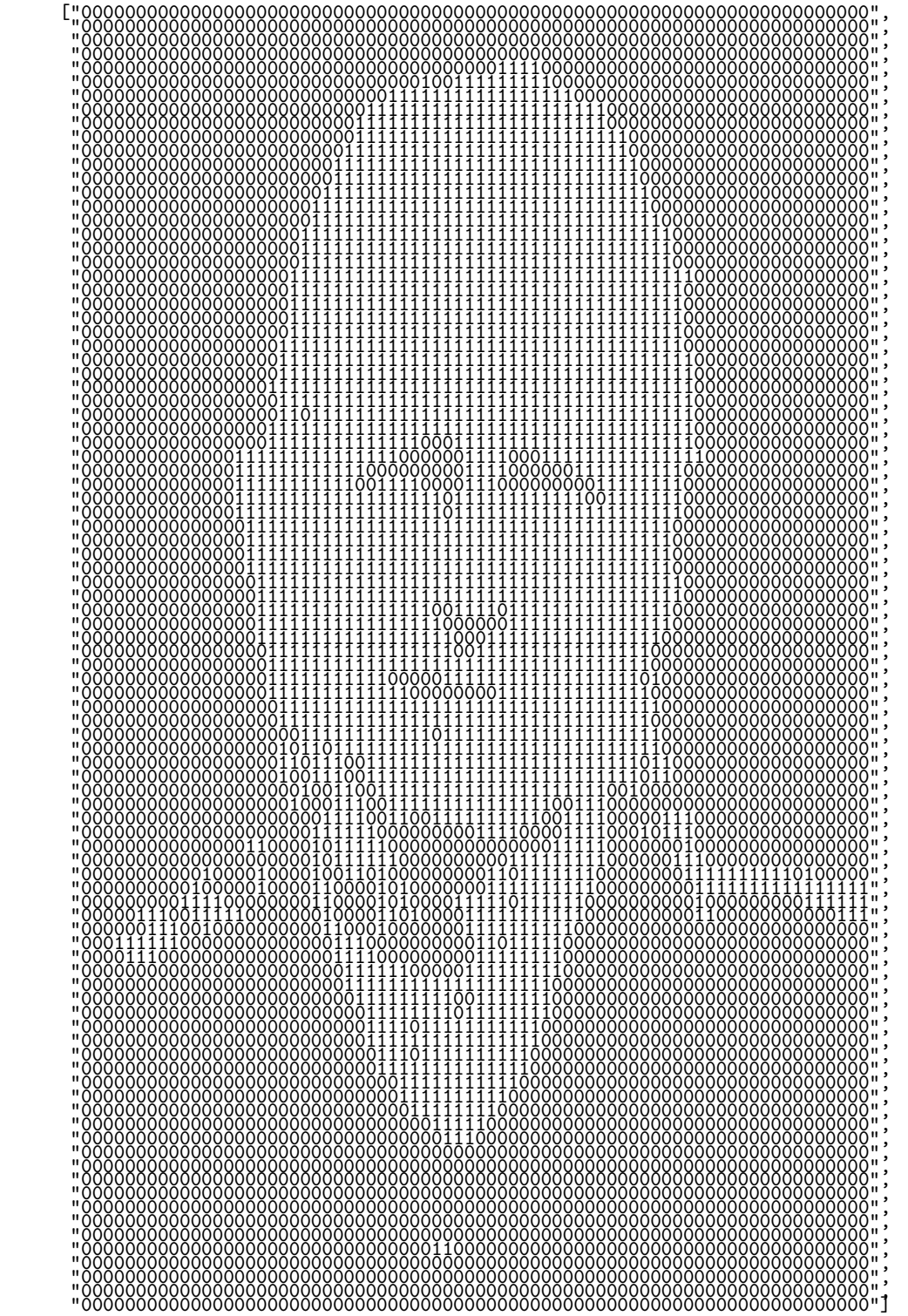


Fig. 1. Doaitse; the 0's denote background for Fig 2, the 1's denote foreground.

So, the preceding schematic definition is completed as follows:

$$
\begin{array}{lcll}
\multicolumn{4}{l}{mkline\ (pat, ds) \quad = \quad mkln\ (cycle\ pat)\ (\#pat)\ (cycle\ pat)\ ds} \\
\multicolumn{4}{l}{} \\
\multicolumn{4}{l}{mkln\ stock\ n\ (p : ps)\ [\,] \quad = \quad [\,]} \\
\multicolumn{4}{l}{mkln\ stock\ n\ (p : ps)\ (d : [\,]) \quad = \quad p : [\,]} \\
\multicolumn{4}{l}{mkln\ stock\ n\ (p : ps)\ (d : d' : ds)} \\
= & p : mkln\ stockH\ nH\ (cyclepatH)\ (d' : ds), & \text{if } d < d' & \text{—— go Higher} \\
= & p : mkln\ stock\ n\ ps\ \ (d' : ds), & \text{if } d = d' & \text{—— stay flat} \\
= & p : mkln\ stockL\ nL\ (cyclepatL)\ (d' : ds), & \text{if } d > d' & \text{—— go Lower}
\end{array}
$$

where

$$
\begin{array}{lcll}
pat & = & take\ n\ ps & \text{—— current pattern} \\
patH & = & drop\ (d'-d)\ pat & \text{—— H: pat decreases} \\
nH & = & n - (d'-d) & \\
stockH & = & (reverse\ .\ take\ (d'-d))\ pat +\!\!+ stock & \\
patL & = & take\ (d-d')\ stock +\!\!+ pat & \text{—— L: pat increases} \\
nL & = & n + (d-d') & \\
stockL & = & drop\ (d-d')\ stock &
\end{array}
$$

Similarly to *depths*, output *picture* is a list of strings. The strings are shown on separate lines by function *lay* with the property *lay* `["abc","defg",...]` = `"abc\ndefg\n..."`. Figure 2 displays *lay picture*.

To help to correctly view the picture, an additional line has been added at the top with two plus symbols at a distance of the length of *pattern*. You should point your eyes to a point *behind* the paper in such a way that you see the two plus symbols as one plus symbol with two vague plus symbols beside. Keep this position of your eyes for a while, and automatically your eyes will accommodate in such a way that the characters on paper will be sharp and clear. You'll see a kind of wall-paper (entirely printed with repetitions of `sdswierstra`) in which Doaitse's head (with holes at the position of the eyes and mouth) protrudes in the foreground.

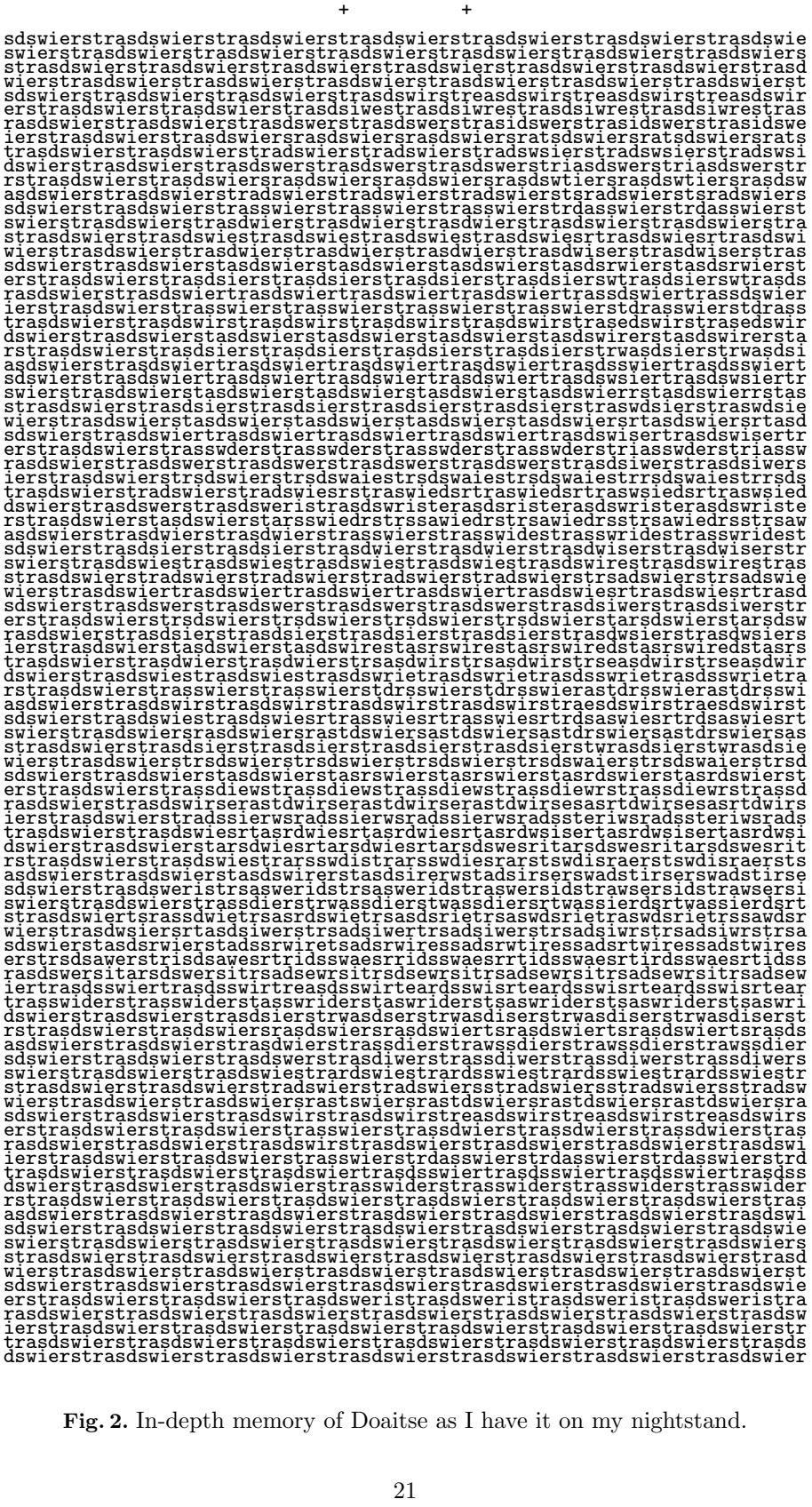


Fig. 2. In-depth memory of Doaitse as I have it on my nightstand.

A Short Example of the Calculational Method

Roland Backhouse[1]

roland.backhouse@nottingham.ac.uk
School of Computer Science, University of Nottingham, Nottingham NG8 1BB, UK

Abstract. Dedicated to Doaitse Swierstra on the occasion of his (formal) retirement: a short, calculational proof of a "challenging" logical problem.

1 Introduction

I first met Doaitse more than twenty five years ago in connection with the STOP (Specification and Transformation of Programs) project. Although I was never formally involved in the project I, like many others, was very much an active participant. Indeed, the project dominated my research activities during the first few years that I was in the Netherlands and for several years afterwards. Through it I got to know and to admire Doaitse's wit, his determination and his dedication to his subject.

The STOP project didn't just have influence on those who participated in it; it also had, I believe, much influence on the development of functional programming — at least more influence than some would acknowledge. The success of the project was due in part to the fact that it didn't conform to the usual norms for research projects; above all, it was not about maximising numbers of publications and/or one's personal kudos index but it was about cooperation in pursuit of a common goal. This was the spirit that Doaitse imbued in the project and I shall be forever grateful to him for the opportunity to take part.

Scientifically, I remember STOP as being about the pursuit of elegance through abstraction and calculation. Although not about programming, I know that the following small calculation would have been welcomed by those taking part, especially Doaitse.

2 The Calculation

Gries [Gri96] reports his solution to proving that, for arbitrary predicates p and q,

$$\begin{array}{lllllll} & \langle \exists x :: \langle \forall y :: p.x \equiv p.y \rangle \rangle & \equiv & \langle \exists x :: p.x \rangle & \equiv & \langle \forall y :: p.y \rangle \\ \equiv & \langle \exists x :: \langle \forall y :: q.x \equiv q.y \rangle \rangle & \equiv & \langle \exists x :: q.x \rangle & \equiv & \langle \forall y :: q.y \rangle \quad . \end{array}$$

(NB. The multiple occurrences of equivales must be read associatively.) This is a problem posed to him by two implementors of mechanical theorem provers, and called "Andrew's challenge".

Gries conjectures that the expression in p and the expression in q are both true, and hence equal. He then proves the truth of the expression in p by mutual implication. The problem I posed myself was to simplify

$$\langle \exists x :: \langle \forall y :: p.x \equiv p.y \rangle \rangle$$

in the sense of separating the existential and universal quantifications. So, I was looking for an equation of the form

$$\langle \exists x :: \langle \forall y :: p.x \equiv p.y \rangle \rangle \;=\; \langle \exists x :: p.x \rangle \;\oplus\; \langle \forall y :: p.y \rangle$$

for some operator "$\oplus$". (In view of Andrew's challenge, examples of "$\oplus$" could be given by

$$u \oplus v \equiv u \equiv v \quad ,$$

$$u \oplus v \equiv u \equiv v \equiv \mathsf{false} \quad ,$$

and

$$u \oplus v \equiv u \equiv v \equiv \langle \exists x :: \mathsf{true} \rangle \quad .)$$

This is the calculation.

$$\begin{array}{ll}
 & \langle \exists x :: \langle \forall y :: p.x \equiv p.y \rangle \rangle \\
= & \quad \{ \quad \text{range splitting — aiming to extract "}p.x\text{"} \quad \} \\
 & \langle \exists x : p.x : \langle \forall y :: p.x \equiv p.y \rangle \rangle \;\vee\; \langle \exists x : \neg(p.x) : \langle \forall y :: p.x \equiv p.y \rangle \rangle \\
= & \quad \{ \quad \text{(preparing the way for the use of Leibniz's rule)} \\
 & \qquad \mathsf{true} \text{ is the unit of boolean equality, definition of negation} \quad \} \\
 & \quad \langle \exists x \;:\; p.x \equiv \mathsf{true} \;:\; \langle \forall y :: p.x \equiv p.y \rangle \rangle \\
 & \vee \quad \langle \exists x \;:\; p.x \equiv \mathsf{false} \;:\; \langle \forall y :: p.x \equiv p.y \rangle \rangle \\
= & \quad \{ \quad \text{Leibniz (i.e. substitution of equals for equals)} \quad \} \\
 & \quad \langle \exists x \;:\; p.x \equiv \mathsf{true} \;:\; \langle \forall y :: \mathsf{true} \equiv p.y \rangle \rangle \\
 & \vee \quad \langle \exists x \;:\; p.x \equiv \mathsf{false} \;:\; \langle \forall y :: \mathsf{false} \equiv p.y \rangle \rangle \\
= & \quad \{ \quad \mathsf{true} \text{ is the unit of boolean equality, definition of negation} \quad \} \\
 & \langle \exists x : p.x : \langle \forall y :: p.y \rangle \rangle \;\vee\; \langle \exists x : \neg(p.x) : \langle \forall y :: \neg(p.y) \rangle \rangle \\
= & \quad \{ \quad \text{distributivity, trading} \quad \} \\
 & (\langle \exists x :: p.x \rangle \;\wedge\; \langle \forall y :: p.y \rangle) \;\vee\; (\langle \exists x :: \neg(p.x) \rangle \;\wedge\; \langle \forall y :: \neg(p.y) \rangle) \\
= & \quad \{ \quad \text{De Morgan} \quad \} \\
 & (\langle \exists x :: p.x \rangle \;\wedge\; \langle \forall y :: p.y \rangle) \;\vee\; (\neg \langle \forall x :: p.x \rangle \;\wedge\; \neg \langle \exists y :: p.y \rangle) \\
= & \quad \{ \quad \text{symmetry of } \wedge\text{, renaming of dummies} \quad \} \\
 & (\langle \exists x :: p.x \rangle \;\wedge\; \langle \forall y :: p.y \rangle) \;\vee\; (\neg \langle \exists x :: p.x \rangle \;\wedge\; \neg \langle \forall y :: p.y \rangle) \\
= & \quad \{ \quad [X \;\equiv\; Y \;\equiv\; (X \wedge Y) \vee (\neg X \wedge \neg Y)] \quad \} \\
 & \langle \exists x :: p.x \rangle \;\equiv\; \langle \forall y :: p.y \rangle \quad .
\end{array}$$

In these eight steps we have established that

$$\langle \exists x :: \langle \forall y :: p.x \equiv p.y \rangle \rangle \;\equiv\; \langle \exists x :: p.x \rangle \;\equiv\; \langle \forall y :: p.y \rangle \quad .$$

The final step in the proof of "Andrew's challenge" is to apply Leibniz's rule again (as did Gries in his report).

In retrospect, it is a wonder that the problem is called a "challenge". The reason that most of us —I include myself here— might find the problem difficult is that we have been

brought up to think of boolean equality as "if and only if". This is like thinking of equality of numbers as "at most and at least"! As a consequence, we fail to exploit Leibniz's rule, and it is precisely the use of this rule (in the third step above) that is crucial to the above calculation. (In practice, I would conflate the first three steps to just one, not making explicit the fact that `true` is the unit of boolean equality and the definition of negation. I have spelt these steps out here in order to make the use of Leibniz's rule more clear.)

References

[Gri96] David Gries. A calculational proof of Andrew's Challenge. http://www.cs.cornell.edu/Info/People/gries/gries.html, August 1996.

Subkwadratische SDS Verificatie

Gerard Tel
Informatica, Univ. Utrecht

Samenvatting We definiëren een gegeneraliseerde versie van de elfproef, de p-proef op Slimme Drentse Service nummers, en bewijzen dat het uitvoeren van deze proef subkwadratisch is in de lengte van SDS.

1 Inleiding: BSN en Lange SDS

Inwoners van Drente hebben een BurgerServiceNummer van 9 cijfers. Doordat het aantal inwoners schrikbarend toeneemt, zullen in de toekomst langere nummers nodig zijn: invoering van het Slim Drents Service nummer (SDS) is aanstaande. De elfproef beveiligt SDS niet tegen elke verwisseling van cijfers, zodat ook een hogere-orde test moet worden ingevoerd.

2 De p-Proef

Een SDS is een integer b met een decimale representatie van n cijfers (uniek gedefinieerd door $\sum_{i=0}^{n-1} c_i 10^i = b$ en $c_i < 10$), dus $10^{n-1} \leq b < 10^n$, dat voldoet aan de p-proef. Eis op de p-proef is: als van b (a) één cijfer fout wordt ingegeven, of (b) twee cijfers worden verwisseld, dan is het nieuwe getal niet p-proef.

Zij p (i) een priemgetal, (ii) groter dan de radix (10) en (iii) groter dan n. De p-som vermenigvuldigt cijfer c_i met de vaste factor $f_i = i + 1$ en sommeert: $S = \sum_i f_i c_i$ en de p-proef slaagt als $p|S$ (dwz. p deelt S).

Als (b) het i^{e} en j^{e} cijfer worden verwisseld krijgen we getal b' met

$$S' = S - f_i.c_i - f_j.c_j + f_i.c_j + f_j.c_i = S - (f_i - f_j)(c_i - c_j).$$

De factor $f_i - f_j$ is niet deelbaar door p (omdat $p > n$ en $1 \leq f_i, f_j \leq n$), de factor $c_i - c_j$ niet (omdat $p > 10$), dus het product ook niet (omdat p priem is). Dus als b aan de p-proef voldoet, dan voldoet b' niet.

In afwijking van voetnoot 20 van de wet (Regeling burgerservicenummer) gebruikt de Drentse elfproef op BSNs de waarde $f_0 = 10$ in plaats van $f_0 = 1$. De bruikbaarheid van de test tegen typefouten lijdt hier niet onder (bewijs dit?), en het is een handig weetje voor bij de borrelpraat.

3 Naieve en Divide-and-Conquer p-Proef

Laat b gegeven zijn als integer in onbeperkte precisie en p en n (en radix $r = 10$) zijn integers die in een machinewoord passen. Je kunt de p-som berekenen (linker pseudocode) door de cijfers van minst naar meest significant te bepalen met een deling door de radix, en met hun factor op te tellen. Helaas kost een deling van b door een kleine integer (ongelijk de radix van representatie in de machine) een tijd *lineair in de lengte* van het getal, zodat deze naieve aanpak $O(n^2)$ tijd kost.

```
S = 0; f = 1;
for (i=0; i<n; i++)
  { (b, c) = (b/10, b%10);
    S += f*c; f++;
  }
```

```
int dcsum(long b, int n, int f)
   if (n == 1) return f * b;
   m = n/2; M = r^m;
   (agnes, doaitse) = (b/M, b%M);
   return dcsum(agnes  , n-m, f+m)
        + dcsum(doaitse, m  , f  );
```

De rechter pseudocode gebruikt Divide-and-Conquer en verdeelt SDS b met een deling door $10^{\lfloor n/2 \rfloor}$ in twee getallen van $\lfloor n/2 \rfloor$ en $\lceil n/2 \rceil$ cijfers. Per helft wordt recursief de p-som berekend, waarbij parameter f de factor van het minst-significante cijfer doorgeeft.

4 Analyse

Wat kost het om SDS in twee delen te splitsen? Zij $D(n)$ de tijd nodig om een getal $b < 10^n$ te delen door $10^{\lfloor n/2 \rfloor}$. De rekentijd $T(n)$ van `dcsum` wordt dan gegeven door $T(n) = D(n) + 2.T(n/2)$. Een klassieke staartdeling is zelf kwadratisch, $D_{\mathbf{s}}(n) = O(n^2)$ wanneer de deler half zo lang is als het deeltal, en is dus niet geschikt om de p-proef te versnellen.

Je kunt ook de inversen van de gebruikte radix-machten opslaan (dat zijn er n, te verbeteren tot $\lg n$) en de deling uitvoeren als vermenigvuldiging. Het algoritme van Karatsuba haalt $D_{\mathbf{k}}(n) = O(n^{\lg 3})$ en Toom-Cook geeft $D_{\mathbf{t}}(n) = O(n^{^3 \log 5})$. In beide gevallen domineert de eerste deling de complexiteit van de gehele p-proef: $T_{\mathbf{k}}(n) = O(n^{\lg 3}) \approx n^{1{,}585}$, resp. $T_{\mathbf{t}}(n) = O(n^{^3 \log 5}) \approx n^{1{,}465}$.

5 Tenslotte...

Je kunt alle begrippen nazoeken op Wikipedia en `wetten.nl`.

Aan p-proef eis (a) is voldaan als p tenminste de radix is.

Met Fast Fourier Transform kun je delen in $D_{\mathbf{f}} = O(n \lg n)$ tijd, wat $T_{\mathbf{f}} = O(n \lg^2 n)$ geeft, maar ik hoop niet dat Drente zo vol wordt dat dit lonend is.

Je hebt aan $\lg n$ inverse radixmachten genoeg als je in `dcsum` voor `m` een tweemacht kiest (grootste tweemacht kleiner dan n).

Is de recursieve methode ook parallel sneller? De naieve methode kan de n delingen pipelinen en met n processors in $O(n)$ tijd de p-proef voltooien.

Doaitse, geniet met je meest-significante wederhelft Agnes van de jaren die voor je liggen! Ik bedank je voor je inzet voor onze universiteit en hoop dat je, dit stukje lezend bij je haardvuur of meer, nog eens terug denkt aan je ex-collega's uit Utrecht,

Gerard Tel, 30 mei 2013.

The history of finding palindromes

Johan Jeuring[1,2]

[1] Department of Information and Computing Sciences, Universiteit Utrecht
[2] School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
J.T.Jeuring@uu.nl

Abstract. This paper describes the history of finding palindromes in computer science. The problem of determining whether or not a string is a palindrome is one of the oldest computer science problems, and algorithms for this problem have been constructed since the early years of computer science. This paper describes the contributions to solving algorithmic problems related to finding palindromes and variants of palindromes.

1 Introduction

Fig. 1. The Sator square, from a Swedish manuscript, 1722, Skara, Sweden [34]

Since 1998 I have worked in the group "SDS", the name used for the Software Technology group in internal documents of the department of Information and Computing Sciences of Utrecht University. The group is called after the initials of Doaitse Swierstra. The most remarkable aspect of these initials is that they constitute a *palindrome*. Until a successor of Doaitse has been appointed, I will be group leader of the Software Technology group, and, preserving all properties, its name should probably change to "JTJ".

Doaitse thinks palindromes are boring. He and I wrote a set of lecture notes together on 'Languages and compilers' [23], which uses palindromes as one of its first examples. One of Doaitse's returning remarks about these notes is that it is so boring to use palindromes as an example. In this paper I show that palindromes have a rich history, also in computer science, and that they are a worthwhile object of study. I will refer to many books and papers about

formal languages, computing models, and algorithms, in which palindromes are used as first or second example, or as inspiration for other results. The books and papers in the list of references of this paper have been cited almost 40.000 times, and this is probably my paper with the most cited citations. I doubt it will be sufficient for Doaitse to change his mind, however. Since Doaitse will hopefully not be the only reader of this paper, I will also include some descriptions of concepts of which Doaitse is well aware, so that readers with a different computer science background can also learn something about the history of palindromes in computer science.

The palindrome concept has a long history. Already around 2000 years ago, the palindrome 'Rotas opera tenet arepo sator', see also Figure 1, was written on the walls of Pompeii. Its precise meaning is unclear. Some 300 years before that, around 250 BC, Sotades reportedly wrote the first verses that also made sense if read backwards. Unfortunately, none of the palindromic verses of Sotades survived.

The oldest Dutch palindrome I could find is 'Neder sit wort trow tis reden', from 1584, see Figure 2. The meaning of this sentence is unclear too, one possibility is 'Humility is the word, loyal the intellect'

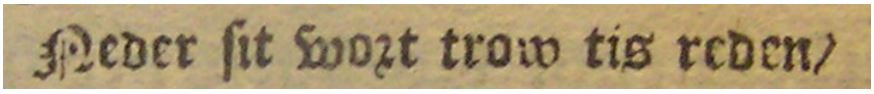


Fig. 2. Dutch palindrome ('letter-kreeftdicht') from 1584 [33]

Palindromes have long been considered interesting curiosities used in word-plays. We now know that palindromes play an important role in DNA. If I search for the keyword palindrome in the electronic publications available at the library of Utrecht University, I get more than 500 hits. The first ten of these hits are all about palindromes in DNA. My guess is that at least 90% of these 500 publications are about palindromes in DNA. For example, the male DNA contains huge approximate palindromes with gaps in the middle [30]. Some of these palindromes are more than a million base-pairs long. The genes for male testes are encoded on one of these palindromes. The male chromosome, XY, consists of a single X and a single Y, and is the only chromosome that does not have two copies of the same DNA string. An important reason why chromosomes have two copies of the same string is to repair possible errors in DNA. Since the Y chromosome lacks a copy, it needs to resort to other mechanisms for repairing itself. The Y chromosome uses palindromes for this purpose: the important genetical information in Y appears in palindromes. Thus palindromes play an important role in saving males from extinction...

We need software to find palindromes in large pieces of text, or approximate palindromes with gaps in DNA. Algorithms for determining whether or not a string is a palindrome, and finding palindromes in strings have a long history in computer science, longer than Doaitse's career. On the occasion of Doaitse's retirement I want to look back to a very small part of the history of computer science, and revisit the history of algorithms for finding palindromes.

2 Finding palindromes

2.1 Formal language theory

Palindromes have been used as examples in formal language theory, and to illustrate the power of various computing models.

The classic book of Noam Chomsky introducing syntactic structures for languages from 1957 [6] uses palindromes as its second example. The first example of a formal language in Hopcroft and Ullman's Introduction to Automata Theory, Languages and Computation [20] is the language of palindromes. A predecessor of this book [19] shows that the language of even-length palindromes is a nondeterministic context-free language: a context-free language that can be recognized by means of a pushdown automaton, but not by a deterministic pushdown automaton. This implies that this language is context-free, but 'of a more involved' kind.

Stephen Cole showed how to recognize palindromes on iterative arrays of finite-state machines in 1964 [7,8], and so did Seiferas on iterative arrays with direct central control [29]. Alvy Ray Smith III used cellular automata to recognize palindromes in 1971 [32]. Also in the 1970s, Fréjvald [13] and Yao [36], independently, looked at recognizing palindromes by means of probabilistic Turing machines. In 1965, Fréjvald [12] and Barzdins [3] independently showed that it requires a number of steps quadratic in the length of the input string on a Turing machine with a single head and a single tape to determine whether or not a string is a palindrome. In his review of these papers, Mullin [27] conjectured that this problem can be solved in linear time on a machine with two heads. Eight years later in 1973, Slisenko [31] showed that if you are allowed to use more than one head on a tape, or multiple tapes, then indeed a palindrome can be determined in a number of steps linear in the length of the input string. Looking at these results from a distance of around 40 years, it seems like in those days it was a sport to study programming with various restrictions, like not using your left hand, or tying your legs together.

The English translation of Slisenko's paper is a 183 page, dense mathematical text. Slisenko announced his result already in 1969, and given the form and the length of his paper (with 183 pages this is more a book than a paper), I find it highly likely that it took him a couple of years to write it. Slisenko's result was a surprisingly strong result, and people started to study and trying to improve upon it. One of these people was Zvi Galil, then a postdoc IBM Yorktown Heights. He had a hard time understanding Slisenko's paper, which I fully understand, but in 1978 he obtained the same result, only he needed just 18 pages to describe it. The problem of finding palindromes efficiently started off an area within computer science now known as stringology, which studies strings and their properties, and algorithms on strings.

2.2 Stringology

A nice example of how palindromes influenced the development of well-known algorithms is given by Knuth in Knuth, Morris and Pratt's paper about fast pattern matching in strings [24]. Daniel Chester had developed a program to recognize strings beginning with an even-length palindrome using a two-way deterministic pushdown automaton. Knuth had just learned about Cook's theorem, which stated that any language recognizable by a two-way deterministic pushdown automaton can be recognized in linear time on a Random Access Machine (RAM) [9]. After he had successfully applied Cook's procedure to obtain a linear-time RAM algorithm for finding even-length palindromes at the start of a string, he realised that he could use a similar procedure to obtain a fast algorithm for pattern

matching. This algorithm later became know as the Knuth, Morris, Pratt (KMP) pattern matching algorithm. Around the same time, and in a similar way by moving between different computation models, Galil found a linear-time algorithm for finding initial palindromes of any length [15].

Some of the machine models on which algorithms for palindromes were developed have rather artificial restrictions, which gives rather artificial algorithms for finding palindromes. But some of the algorithms on the more realistic machine models, such as the RAM model, contain the essential components of a by now relatively well-known linear-time algorithm for finding palindromes. Manacher [26] gives a linear-time algorithm on the RAM computing model finding the smallest initial palindrome of even length. The difference with the algorithms described above is that this algorithm is 'on-line': it finds palindromes as it reads input symbols, and it doesn't need to see the complete input string. He also describes how to adjust his algorithm in order to find the smallest initial palindrome of odd length longer than 3. He did not realize that his approach could be used to find all maximal palindromes in a string in linear time. Zvi Galil and Joel Seiferas did, in 1976. They wrote a paper, titled 'Recognizing certain repetitions and reversals within strings' [17], in which they develop an algorithm that finds all maximal palindromes in a string in linear time. As far as I know, this is the first description of this algorithm.

Twelve years later I rediscovered this algorithm. I published a paper on 'The derivation of on-line algorithms, with an application to finding palindromes' [22], in which I show how to obtain the efficient algorithm for finding palindromes from the naive algorithm for finding palindromes using algebraic reasoning. The method at which I arrive at the algorithm is completely different from the way Galil and Seiferas present their version. I presented the algorithm and the method I use to construct it to several audiences. I was only 22 years old at the time, and I am afraid my presentation skills were limited. Several people that saw me presenting the efficient algorithm for finding palindromes thought they could do better. An example can be found in the book 'Beauty is our business' (which isn't about models, or escort services, but about computer science, and dedicated to the Dutch Turing award winning computer scientist Edsger Dijkstra on his sixtieth birthday), in which Jan Tijmen Udding derives the same algorithm using Dijkstra's calculus for calculating programs [35].

The stringology community went on to develop algorithms with even more refined features for finding palindromes, such as a *real-time* algorithm, an algorithm that finds all palindromes in a string, but only uses a constant amount of time after reading each input symbol [14].

2.3 Finding palindromes in parallel

Zvi Galil developed algorithms for almost all problems related to palindromes in his series of papers on palindromes. In 1985, he developed an algorithm for finding initial, even-length, palindromes using a parallel machine [16] in time logarithmic in the length of the input string. Turing machines, and the other machine models mentioned above, are sequential machines: computations are performed in sequence, and not at the same time. Parallel machines allow computations to be performed at the same time, in parallel. In the previous century few parallel machines were available, but nowadays, even the laptop on which I am typing this text has four cores, and can do many things in parallel. The importance of parallel machines has increased considerably, also because it is expected that increasing the speed of computers by increasing the clock speed is not going to work anymore in the next couple of years. Extra power of computers has to come from the possibilities offered by using multiple cores for computations. To give an example: I searched for palindromes in the human Y chromosome using the efficient, linear-time, on-line algorithm. Since this chromosome has more than 20

million base-pairs, it takes quite some time (in the order of minutes) to find palindromes in it.

To put multiple cores to good use, we need efficient parallel algorithms for the problems we want to solve. Apostolico, Breslauer, and Galil improved upon Galil's first parallel algorithm for finding palindromes in their paper on 'Optimal Parallel Algorithms for Periods, Palindromes and Squares' in 1992 [1], based on an approach to use overhanging occurrences of strings to find initial palindromes introduced by Fischer and Paterson [11]. This optimal algorithm takes $log\ (log\ n)$ time using $n\ (log\ n)$ processors, where n is the length of the input string. The number of processors available to a user is usually fixed, and does not depend on the length of an input string. My machine has four cores, and it is impossible to change that number. Given the number of available processors, Breslauer and Galil determine how much time a parallel algorithm takes to find initial palindromes [4].

There are quite a few different different parallel architectures, depending on for example whether or not two or more processors can read or write a symbol from memory simultaneously. If we want to use one of these architectures to find palindromes, we need an algorithm specifically developed for this architecture, and we can construct different algorithms that optimize time, space, or use a particular number of processors. Further algorithms for finding palindromes on various parallel architectures have been developed by Crochemore and Rytter [10] and, again, Apostolico, Breslauer, and Galil [2].

2.4 Gapped and approximate palindromes

Since the discovery that DNA contains many palindromes, people working on bioinformatics have developed algorithms for finding palindromes in DNA. The first description of these algorithms I could find are in the book Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, by Dan Gusfield [18]. This is one of the standard works in computational biology. Gusfield shows how to compute all maximal palindromes in a string using a suffix tree. He then moves on to discuss gapped palindromes (called separate palindromes in Gusfield's book), and approximate palindromes. A gapped palindrome is a palindrome with a gap in the middle, and an approximate palindrome is a palindrome after a limited number of symbols is changed, deleted or inserted. These kinds of palindromes are important in computational biology, because, for example, the very long palindromes in the Y chromosome all have gaps, and a limited number of 'errors'. Gusfield leaves it to the reader to construct an algorithm that returns approximate palindromes in which only symbols can be changed (not deleted or inserted) in time linear in the product of the maximum number of errors and the length of the input.

After Gusfield, several other scientist have worked on finding gapped and approximate palindromes, sometimes improving on or generalizing Gusfield's results. For example, Porto and Barbosa have developed an efficient algorithm that finds approximate palindromes in which also symbols may be inserted or deleted [28], Kolpakov and Kucherov have developed several algorithms for determining palindromes with gaps [25], and Hsu, Chen, and Chao find all approximate gapped palindromes [21].

3 Conclusions

This short paper shows the history of describing and finding palindromes in computer science by discussing the literature on this topic. I have not included all literature about variants of the problem of finding palindromes: the number of variants is substantial, and listing the

literature about all variants is infeasible. For example, I recently came across a paper that shows how to find approximate palindromes in run-length encoded strings [5].

I expect we will see solutions to more variants of the palindrome problem in the future. For example, the papers about parallel algorithms for finding palindromes listed in this paper all describe algorithms for finding exact palindromes. Parallel algorithms are particularly useful for huge input strings, such as the human Y chromosome. The palindromes occurring in the Y chromosome are gapped and approximate, so we need to develop variants of the parallel algorithms for gapped approximate palindromes. Also from an algorithm-design perspective I think there are still some open questions. The central concept in the design of the efficient algorithm for finding palindromes is the palindromes in palindromes property. This property says that if a large palindrome contains a smaller palindrome that does not appear exactly in the middle, the large palindrome contains a second copy of the smaller palindrome at the other arm of the large palindrome. I think this property should also play a central role in efficient algorithms for finding gapped and approximate palindromes. I have tried for quite a while to design algorithms for these problems using the palindromes in palindromes concept, but failed. I hope someone else will solve this problem.

References

1. Alberto Apostolico, Dany Breslauer, and Zvi Galil. Optimal parallel algorithms for periods, palindromes and squares (extended abstract). In *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 296–307. Springer, 1992.
2. Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141(1 - 2):163–173, 1995.
3. A.M. Barzdin. Complexity of recognition of symmetry on Turing machines (in Russian). *Problémy kibérnétiki*, 15:245–248, 1965.
4. Dany Breslauer and Zvi Galil. Finding all periods and initial palindromes of a string in parallel. *Algorithmica*, 14(4):355–366, 1995.
5. Kuan-Yu Chen, Ping-Hui Hsu, and Kun-Mao Chao. Efficient retrieval of approximate palindromes in a run-length encoded string. *Theoretical Computer Science*, 432(0):28–37, 2012.
6. Noam Chomsky. *Syntactic Structures*. Mouton, The Hague, 1957.
7. Stephen N. Cole. *Real-time computation by iterative arrays of finite-state machines*. PhD thesis, Harvard University, 1964.
8. Stephen N. Cole. Real-time computation by n-dimensional iterative arrays of finite-state machines. *IEEE Transactions on Computers*, C-18(4):349–365, 1969.
9. Stephen A. Cook. Linear time simulation of deterministic two-way pushdown automata. In *IFIP Congress (1)*, pages 75–80, 1971.
10. Maxime Crochemore and Wojciech Rytter. Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays. *Theoretical Computer Science*, 88(1):59–82, 1991.
11. Michael J. Fischer and Michael S. Paterson. String matching and other products. In R.M. Karp, editor, *SIAM AMS Proceedings on Complexity of Computation*, volume 7, pages 113–126, 1974.
12. R. Fréjvald. Complexity of recognition of symmetry on Turing machines with input (in Russian). *Algébra i logika, Séminar*, 4(1):47–58, 1965.
13. R. Fréjvald. Fast computation by probabilistic Turing machines. In *Theory of Algorithms and Programs*, volume 2, pages 201–205. Latvian State University, 1975.
14. Zvi Galil. Real-time algorithms for string-matching and palindrome recognition. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, STOC '76, pages 161–173. ACM, 1976.
15. Zvi Galil. Two fast simulations which imply some fast string matching and palindrome-recognition algorithms. *Information Processing Letters*, 4:85–87, 1976.
16. Zvi Galil. Optimal parallel algorithms for string matching. *Information and Control*, 67(1-3):144–157, 1986.

17. Zvi Galil and Joel I. Seiferas. Recognizing certain repetitions and reversals within strings. In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*, pages 236–252, 1976.
18. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.
19. John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata.* Addison-Wesley, 1969.
20. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
21. Ping-Hui Hsu, Kuan-Yu Chen, and Kun-Mao Chao. Finding all approximate gapped palindromes. In *Proceedings of the 20th International Symposium on Algorithms and Computation*, volume 5878 of *Lecture Notes in Computer Science*, pages 1084–1093. Springer-Verlag, 2009.
22. Johan Jeuring. The derivation of on-line algorithms, with an application to finding palindromes. *Algorithmica*, 11:146–184, 1994.
23. Johan Jeuring and Doaitse Swierstra. *Lecture notes on Languages and Compilers.* Utrecht University, Department of Information and Computing Sciences, 2000.
24. Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1978.
25. Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theoretical Computer Science*, 410(51):5365–5373, 2009.
26. Glenn Manacher. A new linear-time 'on-line' algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22:346–351, 1975.
27. Albert A. Mullin. Review of Fréjvald [12] and Barzdins [3]. *The Journal of Symbolic Logic*, 35(1):159, 1970.
28. Alexandre H.L. Porto and Valmir C. Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 35(11):2581–2591, 2002.
29. Joel I. Seiferas. Iterative arrays with direct central control. *Acta Informatica*, 8:177–192, 1977.
30. Helen Skaletsky, Tomoko Kuroda-Kawaguchi, Patrick J. Minx, Holland S. Cordum, LaDeana Hillier, Laura G. Brown, Sjoerd Repping, Tatyana Pyntikova, Johar Ali, Tamberlyn Bieri, Asif Chinwalla, Andrew Delehaunty, Kim Delehaunty, Hui Du, Ginger Fewell, Lucinda Fulton, Robert Fulton, Tina Graves, Shun-Fang Hou, Philip Latrielle, Shawn Leonard, Elaine Mardis, Rachel Maupin, John McPherson, Tracie Miner, William Nash, Christine Nguyen, Philip Ozersky, Kymberlie Pepin, Susan Rock, Tracy Rohlfing, Kelsi Scott, Brian Schultz, Cindy Strong, Aye Tin-Wollam, Shiaw-Pyng Yang, Robert H. Waterston, Richard K. Wilson, Steve Rozen, and David C. Page. The male-specific region of the human y chromosome is a mosaic of discrete sequence classes. *Nature*, 423(6942):825–837, 2003.
31. A.O. Slisenko. Recognizing a symmetry predicate by multihead Turing machines with input. In V.P. Orverkov and N.A. Sonin, editors, *Proc. of the Steklov Institue of Mathematics*, volume 129, pages 25–208, 1973.
32. Alvy Ray III Smith. Cellular automata complexity trade-offs. *Information and Control*, 18:466–482, 1971.
33. Hendrick Laurensz. Spieghel. *Twe-spraack vande Nederduitsche letterkunst.* Cristoffel Plantyn, Leiden, 1584.
34. Petter Jean Udd, Christian Papke, Petrus Sommar, and Tycho Brahe. *Samlingshandskrift med blandat, företrädesvis historiskt och praktiskt, innehåll.* 1765.
35. Jan Tijmen Udding. The maximum length of a palindrome in a sequence. In David Gries, editor, *Beauty is our business*, pages 410–416. Springer-Verlag New York, Inc., New York, NY, USA, 1990.
36. A.C. Yao. A lower bound to palindrome recognition by probabilistic Turing machines. Technical Report STAN-CS-77-647, Computer Science Department, Stanford University, 1977.

How do I Type Data?
For Doaitse, Emeriti Rule!

Arno Siebes

A.P.J.M.Siebes@uu.nl
Department of Information and Computing Sciences, Universiteit Utrecht

Abstract. One could say that I am spending my sabbatical to understand what I have been doing the past seven years. One of the things I would like to do is to implement some stuff myself, I haven't done that in aeons. Since I have often been told that well-typed programs are correct, I would like to implement in Haskell. But, embarrassingly enough, I am not even able to type my data correctly! Hence, I decided to misuse this Liber to ask Doaitse for help: Doaitse, Help!

1 Introduction

The question in the title of this tributary note – how do I type data? – seems remarkably stupid, even for a computer science professor. If you have data somewhere, you better know its format – otherwise it is just a meaningless string of 0's and 1's. One could employ that format implicitly in a program, or – much more wisely (ask Doaitse) – one could make it explicit by typing the data, i.e., the type is at least the format of the data and possibly more.

ECOL, the very first language I learned to program in[1], did not really have types. Variables stood for numbers, although Ints and Floats may have been distinguished. But all languages I encountered later, such as Algol 68, Algol 60, Fortran, and Pascal (in that order), did have types. In other words, I have been able to type my data from the seventies onward. So where does my question come from?

Simple, it comes from being a pattern set miner with a somewhat unorthodox view on data. So, before we can go into the details of my question, I should probably explain what pattern mining and pattern set mining are and how I see data.

2 Pattern Mining

A pattern is simply a query, a query whose answer on our data set is interesting; i.e., its answer is different from what the user expects. The goal of pattern mining is to find all these patterns in the data.

But, how do we know what answer the user expects? How do we know that for all queries one might execute on the data? The proper Bayesian way would be to formulate a prior on the possible answers to a given query and to do that for all possible queries. Somehow, most users are reluctant to invest that much time, if they have such priors at all.

A somewhat simplified approach is [1] to generalize the well-known equation

$$\text{data} = \text{model} + \text{random\_component}$$

[1] Using "schrapkaarten", i.e., punch cards where one coloured boxes black rather than punched holes

to

$$\text{data} = \text{background\_model} + \text{pattern(s)} + \text{random\_component}.$$

The background model tells us what to expect and in those areas where the data behaves (significantly) different from that model, we patch it up with patterns.

Note that we only need to know the "shape" of the background model, parameters can be estimated from the data. Unfortunately, even this is often too much to ask. Most data we encounter is no longer the familiar rectangular table filled with symbols, such as in table 1, that most of us remember from Statistics classes we took several too many years ago. Rather,

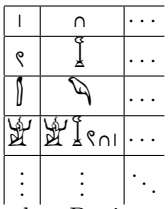


Table 1. How data looked like when Doaitse was young: tables with symbols.

we may have a set of receipts from a cash register, or a set of molecules, or a library of text documents, or a social network, or ... That is, not the kind of data for which the familiar statistical models would provide a good fit.

Hence, data miners take an even more simplified approach. We consider all patterns whose answer is, e.g., "big enough" as potentially interesting. When I said that patterns are queries, I should have been slightly more careful: patterns are selection queries. They select a subset of an extensionally or intensionally defined set of objects.

The size of this selected set – absolute or relative – is known as the *support* of a pattern. One of the games in data mining is *Frequent Pattern Mining*, i.e., finding all patterns whose support exceeds some given threshold [2].

If you think that finding all frequent patterns is very far from finding queries with, subjectively, interesting answers, you are not alone. One approach is to filter the frequent patterns with an *interestingness measure,* of which many have been devised [3, 4]. The other approach uses *constraints* or, more generally, *predicates* in what is sometimes known as *theory mining* [5]. Given a data set D, a collection of patterns $\mathcal{P}$, and selection predicate ϕ which evaluates whether or not $p \in \mathcal{P}$ describes an interesting subset of D, the *theory* of $\mathcal{P}$ in D is defined as:

$$\mathcal{T}h(\mathcal{P}, \phi, D) = \{p \in \mathcal{P} \mid \phi(p, D) = \text{true}\}.$$

While in general computing $\mathcal{T}h(\mathcal{P}, \phi, D)$ could be very expensive – e.g., if it turns out that $\mathcal{T}h(\mathcal{P}, \phi, D) = \mathcal{P}$ and $\mathcal{P}$ is very large – it is usually not that bad in practice; especially when some mild conditions on $\mathcal{P}$, D, and ϕ are satisfied. Frequent item set mining is a good example of this, every item set could be frequent but usually it isn't.

3 Pattern Set Mining

Whether one uses interestingness measures or constraints, one almost always ends up with a large, very large, collection of patterns. Far too large to allow a user to discover the really interesting ones by browsing through the collection. The reason is simple: the resulting sets

of patterns are highly redundant, e.g., because many patterns describe more or less the same subset. The patterns are mostly filtered in (almost) splendid isolation.

So, a better question is: can we find good sets of interesting queries? This is the provenance of pattern set mining [6–14]. Rather than mining for patterns and then decide which are interesting, we mine for *interesting sets of patterns* directly.

So, what makes a set of patterns good? For this, we can take our cue from David Hand's view on patterns. If we don't have a background model, it reduces to:

$$\text{data} = \text{pattern(s)} + \text{random\_component}.$$

That is, a good set of patterns is a set that collectively describe the data well. As in previous research [9], we turn to the Minimal Description Length (MDL) principle [15–17], to determine whether or not a set of patterns describes the data well. According to this principle, given a set of hypotheses, choose the one that compresses the data best. Slightly more precise, let $\mathcal{H} = \{H_1, \ldots, H_n\}$ be a set of hypotheses (models) for a data set D, then (crude) MDL says that we should select the $H_i \in \mathcal{H}$ that minimizes:

$$L(H_i) + L(D \mid H_i),$$

in which $L(H_i)$ is the size of H_i in bits and $L(D \mid H_i)$ is the size of D when encoded with H_i in bits.

To explain how we encode data with patterns, we first should explain what data is; well, how I view data.

4 My View on Data

The hallmark of the digital society is that we are surrounded by information. For almost any question there are resources that, given the right query, can provide the answer. These resources vary from (specialised) databases to document collections and from on-line libraries, to the web itself. One doesn't even need to know details of the resource itself, knowledge of the appropriate query interface is enough. Sometimes this is a tailor-made search tool implemented on top of the resource; in other cases a generic search engine is sufficient.

To most of us, this is exactly what a data source is: a set of answers to a set of queries. That is, we equate the data with the answers they yield to our questions:

> *the information contained in a collection of data equals the set of queries it answers, i.e., it is a set of queries and answers.*

This is what, to me, data is. That is, as far as the information contained is concerned, I am not interested in how the data is stored, in relational databases, as an XML document, or as a NoSQL database. Neither am I interested in the intricacies of query languages and interfaces. All I am interested in is questions and answers.

There are two further remarks with regard to my quasi-definition. The first one is that this "definition" of information isn't too different from the usual definition in Information Theory, i.e., bits [18]. After all, bits reduce the uncertainty with regard to something which can be regarded as the answer to one or more queries. The difference is one of emphasis, we make these queries explicit.

This is also the basis for my second remark: the quasi-definition views the information contained in a resource as something relative to the set of questions that are asked. Assume we have a fixed set of questions we are interested in, say Q. Moreover, assume we have two

different data collections D_1 and D_2. If D_1 and D_2 yield the same answer to each query in Q, then D_1 and D_2 are indistinguishable for us, i.e.,

$$(\forall q \in Q : q(D_1) = q(D_2)) \rightarrow D_1 =_Q D_2$$

There may be differences between D_1 and D_2, but we can't see them with Q as our tool. If Q is all we care about, we don't care about possible differences between D_1 and D_2, i.e., $[D_1 =_Q D_2] \equiv [D_1 = D_2]$.

As in the case of patterns, when I write queries, I mean selection queries. Not that I consider only simple SQL select statements, but in the sense that for individual "objects" – existing or newly created – there is a binary decision – i.e., standard information theory – whether or not they belong to the final result. Data collections are characterized by their answers to a given set of such selection queries.

5 Pattern Set Mining With a View on Data

A data set is a set of queries with their answer and a pattern set is a set of queries with their answer. The goal of pattern set mining is to find a set of patterns that together describe the data well. So, we are done: the sought after pattern set equals the set of queries; aren't we? The answer is, of course, *no*!

The crux is that we can describe – or store – the same information in many different ways. A first approach would be to simply store the complete set of queries with their answer. Another approach would be to store the data in some traditional way, e.g., as a (set of) relational tables, and to compute the answer to queries just as traditionally.

Under the first approach we need storage space in the order of the size of the set of queries, i.e., $O(|Q|)$. While it depends on the expressivity of Q how large $|Q|$ exactly is, it is safe to assume that it is exponential in the size of the original data set, i.e., $O(|Q|) = O(2^{|D|})$. Answering a query under this scheme is, however, very efficient, for all we have to do is to locate the right query and do a look-up for its answer, i.e., we need $O(\log(|Q|)) = O(|D|)$ time, regardless(!) of the expressivity of Q. All queries are answered in linear time!

Under the second approach we obviously need $O(|D|)$ space. How long it takes to compute an answer to a query $q \in Q$ on D depends very much on q itself and on the complexity of Q in general. While query languages are often restricted to some sub-class of polynomial algorithms, they are usually not restricted to linear algorithms only. After all, the more expressive our query language, the more structure we can discover in the data.

Both are two examples of a more general approach, store a subset $G \subseteq Q$ of all queries, such that the answer to any query in Q can be computed from the set G and the answers to G. In the first approach, $G = Q$, in the second it consists of D, i.e., one "indicator" query 1_d for each $d \in D$. if we call such a set a generating set, pattern mining is about finding good generating sets.

One of the factors in deciding what a good generating set is, is *comprehensibility*. Comprehensibility poses at least two constraints on generating sets:

- they should be small, sets with many thousands or even millions of queries are not much more comprehensible than the full set of queries.
- computing the result of a '$q \in Q \setminus G$ should be simple. If such a computation takes "forever", there is little chance the user "sees" what is going on.

The size of G is very much related to the expressivity of Q. The computational costs depend both on this expressivity – or better, its twin, the complexity of Q – and on what computations we allow. The first issue is not relevant here, the second is. We restrict computation to the *algebraic* structure of Q.

6 Structure in Data

Query languages are mostly more than just a set of queries, they have structure. Consider, e.g., the best-known pattern language: item sets [2]. The data is a bag of *transactions* and each transaction is a set of *items*. An item set is also a set of items. A transaction t satisfies an item set I iff $I \subseteq t$.

If we denote the set of items by $\mathcal{I}$, our pattern set $\mathcal{P}$ – our query language Q – equals $P(\mathcal{I})$. In other words, our query language is a lattice with operations such as $\cup$ and $\cap$ – or, more abstractly $\otimes$ and $\oplus$.

Such an algebraic structure allows us to "discover" *syntactical* relations such as:

$$q_1 = q_2 \otimes q_3.$$

In itself, this is not a very interesting observation. After all, it is independent of the data. Imagine how impressed your favourite supermarket chain will be with your discovery that:

$$\{\text{Beer}, \text{Diapers}\} = \{\text{Beer}\} \cup \{\text{Diapers}\}.$$

On the other hand, we can discover *semantical* relations in a data set D, such as:

$$q_1(D) = q_2(D) \times q_3(D).$$

In itself, such an observation isn't very interesting either. The fact that two randomly chosen queries – or expressions of queries – have the same answer doesn't have to mean anything. in this case, imagine how impressed they will be with your discovery that

$$\{\text{Jelly}\}(D) = \{\text{Beer}\}(D) \times \{\text{Diapers}\}(D).$$

Intrigued perhaps by the coincidence, but impressed? No. It *is* interesting if syntax and semantics *coincide*, e.g., if we have both:

$$q_1 = q_2 \otimes q_3 \text{ and } q_1(D) = q_2(D) \times q_3(D).$$

For, in such a case $(q_1, q_1(D))$ adds no information to the "knowledge base" $\{(q_2, q_2(D)), (q_3, q_3(D))\}$; we already know. That is, they will be impressed by your discovery that[2]:

$$\{\text{Beer}, \text{Diapers}\} = \{\text{Beer}\} \cup \{\text{Diapers}\} \textit{ and}$$
$$\{\text{Beer}, \text{Diapers}\}(D) = \{\text{Beer}\}(D) \times \{\text{Diapers}\}(D).$$

More in general, a generating set G such that for any $q \in Q \setminus G$ we have that:

- an expression for q in the elements of G,
- which allows us to "structurally" compute $q(D)$,

is a generating set that shows *structure* in D.

Clearly, there may be many such generating sets. To chose from this collection we use MDL. The expressions we have for the $q \in Q \setminus G$ are *encodings* of these queries. MDL then tells us that we should prefer generating sets that give the shortest encoding.

[2] All the more so given that *the* textbook example is that these purchases are *not* independent.

7 Typing, the Embarrassing Problem

We have now reached a point where there are many, exciting things to do algorithms, theorems, and so on. Things that will help me to understand what I did the past seven years and will – hopefully – give hints what I can do the next seven years. Some of these insights should come from experiments with real and artificial data. That is, an implementation of (some of) my ideas would be very useful.

Repetitio mater studiorum est, Doaitse has been practising this on me for the past $12\frac{1}{2}$ years. So I know there is only one programming language and its name is Haskell. Among its many advantages is its type system. Well-typed programs are correct – it seems that one cannot make algorithmic mistakes in Haskell[3]. But, how do I type my basic concepts? This is where I put my hope on Doaitse. After a long introduction, here are my questions.

First there is Q. Algorithmically – conceptually, if you want – Q is for me just some (finite) set of "abstract" objects which is equipped with some algebraic structure. That is, there are some operators on Q and there are some axioms that hold for these operators.

Ideally, I would like to have a type for this. That is, I would like to be able to write programs for which I do not have to choose whether Q is a monoid, a lattice, or something completely different. That is, I would like a type – perhaps I should call it a kind – that basically says Q is an abstract data type, i.e.,

$$Q :: ADT.$$

This would be useful for all those cases where I just need to "talk" about queries – statements like $q : Q$ – but don't need any form of computation. Next – still in the ideal world – I would be able to specialise (or instantiate) this very abstract kind to a slightly more concrete type or kind that states that Q is a lattice with all known axioms etcetera, i.e.,

$$Q :: ADT[Lat].$$

This would be useful for being able to use expressions like $q_1 \otimes q_2$. At this level we are still not specifying a lattice of "what"; programs at this level would work for any lattice. By the way, I would also like some form of inheritance here, programs that work for monoids should also work for groups.

At the next level, I would like to be able to specify that it is actually a lattice of sets over the elements $\{q_1, \ldots, q_n\}$ and "bind" the abstract operators $\otimes$ and $\oplus$ to $\cap$ and $\cup$, respectively, i.e.,

$$Q : ADT[Lat][set, n, \otimes \to \cap, \oplus \to \cap].$$

So, at this level we are can compute in Q with syntactic computations like $\{\text{Beer}, \text{Diapers}\} = \{\text{Beer}\} \cup \{\text{Diapers}\}$; syntactic since we don't evaluate anything on the data. Clearly, any program would require the third level to achieve meaningful computation. The point of the two higher levels is that they allow us to abstract away from concrete evaluations as much as possible – just as one does when one writes an algorithm. Phrased differently, it would give us optimal code re-use.

Unfortunately, I have no idea if this is possible in Haskell at all. If it isn't I would settle for being forced to skip the ADT-level. Which would mean that I would have to write different programs for, say, monoids and lattices. That is, I would settle for:

$$Q :: Lat$$

[3] This kind of unwarranted sarcasm of course only proves that I am not a very devout convert, yet.

and

$$Q :: Lat[set, n, \otimes \to \cap, \oplus \to \cap].$$

To some extend, this should be possible. After all, Haskell has, e.g., *monads*, which – to the layman – seem very close to monoids. What I don't know is if and how one can bind the natural transformations in the definition. That is, does it allow for abstract expressions like $q_1 \otimes q_2$ and/or symbolic evaluations like $\{q_1\} \cup \{q_2\} = \{q_1, q_2\}$?

Just as important. As far as I can see, monads are a primitive built into the language. How easy is it extend Haskell, e.g., with lattices? If asking for an unlimited supply of ADTs is a bit to much to ask for, having lattices are the minimum I would go for. This type of structure is ubiquitous among pattern languages and central to much I have done as well as much that I want to do.

So much for Q. Next on the agenda is the data itself. To discuss the type of our data we need to assume that we know how to type Q, let us denote that type by τ_Q. Given this type, it is actually quite straightforward to type my data. Since a data set is given by the queries in Q and their answers, data is simply a function that assigns an element in $[0, 1]$ to each $q \in Q$. That is,

$$D : \tau_Q \to [0, 1]$$

There is a constraint on these functions, they should respect the partial order on the lattice – did I already explain that I want both the partial order and the algebra for a lattice? – i.e.,

$$q_1 \preceq q_2 \to D(q_1) \geq D(q_2),$$

but it wouldn't be too much of a problem if the type system could not enforce this constraint – real and artificial data satisfy this constraint automatically.

Finally, we need to "structurally" compute $D(q)$ given some expression of type τ_Q for q. For example, if we have that $q = q_1 \otimes q_2$, then $q(D)$ would be given by an expression like:

$$D(q) = h(f(D(q_1)), f(D(q_2)))$$

Since we need both the expression and the data for the patterns from which the expression is build, this has type

$$\tau_q \times (\tau_Q \to [0, 1]) \to [0, 1]$$

So, it seems that given the type τ_Q the rest is straightforward. The big question is, thus, what would τ_Q be in Haskell?

8 Conclusion

Dear Doaitse, it has been a pleasure and an honour to work in the same department as you. Both the first few years and the last few years our offices happened to be close together. This enabled frequent conversations and discussions on topics as diverse as the intricate workings of university administration, the (lack of) policy for Computer Science in the Netherlands, the strange habits, behaviour and expectations of an outlandish tribe called students, but also on topics like investments, tax regulation, and the economy in general, as well as topics like "how to spend your holidays", "gastronomy vs honest affordable food, "wine vs fermented grape juice" and our differing views on what "the good life" is in general.

I have enjoyed all of this very much and I learned a lot. Thanks very much! I hope we will have many more such discussions, even though you went to live on (or over?) the edge of the civilized world. The best to you and Agnes.

References

1. Hand, D.J.: Pattern detection and discovery. In Hand, D.J., Adams, N.M., Bolton, R.J., eds.: Pattern Detection and Discovery, Proceedings of the ESF Exploratory Workshop. Volume 2447 of Lecture Notes in AI., Springer (2002) 1–12
2. Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: Proceedings of the ACM SIGMOD Conference on Management of Data. (1993) 207–216
3. Geng, L., Hamilton, H.J.: Interestingness measures for data mining: A survey. ACM Computing Surveys **38**(3) (2006)
4. McGarry, K.: A survey of interestingness measures for knowledge discovery. The Knowledge Engineering Review **20**(1) (2005) 39–61
5. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery. Data Mining and Knowledge Discovery **1**(3) (1997) 241–258
6. Knobbe, A.J., Ho, E.K.Y.: Pattern teams. In Fürnkranz, J., Scheffer, T., Spiliopoulou, M., eds.: Knowledge Discovery in Databases: PKDD 2006, 10th European Conference on Principles and Practice of Knowledge Discovery in Databases. Volume 4213 of Lecture Notes in Computer Science., Springer (2006) 577–584
7. Knobbe, A.J., Ho, E.K.Y.: Maximally informative k-itemsets and their efficient discovery, ACM (2006) 237–244
8. Geerts, F., Goethals, B., Mielikinen, T.: Tiling databases. In Suzuki, E., Arikawa, S., eds.: Discovery Science, Proceedings of the 7th International Conference. Volume 3245 of Lecture Notes in Computer Science., Springer (2004) 278–289
9. Siebes, A., Vreeken, J., van Leeuwen, M.: Item sets that compress. In Ghosh, J., Lambert, D., Skillicorn, D.B., Srivastava, J., eds.: Proceedings of the Sixth SIAM International Conference on Data Mining, SIAM (2006)
10. Raedt, L.D., Zimmermann, A.: Constraint-based pattern set mining. In: Proceedings of the Seventh SIAM International Conference on Data Mining, SIAM (2007)
11. Bringmann, B., Zimmermann, A.: The chosen few: On identifying valuable patterns. In: Proceedings of the 7th IEEE International Conference on Data Mining, IEEE Computer Society (2007) 63–72
12. Kontonasios, K.N., Bie, T.D.: An information-theoretic approach to finding informative noisy tiles in binary databases, SIAM (2010) 153–164
13. Webb, G.I.: Discovering significant patterns. Machine Learning **68** (2007) 1–33
14. Webb, G.I.: Self-sufficient itemsets: An approach to screening potentially interesting associations between items. ACM Transactions on Knowledge Discovery from Data **4**(1) (2010)
15. Grünwald, P.D.: Minimum description length tutorial. In Grünwald, P., Myung, I., eds.: Advances in Minimum Description Length. MIT Press (2005)
16. Grünwald, P.D.: the Minimum Description Length principle. MIT Press (2007)
17. Rissanen, J.: Information and Complexity in Statistical Modeling. Springer (2007)
18. Cover, T.M., Thomas, J.A.: Elements of Information Theory, second edition. Wiley (2006)

Haskell in the Large

Jurriaan Hage

J.Hage@uu.nl
Dept. of Computer Science, Utrecht University

Abstract. In this paper I describe my experiences during a few weeks of performance debugging (generated) Haskell and attribute grammar (AG) code.

1 Introduction

Haskell may be suitable for writing small programs elegantly, but as a law of Software Engineering (DeRemer's, [1]) tells us: what applies in the small does not (always) apply in the large. I have found that to some extent this is still very much true for Haskell, and much work will need to be done to overcome these problems. This is also good news, because it will keep us off the streets for just a bit longer.

In this paper I describe some of the adventures I had (in March/April 2012) while performance debugging a large Haskell implementation of the Asic bytecode instrumenter for Shockwave Flash that was developed as part of the Fittest project (for more information consultl http://www.facebook.com/FITTESTproject). It is a somewhat incomplete and anecdotal story, because I tried many things, and did not keep an extensive diary. Indeed, much of what I write here is reconstructed from reading old e-mails, and looking at svn commits.

Although performance has improved to a sufficiently high level, and there is help to be found for this task (both from books like Real World Haskell [2] and the more recent implementations of the attribute grammar system [3]), I found that performance debugging is very much an art, and by no means simple. Moreover, when you consider applications that work at a scale such as this, you quickly get the feeling that you are the first to do so, running into issues that nobody you talk to has ever run into.

2 Performance debugging the Asic compiler

The Asic compiler is a bytecode instrumenter for the ActionScript language (which compiles to Shockwave Flash (.swf) files). It was developed largely by Arie Middelkoop when he served as a scientific programmer within the Fittest project. There is a paper about this work [4], but what follows was actually work done after that paper had been published.

2.1 The issue at hand

The bytecode instrumenter had been tested on some realistic Flash executables, such as the FlexStore, but not on Sulake's Habbo Hotel (www.habbo.com) which is a much larger program, and which, at that time, was supposed to be our case study. In Shockwave Flash files, code, and other resources such as graphics, are organised into so called tags. A single code tag typically contains the implementation of one class. Therefore, code is usually spread over a large number of tags, each of them reasonably small. One reason why Habbo Hotel

was so difficult to deal with, is that the people of Sulake had merged the code of numerous tags into just three tags, in order to obtain a smaller bytecode file.

When compiled, the original Habbo has no more than 47,082 instructions in any of its tags, and only a small percentage goes over 10,000. The merged version we had to work with had a code tag with 1,479,958 instructions. Tags can be (are, in fact) transformed independently of each other. But our implementation was (and is) not so clever that it can divide a large tag into pieces and consider these pieces separately.

After Arie had left, we found that instrumenting the Habbo Hotel bytecode was way beyond our capabilities: a (sizable) instrumentation on the code led to memory consumption exceeding 32 GB. (We do not know how much it really was. All we know is that the program crashed on a 32 GB Linux machine.) Since we wanted an instrumentation of this kind to be performed on a "simple" MacBook with 4 GB memory, and we saw no reason why our instrumentation was supposed to take so much memory, I set myself the task of performance debugging the compiler. Since there was so much code, and pretty complicated code at that, this could only work effectively if I could tweak the performance by non-invasive, localized changes. This took some doing, but in the end, memory consumption was brought back back to 2.7 GB (saving quite a bit of running time as well). Below, I reconstruct (I have no exact records) some of what I, and others, did to attain this result.

2.2 The first few steps are easy

The first step is familiarisation with the application: there was a program called `asli` that depended on a library called `asil` (later renamed into `abci`). The library has 14,361 lines of hand-written Haskell and AG code, the program only contains 36 lines of code (in both cases, excluding empty lines and comments).

After looking at the top level functions, two things struck me. It seemed that somebody had tried to improve memory consumption by adding explicit strictness annotations here and there. However, when you parse a large file, putting a single strictness annotation (like ! or a seq) may not help much. What was needed here, in order to read in the whole file at once, was to deepseq the reading of the file. Second, there was a line of code that generated what amounted to a huge disassembly file of the bytecode. Although that may be useful for debugging purposes, this single line was the major reason for the huge memory consumption. After removing it, the program was down to 23 GB internal memory usage. This is a substantial improvement compared to not knowing at all what the actual memory consumption is. I also saw a reduction of memory consumption from 260 MB to 110 MB and running time from 60 to 40 seconds for the identify transformation on the smaller test program `test_apdf.swf`. After these easy wins the real work started.

3 The uphill slope

A trying part of this work was to have a suitable environment for running the (profiling) experiments. On the one hand I needed a machine with lots of memory, on the other I needed to be able to profile the code. This combination did not exist on any computer in our faculty accessible to me. In Haskell, for profiling to work, you need access to profiled versions of the base libraries. Unfortunately, the Linux server that enjoyed eight cores and 32 GB of internal memory did not have a GHC installed with the profiled base libraries. I asked system administration to install them, but when they finally told me they had done so, the work described here was already done. Instead, I chose a simpler route: first I verified that a similar memory problem arose for a smaller case study, and then I used that case

study to see what the effects of changes were. Another advantage of this way of working that the transformations, when they succeeded, took much less time.

But how to profile Haskell code? I had no experience whatsoever at the time, but found the very helpful Chapter 25 of the book Real World Haskell [2]. The protocol described there strongly resembles my approach, so I will not go into it deeply; you can read it there. I also needed access to the list of flags that GHC comes with, because Real World Haskell does not give you the whole story [5].

Essentially, you typically need to run your program *multiple* times with different parameters for what the profiler needs to track for you. The first step is typically to start with the `hc` flag:

```
asli ../InjectionSpec.txt test_apdf.swf Hex +RTS -hb -s
```

runs the `asli` program, with a specified transformation (from `../InjectionSpec.txt`), on the Shockwave Flash file `test_apdf.swf`, resulting in the transformed output `Hex`. The profiling is turned on by passing `-hb` and `-s` to the run time system, which is why they follow `+RTS`. The flag `hb` (b stands for *biographical*) says that you want to know, over time, what amount of memory is lag, use, drag and void. The latter describes memory that is never used (which may, for example, be due to some strictness flags being in the wrong place). *Drag* describes memory from its last use to its final garbage collection. These two types are the ones to look at first. The *lag* kind of memory is memory that is allocated long before its use. This can be problematic too, because the bottleneck of memory, and what I was trying to reduce here, is the maximum amount of memory in use across execution (reduction of memory *volume* is also nice, but that tends to follow the reduction of the maximum memory usage as a matter of course). *Use* is the time between the first and last use of a memory cell. The `hb` profile quickly gives you an idea of whether quick gains can be made. I used it too, but not overly much (discovering its use fairly late in the game).

The flags I employed the most were `hy`, `hc` and `hr`. The former informs you how much of your memory is occupied by each of the constructors in your program. Here, you may find out for example that lists taken an inordinate amount of memory, and if you happen to be working with long strings, you may want to switch to `Text`.[1]

The second important flag, `hc`, tells you which function created the need for a particular memory cell. The third flag, `hr`, tells you which part of the program *retains* how much of the memory. In this particular case, many of these were semantic functions generated by the AG compiler [3]. After consulting Jeroen Bransen and Arie Middelkoop, I learned about some recent innovations to the system, including a flag called `kennedywarren`. The purpose of this flag to decrease memory consumption by statically deciding on the order in which attributes are evaluated. This certainly helped, but not enough to attain my goal.

The progression of results is illustrated by the sequence of profiles made for an empty transformation (which reads and analyzes but does not transform the bytecode) in Figure 1 and 2. The full injection takes more time and memory, but generally follows this same pattern. In Figure 1(a) we can see a large spike between the 20 and 35 second mark. This is due to the fact that the disassembly file will be written out shortly. This spike is absent in the other two images. In all images we have depicted the retainer profile. In the legend on the side it can be seen that many retainers are functions starting with `sem_`; this implies that these are semantic functions generated by the attribute grammar system. There are many more such pictures to be displayed here. However, these three should be enough to

[1] We did not need that here, because we worked with binary files. But in another performance debugging session (on the Haslog library that we also use within Fittest) we gained a huge win by switching to the `Attoparsec` parser library, and by switching from `String` to `Text`.

convey the general flavour of these profile images. Note that these pictures are still very much in the early stages. Later, we got the memory consumption for a *full* instrumentation of this file below 100 MB.

3.1 The waiving of flags

To close this gap, Arthur Baars and I sat down at the Utrecht Hackathon for an extensive profiling session. Here we found that what seemed to be the bottleneck for the computation: a small piece Haskell code that was generated from AG code, only twenty lines or so.

By hand, the code could be made to evaluate more strictly, thereby resolving the last of the memory bottlenecks. The problem was that I had to tweak the generated code, and had, as yet, no way of tweaking the AG code in a way that would have that effect on the generated code. Arthur Baars told me that he found some flags in the AG system that had the effect I was looking for. Indeed, after adding the `bangpats` and `optimize` to each of the AG modules, almost completely solved the remaining memory issues. Almost completely? Yes. Almost.

Maybe you have once tried to get the air out an inflatable mattress by putting yourself squarely on top if it. If you do not do so with care, often the effect is that the air is not forced out, but to a different part of the mattress. This is what seemed to happen here as well: a new bulge cropped up where there had not been one.

On a whim, I decided to omit the `bangpat` flag for the module that seemed to retain the memory in this spike, and so managed to get rid of this final memory hiccup in the profile. Running this version for the full injection on a particular version of the Habbo case study gave a maximum memory usage of 1.8 GB. Of course, at this point the application still uses quite a bit of memory, but at least the profile is now pretty much horizontal.

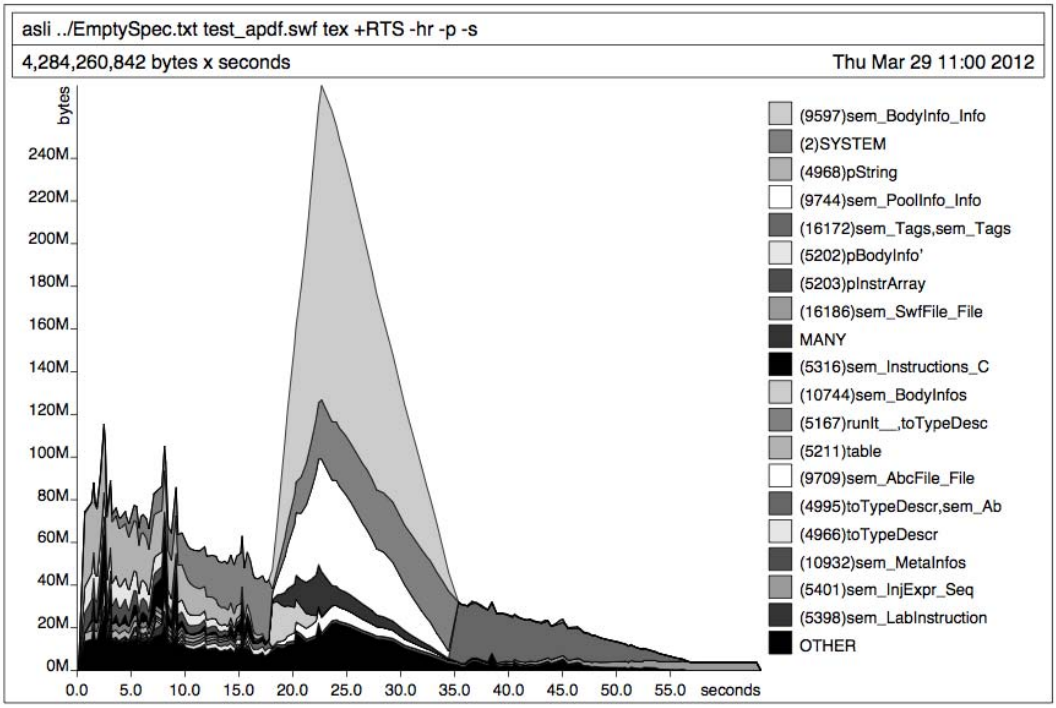
3.2 Tune in, tune out

Now that we seem to have the code under control, it was time for some fine-tuning. A lesson well-learnt is that if you have memory to spare, you can pass as argument `-H2000M` to the runtime, because then the program will "waste" much less time on garbage collection. In the case of full injection on habbo_secure, the running time goes down from 975 seconds to just over 300. Of course, the memory footprint increases, because the garbage collector is rarely called, and therefore do es not economically compact. This experiment was repeated on a Linux server that has many cores and much memory. The base timings for full injection without a `-K` option are 1526.72s When passing `-H4000M` to the execution, this becomes 658 sec. Surprisingly, when you choose `-H2000M` on this machine, the running time becomes smaller, 546s. So time is a function of the H option, but it is not monotonically decreasing with memory increase. It seems that anything between `1900M` and `2500M` gives comparable results in this case, and they get worse outside this interval.

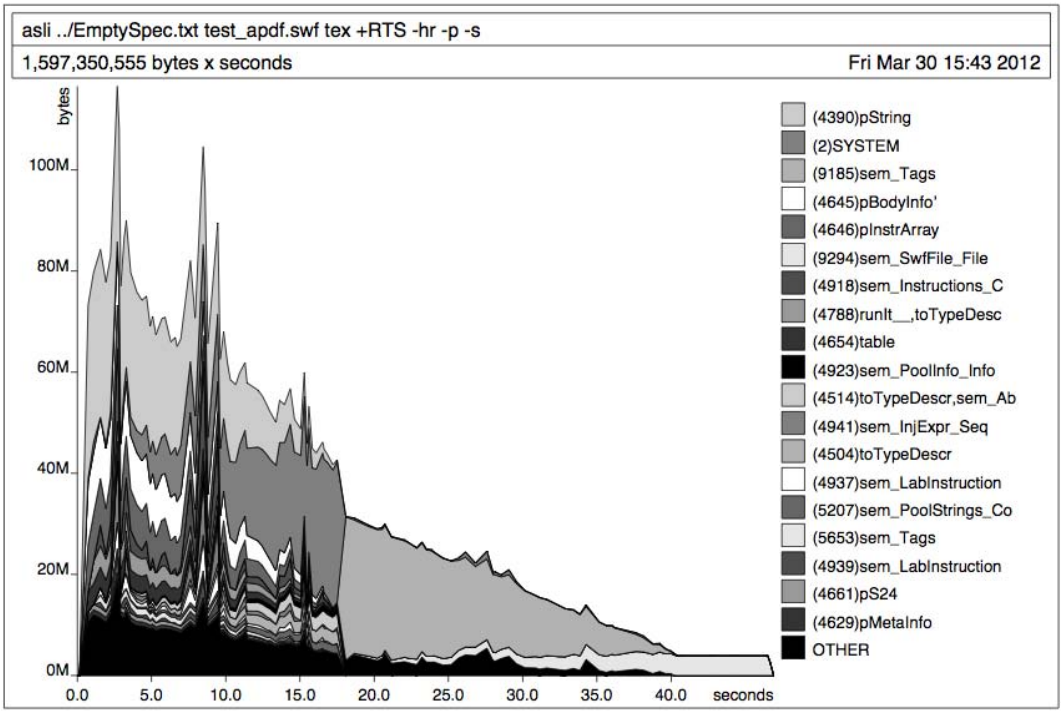
Note that setting aside `4000M` for computation implies that when gc starts, the process will need quite a bit more than that. In fact, it used about double what I set. This implies that setting a low value is better if that does not negatively affect run time too much. Using threads and parallel GC's only served to increase running time, sometimes by inordinately much. In some cases, I could do the transform faster by hand.

3.3 Some further considerations

Suppose you set yourself the task of bringing the running time of some application down to x milliseconds. It is then important to remember that the resource consumption of a program

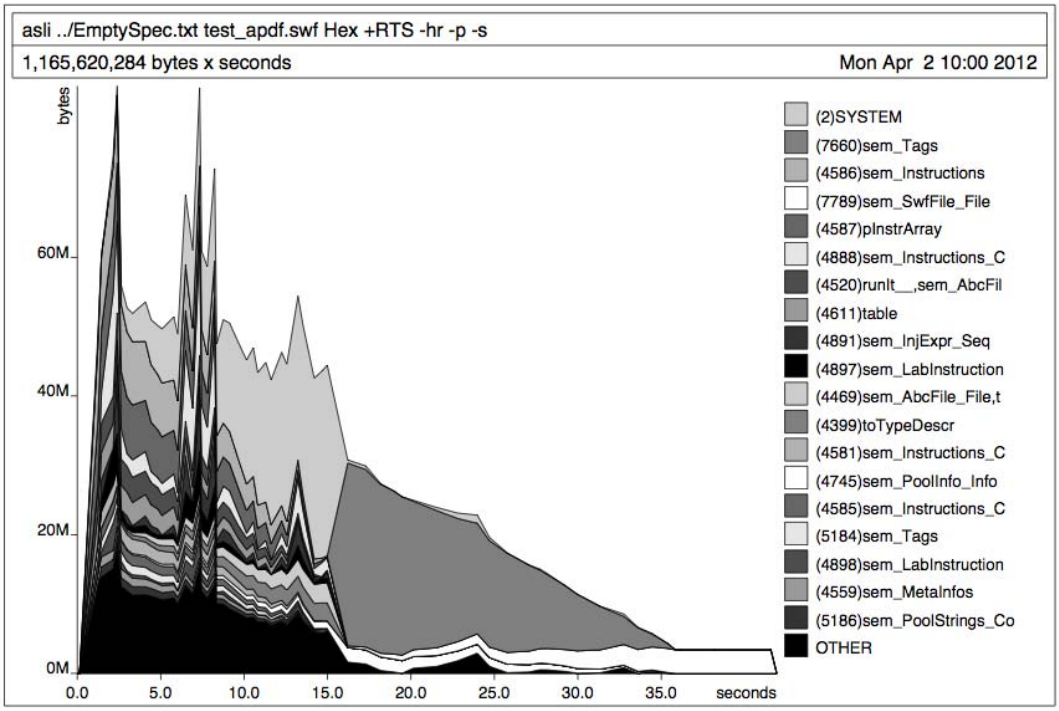


(a)



(b)

Fig. 1. The first and second profile



(a)

Fig. 2. The third profile

that has been compiled with profiling turned on is much higher than of a program for which this is not the case, *even if you do not ask the profiled program to actually deliver a profile.* This overhead is substantial, so it may be wise to, once in a while, compile a clean version without profiling and debug information to see what the situation really is like. According to the GHC user guide, compiling with profiling on gives code that has about 30 percent memory overhead.

About garbage collection, the GHC User Guide has the following to say [5]:

> Garbage collection requires more memory than the actual residency. The factor depends on the kind of garbage collection algorithm in use: a major GC in the standard generation copying collector will usually require 3L bytes of memory, where L is the amount of live data. This is because by default (see the +RTS -F option) we allow the old generation to grow to twice its size (2L) before collecting it, and we require additionally L bytes to copy the live data into. When using compacting collection (see the +RTS -c option), this is reduced to 2L, and can further be reduced by tweaking the -F option. Also add the size of the allocation area (currently a fixed 512Kb).

To this information I can add my own findings during this experiment that the overhead for the copying collector seems close to 2L, while the compacting garbage collector scores worse (both in terms of running time, and in terms of memory consumption) than the copying garbage collector. The former was no surprise, but the second came quite as a surprise to me. I also found during these experiments with the compacting collector that it sometimes

simply crashed the application. I expect that this is because not many people still use it, and maintainance of this code is not a high priority.

It will interest at least one reader that we made good use of Doaitse Swierstra's error correcting parsers while building the bytecode instrumenter [3]. The reason is this: Arie quickly found that the bytecode specification and actual bytecode as it was understood by the Flash runtime diverged. In order to debug the specification (since we *had* to deal with actual bytecode generated by the ActionScript compilers), we employed the error correcting parsers to discover what the specs should have said, thereby obtaining a more precise (and even executable) specification of said specs.

When the Asic compiler was performance debugged, and we ran it on the Habbo Hotel testcase, we ran into another problem. Flash code needs to maintain an invariant on the size of the stack: for any given instruction, the height of the stack should always be the same when you execute that instruction, independent of how the statement is reached. To verify that our implementation did not break that invariant, Arie had implemented a dataflow analysis. It then turned out that after instrumentation Asic generated messages like:

```
param analysis at method 6 and instruction 726: warning: different
stack usages on incoming CFG paths: stack[1] { lbls: 703} : ...
and  stack[0] ...
```

The question was then: is our instrumenter wrong, or is something else the matter. After finding that running an identity transform on the case study revealed the same problem, I concluded that the problem was already in the original code. But why did the Flash runtime not scream bloody murder when we ran it? Then I remembered that Sulake performs extensive obfuscations on their code, merging tags, changing identifier names. Maybe the broken invariants are due to something they did to their code, but in a way that goes unnoticed when you execute it. With all these ingredients, the solution was simple: Sulake introduced *dead* code that when executed would break the invariant. Our analysis was not precise enough to discover that the dead code was dead, which led to our failure to decide that the invariant was in fact not broken. To fix this, we simply allowed invariant checking to be turned off, because it was mostly useful in the development phases. A pleasant side effect is that this again reduced the running time of our instrumenter.

4 Closing words

It took me quite a bit of time to decide what to actually write about. I could have written about work Stefan, I and others have done on type and effect systems ([6, 7]), or about the topic of type error diagnosis that you initiated at Utrecht with the hiring of Bastiaan Heeren [8–12] that I am currently actively pursuing again. I could also have finished that (strongly typed) implementation of switching classes, seeing how the efficiency of counting the number of trees (or otherwise) [13–17] in a switching class would measure up against my C implementation (that it is faster than my earlier Scheme implementation can be considered a given). Instead, I opted for a topic that, I believe, is more to your heart, involving both Haskell and your own AG system.

Over the years, I have learned a lot from you. And although I was not always happy with your decisions, I have always found your loyalty and sense of duty towards the people in our group, to the department, and to science, admirable. I hope you enjoy your well-earned rest up there in the north, far away from people who put dead links on websites, who think that a lecturer need not sign for a passing grade of a student, or who believe that you publish papers, not results. As you well know, la vida en la ciudad es muy complicada. I just hope

life in the village of Tynaarlo will not be too sedate.

Acknowledgements I gratefully acknowledge the assistance of Arie Middelkoop and Arthur Baars in the process of performance debugging.

References

1. DeRemer, F., Kron, H.: Programming in-the-large versus programming-in-the-small. In: Proc. Int. Conf. Reliable Software, IEEE Computer Society Press (1975) 114 – 121
2. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. 1st edn. O'Reilly Media, Inc. (2008)
3. Swierstra, S.D., Rodriguez, A., Middelkoop, A., Baars, A.I., et al., A.L.: The Haskell Utrecht Tools (hut) http://www.cs.uu.nl/wiki/HUT/WebHome.
4. Middelkoop, A., Elyasov, A.B., Prasetya, W.: Functional instrumentation of actionscript programs with asil. In Gill, A., Hage, J., eds.: Implementation and Application of Functional Languages - 23rd International Symposium, IFL 2011, Lawrence, KS, USA, October 3-5, 2011, Revised Selected Papers. Volume 7257 of Lecture Notes in Computer Science., Springer (2012) 1–16
5. The GHC Team: The Glorious Glasgow Haskell Compilation system user's guide, version 7.0.1
6. Holdermans, S., Hage, J.: Making "stricterness" more relevant. Higher-Order and Symbolic Computation **23** (2011) 315–335
7. Holdermans, S., Hage, J.: Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In: Proceedings of the 15th ACM SIGPLAN 2010 International Conference on Functional Programming (ICFP '10), ACM Press (2010) 63–74
8. Heeren, B.: Top Quality Type Error Messages. PhD thesis, Universiteit Utrecht, The Netherlands (2005) http://www.cs.uu.nl/people/bastiaan/phdthesis.
9. Hage, J., Heeren, B.: Heuristics for type error discovery and recovery. In Horváth, Z., Zsók, V., Butterfield, A., eds.: Implementation of Functional Languages – IFL 2006. Volume 4449., Heidelberg, Springer Verlag (2007) 199 – 216
10. Hage, J., Heeren, B.: Strategies for solving constraints in type and effect systems. Electronic Notes in Theoretical Computer Science **236** (2009) 163 – 183 Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008).
11. el Boustani, N., Hage, J.: Improving type error messages for generic java. Higher-Order and Symbolic Computation **24**(1) (2012) 3–39 10.1007/s10990-011-9070-3.
12. Weijers, J., Hage, J., Holdermans, S.: Security type error diagnosis for higher-order, polymorphic languages. In: Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation. PEPM '13, New York, NY, USA, ACM (2013) 3–12
13. Hage, J., Harju, T.: A characterization of acyclic switching classes using forbidden subgraphs. SIAM Journal on Discrete Mathematics **18**(1) (2004) 159 – 176
14. Hage, J., Harju, T., Welzl, E.: Euler graphs, triangle-free graphs and bipartite graphs in switching classes. In Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G., eds.: Graph Transformation, First Int. Conf, ICGT 2002. Volume 2505 of Lecture Notes in Computer Science., Berlin, Springer Verlag (2002) 48–60
15. Hage, J., Harju, T.: The size of switching classes with skew gains. Discrete Math. **215** (2000) 81 – 92
16. Hage, J.: The membership problem for switching classes with skew gains. Fundamenta Informaticae **39**(4) (1999) 375–387
17. Hage, J., Harju, T.: Acyclicity of switching classes. European J. Combin. **19** (1998) 321–327

Verstandige Zaken of Het Ontleden van Haskell

Hans L. Bodlaender

h.l.bodlaender@uu.nl
Department of Computer Science, Utrecht University

Samenvatting In het voorjaar van 2011, bij gesprekken over de voorbereiding van [1], zei Doaitse tegen me, dat als ik een stukje voor een 'Liber' voor hem zou schrijven, ik *zou moeten schrijven dat ik hoopte dat hij zich zou bekeren.* Bij deze! In dit stuk worden verbanden gelegd tussen Doaitse's favoriete programmeertaal en de Bijbel, en verder, . . . , lees zelf maar.

1 Introductie

In de ongeveer dertig jaar dat Doaitse en ik werkten bij Informatica in Utrecht hebben we over veel verschillende dingen gepraat. Over sommige dingen waren we het hartgrondig eens, over andere dingen niet. Dit stukje heeft deels met de tweede categorie te maken, maar ook met het ontleden van Haskell, maar anders dan bijvoorbeeld in [2].

2 Het Parsen van Haskell

De programmeertaal Haskell is genoemd naar de logicus Haskell Brooks Curry (1900 – 1982), wiens wetenschappelijk werk van onschatbaar belang was, onder andere voor de ontwikkeling van functionele programmeertalen. Haskell Curry was promovendus van Hilbert; hij heeft ook een tijd in Nederland gewerkt: van 1966 tot 1970 aan de UvA. De techniek *currying* is naar hem genoemd.

In dit stuk wil ik kijken naar de afkomst en betekenis van het woord Haskell. Ik ga hierbij uit van de verklaring die het vaakst voorkwam in mijn zoektocht. Er bestaan ook andere verklaringen.

Er zijn niet veel mensen met de voornaam *Haskell*. De oorsprong van de naam is Hebreeuws. In dit stukje wil ik het woord *ontleden*, of met een goed gekozen Anglicisme *parsen*.

Haskell in het Hebreeuwse alfabet geschreven wordt: הַשְׂכֵּל. In deze spelling worden de klinkers met behulp van zgn. *nikkud* aangegeven. Deze nikkud zijn kleine stipjes en streepjes die rond de medeklinkers gezet worden. Vaak worden de nikkud (klinkers) weggelaten; in ons geval krijgen we de volgende versie van het woord: השכל. Omdat we in dit stukje dit woord nauwkeurig gaan bekijken, hieronder nogmaals de twee versies , maar nu in groter font.

הַשְׂכֵּל en השכל

Hebreeuws wordt gelezen van rechts naar links. De achtereenvolgende letters zijn:

- De meest rechtse letter ה is de hee (of hé), de vijfde letter uit het Hebreews alfabet.
- Onder de hee zien we een streepje. Deze nikkud geeft de klinker a aan. Klinkers worden aangegeven bij de voorafgaande medeklinker.
- De een-na-meest rechtse letter שׂ is de shien of shin, een 'zachte' s. Een bekend woord dat met de shien begint is שָׁלוֹם (shalom).

– In de shien staat een stipje. Dit wordt een dagesh genoemd; deze beinvloeden de uitspraak of het splitsen in lettergrepen.
– Onder de shien staan twee stippen onder elkaar, een Sh'wa. Deze kunnen verschillende betekenissen hebben maar geven nu de afwezigheid van een klinker weer.
– De derde letter van rechts ך is een kaph (k).
– Onder de kaph staan twee stippen naast elkaar; deze nikkud geeft weer een klinker aan, een *e*. De klank is ongeveer als in woorden als *bed* en *fel*. De naam van deze nikkud is *zeire*.
– In de kaph staat weer een stipje, een dagesh, die de klank van de k harder maakt.
– De laatste letter, de meest linkse, is een lamed ל (l).

Van Syntax naar Semantiek Een enkel Hebreeuws woord kan soms in de vertaling een hele zin zijn: een enkele medeklinker (met bijbehorende klinker) kan bijvoorbeeld een voorzetsel of voegwoord aan de zin toevoegen. Haskell הַשֵּׂכֶל is uit twee delen opgebouwd:

– De vervoeging הַ (Ha – de letter hee met klinker), en
– De stam שֵׂכֶל (skel).

Een ה (he) als begin van een woord kan een zin vragend maken en kan een bepalend lidwoord (de, het) aanduiden; in ons geval is dit een bepalend lidwoord.

Dat geeft als stam van het woord שכל. Het zelfstandige naamwoord wat bijna identiek is, is שׂכל: we zien dat de stip op de shien naar links is verschoven. Hiermee wordt de s scherper uitgesproken, meer als de s in *stop*. Een Joodse collega van me gaf als mogelijke verklaring van de verschuiving een invloed van het griekse *skola* (school).

Het woord שׂכל is een zelfstandig naamwoord dat **inzicht** of **verstand** betekent. Een vertalen van Haskell in het Nederlands zou dus *Het inzicht* of *Het verstand* kunnen luiden, inclusief het lidwoord.

Andere Verklaringen Een andere oorsprong van de naam Haskell kan een verbastering van Ezechiël zijn, de naam van een profeet en een naar hem genoemd Bijbelboek. Ook is Haskell een familienaam uit Wales[1], maar de naam Haskell met deze achtergrond lijkt vooral als *achternaam* voor te komen.

3 Haskell in de Joodse Liturgie en in de Bijbel

In zijn vorm met lidwoord vond ik Haskell niet in de Bijbel. Wel komt deze vorm voor in het Joodse Achttiengebed; zonder lidwoord zien we het woord een aantal keren in de Bijbel staan, onder andere in het boek Spreuken.

Het Joodse Achttiengebed Het Achttiengebed (Amida) is een belangrijk gebed uit de Joodse liturgie en wordt in veel synagogen meerdere malen per dag gebeden. Het gebed heet het Achttiengebed omdat het oorspronkelijk uit achttien delen bestond; tegenwoordig bestaat het uit negentien onderdelen. Het vierde onderdeel is een gebed voor kennis en inzicht; de tweede van drie regels luidt:

חננו מאתך דעה בינה והשׂכל

De tekst hierboven kan als volgt vertaald worden.

Begunstig ons — van U — kennis — verstand — en het inzicht

[1] http://www.houseofnames.com/haskell-family-crest

De vijf woorden uit de Hebreeuwse tekst moet je van rechts naar links lezen; in de Nederlandse vertaling staan ze opgesomd van links naar rechts. In het laatste woord van de zin (helemaal links in het Hebreeuws) zien we het woord Haskell, de ו geeft het voegwoord *en* aan.

Het Strong-nummer van (Ha)skell De Amerikaanse Bijbelgeleerde James Strong (1822 – 1894) is bekend om zijn concordantie van de Bijbel. Hij heeft ook een nummering gemaakt van de Hebreeuwse en Griekse woorden die in de Bijbel voorkomen, de zgn. Strong-nummers.

Er zijn verschillende woorden in Strongs concordantie met als medeklinkers שׂכל. De woorden verschillen van betekenis en beklinkering. Ik ga er van uit dat het woord waarop Haskell gebaseerd is het zelfstandige naamwoord met Strong-nummer 7922 is: sekel oftewel שֵׂכֶל. Als vertaling geeft Strong [5]: *understanding, wisdom, discretion.* Verwand lijkt Strong 7919: het woord שָׂכַל (sakal) is het werkwoord met o.a. de betekenissen *succes hebben, inzicht hebben.*

Voorkomens in de Bijbel Het woord sekel שֵׂכֶל komt 22 keer voor in de Bijbel. (De afkortingen hieronder geven aan welke Bijbelvertaling ik heb gebruikt: NBV = Nederlandse Bijbel Vertaling, GNB = Groot Nieuws Bijbel.)

Bekend is bijvoorbeeld de tekst in Psalm 111:10: *De vreze des HEREN is het begin der wijsheid, een goed inzicht hebben allen die ze betrachten. Zijn lof houdt eeuwig stand.* (NBV) Hierbij is *inzicht* de vertaling van שֵׂכֶל.

In het Bijbelboek Spreuken komt het woord vaak voor. Sommige van deze Spreuken gaan over wijsheid en dwaasheid. Toegepast op het hedendaagse academisch onderwijs zijn ze nog steeds actueel.

Spreuken 12:8 *Men prijst een mens naar de maat van zijn verstand, een warhoofd wordt geminacht.* (NBV)

Spreuken 16:22 *Inzicht is een bron van leven, dwazen worden met dwaasheid gestraft.* (NBV) Of, met de woorden van de King James vertaling: *Understanding is a wellspring of life unto him that hath it: but the instruction of fools is folly.*

Spreuken 23: 9 *Verspil je woorden niet aan een dwaas, want hij waardeert toch niet wat je zegt.* (GNB) In de King James vertaling klinkt het, vind ik, opnieuw nog mooier: *Speak not in the ears of a fool: for he will despise the wisdom of thy words.*

4 Wijsheidsliteratuur, en een Slotwoord

Spreuken behoort tot de Joodse Wijsheidsliteratuur, net als de boeken Job en Prediker.

Prediker analyseert de wereld om zich heen (wat er is *onder de zon*[2]) op een messcherpe manier, en komt tot de conclusie veel slechts *IJdelheid der ijdelheden* (ijdelheid is de vertaling van הבל, hebel, letterlijk *damp*) is: weinig zinvol en tot niets leidend. Ook dat is een observatie die, helaas, niet aan actualiteit lijkt te hebben ingeboet. Misschien niet in de termen van Prediker, maar veel van deze observaties zitten dicht bij wat ik aan het begin van dit stukje de eerste categorie genoemd had.

Ik ben niet de eerste die de duidelijke overeenkomsten tussen Prediker en Doaitse ziet: beide weten op een overtuigende wijze de vinger te leggen op het gebrek aan doel of doeltreffendheid van wat er 'onder de zon' gebeurd en ze doen dat met woorden die op een unieke manier gekozen zijn, en laten daarbij een grote hoeveelheid scherp inzicht en wijsheid zien.

[2] "Onder de zon" is een uitdrukking in Prediker die vertaald kan worden met 'op deze wereld'.

En, terwijl de woorden lang niet altijd bij iedereen instemming vinden, is wel altijd de grote originaliteit en logica onmiskenbaar.

Veel van het boek Prediker gaat over wat niet zinvol is. Maar, naast alles wat gezien wordt als chaos en ledigheid, komt Prediker met een doeltreffend advies: (Prediker 9: 7-9, NBV)

> *Dus eet je brood met vreugde, drink met een vrolijk hart je wijn. God ziet alles wat je doet allang met welbehagen aan. Draag altijd vrolijke kleren, kies een feestelijke geur. Geniet van het leven met de vrouw die je bemint. Geniet op alle dagen van je leven, die God je heeft gegeven. Het bestaan is leeg en vluchtig en je zwoegt en zwoegt onder de zon, dus geniet op elke dag. Het is het loon dat God je heeft gegeven.*

Dat genieten van wat God ons geeft, dat kan op veel manieren: samen met je geliefde een film kijken, door een mooi gebied fietsen (bijvoorbeeld Amelisweerd) of wandelen (bijvoorbeeld een mooi Drents beekdal), een mooi algoritme verzinnen, een prachtig technisch boek lezen (ik zou zelf *The Art of Computer Programming* kiezen, maar Doaitse kiest misschien wel wat anders) of een mooie roman (of een interessante roman, bijvoorbeeld [6]), en zo voorts.

Doaitse, ik wens je toe dat je nog heel veel zult genieten van al die mooie dingen.

Ik heb nu al ongeveer dertig jaar mogen genieten van bijzondere gesprekken, je enorme originaliteit, inzichten en hartelijkheid, etc. Heel hartelijk dank daarvoor!

Enkele Bronnen Deel van de informatie is afkomstig van internetbronnen, o.a.: Wikipedia, http://www.my-hebrew-name.com/haskell-haskell-9872.html en http://www.blueletterbible.org. Voor informatie over het Hebreeuwse alfabet baseerde ik me o.a. op [3,4]. Dank aan Johann Makowsky voor aanvullende informatie. Voor weergave van het Hebreeuws gebruikte ik het LaTeX-pakket cjhebrew.

Referenties

1. H. L. Bodlaender and E. J. van Leeuwen, editors. *Facetten van Jan. Liber Amicorum voor Jan van Leeuwen*, 2011.
2. A. Dijkstra, J. Fokker, and S. D. Swierstra. The structure of the essential Haskell compiler, or coping with compiler complexity. In *Proceedings of the 19th International Workshop on Implementation and Application of Functional Languages, IFL 2007*, pages 57–74, 2007.
3. R. L. Modeth, J.Strijker, and R. S. van der Giessen. *Hebreeuws in Zes Dagen*. Uitgeverij Oriënt Press, 2008.
4. M. Paul, G. van den Brink, and J. C. Bette. *Studiebijbel Oude Testament. Psalmen II, Spreuken, Prediker*. Centrum voor Bijbelonderzoek, 2011.
5. J. Strong. *The Strongest Strong. Exhaustive Concordance of the Bible*. Zondervan, 2001. Revision by J. Kohlenberger III and J. A. Swanson.
6. J. J. Voskuil. *Het A.P.Beerta-instituut*. Het Bureau. Oorschot, 2003.

Incremental Evaluation, Again

Matthijs Kuiper

matthijs@data-x.nl
Data eXcellence, Nieuwegein, The Netherlands

Abstract. The performance of data transformation applications is improved through the application of techniques for incremental evaluation. Some of the techniques were inspired by work on incremental attribute evaluation.

1 Data conversion systems

Data conversion systems transform data from a source system to a form that can be loaded into and processed by a target system.

Most data conversion systems take 6 to 12 months of development and test before they are put into production. The systems we develop are executed only once in production (or twice if something unexpectedly goes wrong). The testing is extensive to assure that the only and final conversion run is successful.

During development data processing components are built. Besides that special test components are developed that verify that the data processing components produce the correct data. Although technically not exactly correct one could say that the data processing components are built twice (by different developers) and that testing consists of comparing their outputs for equality.

Data processing components have as input one or more database tables and store their result in a database table. A table produced by a component can be used as input for other components. In a sense the conversion systems are a variant of ETL systems (Extract, Transform, Load): they are LTTTE systems: source data is loaded and then transformed by a chain of data processing components and the final results are extracted. Some chains contain more than 20 Transforming components.

The data conversion systems we consider are mostly used for processing data from core systems of financial institutions. The owners of these systems require that data conversion systems produce extensive audit trails. Each conversion system therefore maintains a database with details about all executed components, known as the data conversion administration (DCA).

2 Testing Data Conversion Systems (DCS)

Developers test their components by running the DCS on a small representative test set. In addition, at the end of the day a version of the conversion system is build that integrates all components of the system. The resulting version of the software is run on the test set and, if all is fine, a nightly run on a complete data set is performed.

During the day each programmer performs several runs on the same test set. The execution time of such a test run varies between fifteen and thirty minutes. As a consequence waiting for the results of test runs takes a significant amount of time.

Most test runs differ only slightly from the test runs preceding it; sometimes only one or two components have changed and in almost all cases the test sets are equal. So the input

data stays the same but the software that processes that data changes slightly between test runs.

3 Approach

To reduce the time of test runs we keep the data tables of the previous run and skip all convertor and load invocations preceding a changed component. A changed component is executed as are all subsequent components. Before a convertor is executed its target table is emptied with a truncate operation.

Our approach requires that the order of invocations is known (to determine preceding and subsequent in the previous paragraph) and that the target tables of executed convertors are known. This information is available in the audit trail of the basis run. The basis run produces an overview of invocations, including their start and end times. For a convertor invocation the latest modification time of the convertor is stored; for a load the LMT of the file is stored. The incremental run starts with an analysis to determine the changed components and the components that must be executed. This information is stored in the DCA. Next the scheduler is started and the components are executed. Each component decides if it can skip work, based on the information in the DCA and the status of the run.

The analyser has information about the tables that are being filled by a convertor, but it lacks knowledge about the tables being used by a convertor. The analyser therefore makes the pessimistic assumption that a convertor uses all tables that were filled preceding its execution. This implies that once a changed convertor is executed all convertors following it must also be executed. We rejected the idea to let developers specify which tables are used in a convertor, since we cannot trust the correctness of such a specification.

The analyser applies the following rules.

1. If a convertor has changed, is removed or new then its target table must be emptied and all convertors that fill that table must be rerun.
2. If a convertor is run then all tables that it fills must be emptied.
3. If a convertor is run then all convertors that follow it must be run.

Note that tables can be filled by more than one convertor. If one of them has changed then the table is emptied and thus the other ones must also be reexecuted (if they follow the changed convertor) or they must be reloaded (if they precede the changed convertor)

4 Examples

This section contains some examples of the analysis being performed on the convertors in the incremental run. The first example in Figure 1 is a simple one where a convertor is changed. We use the convention that convertor Ci produces records for table Ti; Ei stands for emptying table Ti. In the example the second run, when performed from scratch, would execute all convertors. The analyser determines that T1 en T2 can remain untouched and that T3 and T4 are effected by the change to C3.

The second example, in Figure 2, shows that changing a convertor can result in the reexecution of convertors that precede the changed convertor. In this example convertors C1a and C1b fill the same table, T1. Convertor C2 has changed and therefore its target table must be recomputed. Since C1b follows C2' and might use data produced by C2 it must be run again. So T1 will be emptied with the effect that C1a must be executed as well.

Basic run	C1;C2;C3;C4
Second run	C1;C2;**C3'**;C4
Affected tables	T3;T4
Incremental run	E3;E4;C3';C4

Fig. 1. Changing a convertor

Basic run	C1a;C2;C1b;
Second run	C1a;**C2'**;C1b
Affected tables	T1;T2
Incremental run	E1;E2;C1a;C2';C1b

Fig. 2. Changing a convertor with retroactive effect

5 Effects of incremental execution

Figure 3 gives an overview of the effect of incremental evaluation. The execution time of the incremental run is approximately 40% of that of the full run. The convertors executed in the incremental run were convertors that filled temporary tables that were later used as input to other convertors. These tables were not really temporary. At the start of the incremental run all temporary tables are removed from the database and they must be created and filled again in the incremental run. The test runs suffer from what might be called an administrative burden: runs perform extensive logging and produce many reports as an audit trail. Not all these are needed in all test runs and runs can be configured in such a way that they skip the production of reports. Developers are however reluctant to do this, they want as little difference as possible between the software in test runs and in runs on the full data set and they therefore prefer to produce all reports in all runs.

	basis	*incremental*	*incremental run as percentage of basis run*
Records produced by convertors	860.001	237.998	27,7%
Sum of execution times of components (seconds)	1531	663	43,3%
Execution time of run (minutes)	19:36	7:43	39,4%

Fig. 3. Effects of incremental execution

6 Final remarks

The approach sketched in this paper was inspired by the work on incremental attribute evaluation of Pennnings and Saraiva. The time savings of 60% that was achieved by incremental execution is substantial. Some further optimizations were needed to reduce the time of test runs to a desired level.

This paper presented the main idea and has ignored many practical complications, such as convertors that fill more than one table, components that crash, parallel execution of components and runs that are aborted.

The idea of a cache, i.e. using something that is available instead of recomputing it, such as the filled tables in a database as we described here, seems in general a good idea. Of course one must somehow know what is available and, in this case, how it was produced. We were fortunate that the audit trail could be used as a description of what was available for reuse.

The earlier work on caching that inspired our approach was supervised by Doaitse and so he has indirectly and unknowingly contributed to the work in this paper. We have used the existing scientific work as a cache of ideas to solve a practical performance problem.

Traversals with Class

Bastiaan Heeren

bastiaan.heeren@ou.nl
School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands

Abstract. This paper describes a set of combinators for specifying traversal strategies. These combinators are based on Haskell 98 type classes (without extensions) for iterating over sequences and navigating in trees. Since traversal strategies are commonly found in procedures for solving stepwise exercises in tutoring systems, we show how the traversal combinators can be used in a strategy language. Since feedback is generated automatically from a rewriting strategy, we need to have intermediate terms and allow non-determinism in our strategies.

Ik ken Doaitse nu zo'n 15 jaar: eerst als student Informatica, later als AIO en junior docent aan de Universiteit Utrecht in de vakgroep van Doaitse. De laatste jaren werk ik voor de Open Universiteit. Dankzij mijn werkkamer bij de Utrechtse Software Technology groep is het contact er niet minder op geworden.

In 1999 zijn we samen naar Bolivia afgereisd waar ik zou gaan werken aan mijn afstudeerscriptie. Typerend voor Doaitse was dat we na onze aankomst om 7 uur 's ochtends, na een vlucht van 25 uur, een "gewone" werkdag hadden met een welkomstdiner in de avond. De eerste weken ben ik door Doaitse wegwijs gemaakt in dit verre land en kreeg ik van alles te zien, waaronder een bezoek aan de Altiplano. Het was gelijk ook mijn eerste kennismaking met onderzoek.

Een opleiding tot onderzoeker bij Doaitse betekent haast automatisch ook een voorliefde voor Haskell, parser combinators, attributengrammatica's en vertalerbouw. Met dit in gedachten heb ik voor zijn afscheid van de academische wereld de Haskell code voor het opstellen van traversal-strategieën in het Ideas raamwerk beschreven. Haskell krijgt steeds meer extensies om de uitdrukkingskracht te vergroten en die extensies zijn regelmatig terug te vinden in Doaitse's eigen code. De code in dit artikel gebruikt geen enkele extensie en laat zien dat je ook met Haskell 98 typeklassen en wat standaard programmeertechnieken tot een flexibele oplossing kunt komen.

1 Introduction

Traversal strategies are recursive procedures that specify how and where a given transformation has to be applied when rewriting an expression. There is quite some variation in these traversal strategies, such as full traversals (everywhere) or single location traversals, top-down versus bottom-up traversals, right-to-left traversals (instead of the standard left-to-right direction), and fixed-point traversals. These variations are orthogonal, and even more dimensions can be added to further increase flexibility. Controlling this flexibility is challenging from a software engineering point of view.

Traversals are important in interactive exercise assistants that are based on rewriting strategies [1]. For example, when rewriting a logical proposition to disjunctive normal form, negations have to be pushed inside the expression, for instance with the De Morgan rules.

Using a top-down approach will generally lead to fewer rewrite steps, which we would like to specify in the strategy. Another example of a traversal strategy is the outermost evaluation strategy for the lambda calculus in a functional programming tutor.

In many cases, strategies have to control where certain transformations are applied, and for this, position information is needed. When simplifying rational expressions (e.g. $\frac{2x}{x-1} + \frac{x}{x+2}$), it is customary to expand the numerator of the expression, but to factor the denominator (as in $\frac{3x^2+3x}{(x-1)(x+2)}$). In the Ask-Elle functional programming tutor [2], strategies determine which hole has to be refined.

In this paper, we show how traversals and position information are added to the strategy combinator language. The new combinators are polymorphic and do not depend on the type of expressions that are rewritten. The presented solution is in Haskell 98 and is based on type classes for giving the desired flexibility. Adapter types change the behavior of instances of these type classes.

The literature on traversals is quite extensive [3,4]. In the context of interactive exercises, however, we are also interested in the intermediate steps of a traversal, and not only in the end result. Furthermore, strategies for interactive exercises are typically non-deterministic, leaving some choice to the student, and the traversal combinators also have to facilitate this. In the remainder of the paper we will explore zipper-like structures [5] for maintaining a focus in an expression, and define traversal combinators with the zipper interface.

2 Iterators

We use functions of the type $a \to \mathit{Maybe}\ a$ for representing transformations that either succeed with a single result, or fail. Although we will refrain from defining numerous combinators for this function type, the following sequencing combinator (also defined in `Control.Monad`) will turn out to be useful:

$$(>=>) :: (a \to \mathit{Maybe}\ a) \to (a \to \mathit{Maybe}\ a) \to a \to \mathit{Maybe}\ a$$
$$(f >=> g)\ a = f\ a >>= g$$

We start by defining a type class for iterating over a sequence of elements, bringing each element in focus. This class *Iterator* is named after the design pattern by the Gang of Four [6] and provides transformations for moving forwards (*next*) and backwards (*previous*) in a list. We use the *Maybe* data type to signal that we have reached the end of either side of the sequence: this is more informative than returning the current element. The navigation functions *first* and *final* bring the focus to the first and final (i.e., last) element, respectively. Function *position* returns the index of the element in focus.

```
  -- minimal complete definition: previous and next
class Iterator a where
  previous, next :: a → Maybe a
  first, final   :: a → a
  position       :: a → Int
    -- default implementations
  first    = fixp previous
  final    = fixp next
  position = pred ∘ length ∘ fixpl previous
```

Figure 1 shows the navigation functions of the *Iterator* type class, where the box denotes the element that is currently in focus, and a dashed arrow represents zero or more steps. The

Fig. 1. Navigation functions of the *Iterator* type class

type class provides default implementations for *first*, *final*, and *position*: these functions can be expressed by repeatedly applying *previous* and *next*. The helper functions *fixpl* and *fixp* are defined below, where *fixpl* returns the list of intermediate values until the fixed-point is reached:

$$
\begin{aligned}
&\mathit{fixpl} :: (a \to \mathit{Maybe}\ a) \to a \to [a] \\
&\mathit{fixpl}\ f\ a = a : \mathit{maybe}\ [\,]\ (\mathit{fixpl}\ f)\ (f\ a) \\
&\mathit{fixp} :: (a \to \mathit{Maybe}\ a) \to a \to a \\
&\mathit{fixp}\ f = \mathit{last} \circ \mathit{fixpl}\ f
\end{aligned}
$$

A minimal complete instance must supply definitions for *previous* and *next*. In this paper, instances are limited to this minimal set, although more definitions can be provided in a real implementation for reasons of efficiency. More functions can be defined with the *Iterator* type class, such as the following four predicates:

$$
\begin{aligned}
&\mathit{hasPrevious}, \mathit{hasNext} :: \mathit{Iterator}\ a \Rightarrow a \to \mathit{Bool} \\
&\mathit{hasPrevious} = \mathit{isJust} \circ \mathit{previous} \\
&\mathit{hasNext}\ \ \ \ = \mathit{isJust} \circ \mathit{next} \\
&\mathit{isFirst}, \mathit{isFinal} :: \mathit{Iterator}\ a \Rightarrow a \to \mathit{Bool} \\
&\mathit{isFirst}\ = \mathit{not} \circ \mathit{hasPrevious} \\
&\mathit{isFinal} = \mathit{not} \circ \mathit{hasNext}
\end{aligned}
$$

It should be noted that the type class does not provide a function for getting the current element. In fact, it might be tempting to define *Iterator* as a type constructor class (of kind $* \to *$), allowing us to access and change the current element. Instead, we keep the *Iterator* type class simple and introduce another type class for this functionality.

2.1 Iterator properties

Before we start defining instances of the *Iterator* type class, it is a good practice to think about the laws that each instance should satisfy. Intuitively, the transformations *previous* and *next* cancel each other, unless we are near the end of the sequence. Definitions for the other three functions (*first*, *final*, and *position*) are tested against the default definitions of the type class.

The class laws can be specified as QuickCheck [7] properties. We first lift preconditions ($\Longrightarrow$) and equality ($\equiv$) to functions:

$$
\begin{aligned}
&\mathbf{infixr}\ 0\ \overset{\circ}{\Longrightarrow} \\
&\mathbf{infix}\ \ 4\ \ \overset{\circ}{\equiv} \\
&(\overset{\circ}{\Longrightarrow}) :: \mathit{Testable}\ \mathit{prop} \Rightarrow (t \to \mathit{Bool}) \to (t \to \mathit{prop}) \to t \to \mathit{Property} \\
&(p \overset{\circ}{\Longrightarrow} q)\ a = p\ a \Longrightarrow q\ a \\
&(\overset{\circ}{\equiv}) :: \mathit{Eq}\ a \Rightarrow (t \to a) \to (t \to a) \to t \to \mathit{Bool} \\
&(f \overset{\circ}{\equiv} g)\ a = f\ a \equiv g\ a
\end{aligned}
$$

Next, the two class laws are defined as:

$$propPreviousNext, propNextPrevious :: (Eq\ a, Iterator\ a) \Rightarrow a \rightarrow Property$$
$$propPreviousNext = hasPrevious \overset{\circ}{\Longrightarrow} (previous \gg\!= next) \overset{\circ}{\equiv} Just$$
$$propNextPrevious = hasNext \overset{\circ}{\Longrightarrow} (next \gg\!= previous) \overset{\circ}{\equiv} Just$$

2.2 List iterator

A simple data type to iterate over is the list type, and for this we introduce the *ListIterator* data type. A *ListIterator* consists of the first part of the list (in reversed order), the element in focus, and the last part of the list:

$$\textbf{data}\ ListIterator\ a = LI\ [a]\ a\ [a]\ \textbf{deriving}\ Eq$$

The function *makeListIterator* constructs a *ListIterator* from a list, putting the first element in focus. Function *fromListIterator* returns the list from the *ListIterator* and discards the focus.

$$makeListIterator :: [a] \rightarrow Maybe\ (ListIterator\ a)$$
$$makeListIterator\ [\,] = Nothing$$
$$makeListIterator\ (x : xs) = Just\ (LI\ [\,]\ x\ xs)$$
$$fromListIterator :: ListIterator\ a \rightarrow [a]$$
$$fromListIterator\ (LI\ xs\ y\ ys) = reverse\ xs +\!\!+ y : ys$$

We turn *ListIterator* into an *Iterator* by defining the instance declaration. Because the first part of the list is stored in reversed order, moving to the next element means that the current element is put in front of the first part.

$$\textbf{instance}\ Iterator\ (ListIterator\ a)\ \textbf{where}$$
$$previous\ (LI\ (x : xs)\ y\ ys) = Just\ (LI\ xs\ x\ (y : ys))$$
$$previous\ \_ = Nothing$$
$$next\ (LI\ xs\ x\ (y : ys)) = Just\ (LI\ (x : xs)\ y\ ys)$$
$$next\ \_ = Nothing$$

This instance satisfies the type class laws, although we will not demonstrate this by defining a QuickCheck generator and checking the properties. With an instance for the *Show* type class we can start running some examples.

$$\textbf{instance}\ Show\ a \Rightarrow Show\ (ListIterator\ a)\ \textbf{where}$$
$$show\ (LI\ xs\ y\ ys) =$$
$$\textbf{let}\ listLike = brackets \circ intercalate\ \texttt{","}$$
$$brackets\ s = \texttt{"["} +\!\!+ s +\!\!+ \texttt{"]"}$$
$$focusOn\ s = \texttt{"<<"} +\!\!+ s +\!\!+ \texttt{">>"}$$
$$\textbf{in}\ listLike\ (map\ show\ (reverse\ xs) +\!\!+ focusOn\ (show\ y) : map\ show\ ys)$$

The following ghci session demonstrates how to navigate over a string:

```
*Main> let Just lit = makeListIterator "Doaitse"
*Main> (next >=> next) lit
Just ['D','o',<<'a'>>,'i','t','s','e']
```

2.3 The Update type class

We introduce a type class for extracting the element in focus and for changing this element. Function *update* returns the current element for some structure, together with a function to put a new value back into the structure. We turn the *ListIterator* type into an instance of this type class.

```
class Update f where
  update :: f a → (a, a → f a)
instance Update ListIterator where
  update (LI xs y ys) = (y, λz → LI xs z ys)
```

Helper function *changeM* applies a transformation to the current element.

```
current :: Update f ⇒ f a → a
current = fst ∘ update
changeM :: Update f ⇒ (a → Maybe a) → f a → Maybe (f a)
changeM g x = let (a, h) = update x in fmap h (g a)
```

A generalization of *update*'s type is not restricted to types of the form $f\ a$:

```
type UpdateType a b = a → (b, b → a)
```

This type is a category, with the identity morphism $\lambda x \to (x, id)$, and the (left-to-right) composition operator defined below:

```
(>>>) :: UpdateType a b → UpdateType b c → UpdateType a c
(f >>> g) a = let (b, ff) = f a
                  (c, gg) = g b
              in (c, ff ∘ gg)
```

3 Adapters for type classes

Adapters change the behavior of type class instances, without changing the type class or the instances themselves. Adapters are a well-known approach to introducing variation for type classes, such that these variations can be easily composed. In this section we consider mirrors and filters.

3.1 Mirrors

We define a mirror that swaps the navigation functions *previous* and *next* of the *Iterator* type class. We make the *Mirror* data type an instance of *Update*, so that we can use *changeM*.

```
newtype Mirror a = Mirror a deriving (Show, Eq)
instance Update Mirror where
  update (Mirror a) = (a, Mirror)
instance Iterator a ⇒ Iterator (Mirror a) where
  previous = changeM next
  next     = changeM previous
```

For instance, if we move to the first element of a mirrored *ListIterator*, the focus will be on the last element of the underlying list.

```
*Main> first (Mirror lit)
Mirror ['D','o','a','i','t','s',<<'e'>>]
```

3.2 Filters

A second example of an adapter is the data type *Filter*, which we use to skip certain elements based on some predicate.

```
data Filter a = Filter (a → Bool) a
instance Show a ⇒ Show (Filter a) where
  show (Filter _ a) = show a
instance Update Filter where
  update (Filter p a) = (a, Filter p)
```

Given a predicate and a transformation, we search for the first element that satisfies the predicate (if such an element exists). Note that *searchStep* starts by applying the transformation, whereas *searchWith* first uses the predicate.

```
searchStep :: (a → Bool) → (a → Maybe a) → a → Maybe a
searchStep p f = f >=> searchWith p f
searchWith :: (a → Bool) → (a → Maybe a) → a → Maybe a
searchWith p f a | p a          = Just a
                 | otherwise = f a >>= searchWith p f
```

With these helper-functions, the instance for the *Iterator* type class becomes easy to define:

```
instance Iterator a ⇒ Iterator (Filter a) where
  previous (Filter p a) = fmap (Filter p) (searchStep p previous a)
  next     (Filter p a) = fmap (Filter p) (searchStep p next a)
```

Two observations about the *Filter* instance are significant. Firstly, we do not filter based on the current element of the structure inside the filter, but on the entire structure. For example, if we have a value of type *Filter* (*ListIterator Int*), then the predicate works on the *ListIterator Int* value at hand. This has the advantage that we can also inspect the current position of the iterator, or visit neighboring elements. Secondly, when constructing (or changing) a filter, we have to make sure that the focus is always on an element that satisfies the predicate. If we do not take care of this invariant of the Filter data type, the *Iterator* type class laws no longer hold for the *Filter* instance, resulting in unpredictable behavior. To circumvent this issue, we introduce a smart constructor that uses the function *next* for finding a suitable focus.

```
makeFilter :: Iterator a ⇒ (a → Bool) → a → Maybe (Filter a)
makeFilter p = fmap (Filter p) ∘ searchWith p next
```

We present an example of a filtered *ListIterator* that only focuses on vowels.

```
*Main> let isVowel = (`elem` "aeiou")
*Main> makeFilter (isVowel . current) lit >>= next
Just ['D','o',<<'a'>>,'i','t','s','e']
```

4 Navigators

The *Iterator* type class is for iterating over a sequence of elements. We introduce the *Navigator* type class for navigating in a tree-like structure.

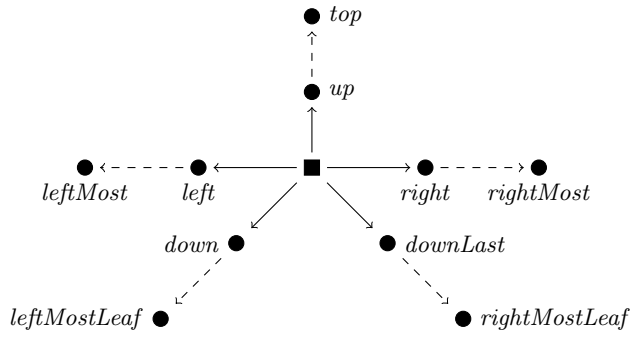


Fig. 2. Navigation functions of the *Navigator* type class

The type class offers navigation functions for moving *up* (to the parent), *left* or *right* (to a sibling), and *down* (to the first child). For symmetry, we also introduce *downLast* for moving to the last child. Finally, functions *childnr* and *location* are added to the type class to provide position information. Figure 2 provides an overview of the navigation functions of the *Navigator* type class.

```
  -- minimal complete definition: up, down, left, and right
class Navigator a where
  up, down, downLast :: a → Maybe a
  left, right        :: a → Maybe a
  childnr            :: a → Int
  location           :: a → [Int]
    -- default definitions
  downLast = fmap (fixp right) ∘ down
  childnr  = pred ∘ length ∘ fixpl left
  location = map childnr ∘ drop 1 ∘ reverse ∘ fixpl up
```

For each of the five navigation functions, we define a function that calculates the fixed-point: *top*, *leftMost*, *rightMost*, *leftMostLeaf*, and *rightMostLeaf* have type $Navigator\ a \Rightarrow a \to a$. We omit their definitions in this paper. Similarly, we define predicates *isTop*, *isLeaf*, *hasUp*, *hasDown*, *hasLeft*, and *hasRight* that inspect whether a certain transformation succeeds or not. Predicate *isTop* is the negation of *hasUp*; *isLeaf* is the negation of *hasDown*.

4.1 Navigator properties

Let us now consider the properties of the *Navigator* type class. These are more involved since there are five directions in which we can move. Only the interaction between the functions of the minimal complete set are examined. All properties have type $(Eq\ a, Navigator\ a) \Rightarrow a \to Property$.

The first group of laws specifies the interaction between *up* and *down*. Going down followed by going up is the identity transformation. The other way around, however, is identical to moving to the leftmost sibling. The third property is the symmetrical case when using *downLast*.

$$
\begin{aligned}
&\mathit{propDownUp} &&= \mathit{hasDown} \overset{\circ}{\Longrightarrow} (\mathit{down} \gg\!= \mathit{up}) &&\overset{\circ}{\equiv} \mathit{Just}\\
&\mathit{propUpDown} &&= \mathit{hasUp} \overset{\circ}{\Longrightarrow} (\mathit{up} \gg\!= \mathit{down}) &&\overset{\circ}{\equiv} \mathit{Just} \circ \mathit{leftMost}\\
&\mathit{propUpDownLast} &&= \mathit{hasUp} \overset{\circ}{\Longrightarrow} (\mathit{up} \gg\!= \mathit{downLast}) &&\overset{\circ}{\equiv} \mathit{Just} \circ \mathit{rightMost}
\end{aligned}
$$

The next two laws specify that transformations *left* and *right* cancel each other.

$$
\begin{aligned}
&\mathit{propRightLeft} = \mathit{hasRight} \overset{\circ}{\Longrightarrow} (\mathit{right} \gg\!= \mathit{left}) \overset{\circ}{\equiv} \mathit{Just}\\
&\mathit{propLeftRight} = \mathit{hasLeft} \overset{\circ}{\Longrightarrow} (\mathit{left} \gg\!= \mathit{right}) \overset{\circ}{\equiv} \mathit{Just}
\end{aligned}
$$

The final laws state that moving left or right before moving up has no effect: instead, one can immediately go up.

$$
\begin{aligned}
&\mathit{propLeftUp} &&= \mathit{hasLeft} \overset{\circ}{\Longrightarrow} (\mathit{left} \gg\!= \mathit{up}) &&\overset{\circ}{\equiv} \mathit{up}\\
&\mathit{propRightUp} &&= \mathit{hasRight} \overset{\circ}{\Longrightarrow} (\mathit{right} \gg\!= \mathit{up}) &&\overset{\circ}{\equiv} \mathit{up}
\end{aligned}
$$

4.2 Uniplate navigator

The zipper data structure [5] maintains a focus in a tree-like structure, and is an obvious candidate for the *Navigator* type class. Zippers can be defined generically [8]. We will do the same based on the *Uniplate* library [9]. This library defines the *Uniplate* type class which provides access to the (direct) children of a value by means of the function *uniplate*. This function has type $a \to ([a], [a] \to a)$, which is equal to $\mathit{UpdateType}\ a\ [a]$.

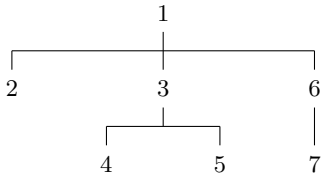
Because the *uniplate* function provides access to the list of children of a value, we reuse the *ListIterator* data type for representing this list and providing horizontal movement (*left* and *right*). In addition, we use a list of functions that encodes the context in which this *ListIterator* operates. The representation of *UniplateNavigator* and its instance are as follows:

```
data UniplateNavigator a =
       U [ListIterator a → ListIterator a] (ListIterator a)
instance Uniplate a ⇒ Navigator (UniplateNavigator a) where
  up    (U []       _) = Nothing
  up    (U (f : fs) a) = Just (U fs (f a))
  down (U fs       a) = fmap (U (f : fs)) (makeListIterator cs)
    where
      (cs, g) = (update >>> uniplate) a
      f       = g ∘ fromListIterator
  left  (U fs a) = fmap (U fs) (previous a)
  right (U fs a) = fmap (U fs) (next a)
```

Most of the complexity is found in the definition of *down*. In this definition, we update the *ListIterator* a, and calculate the children of the current element. Note how *update* and *uniplate* can conveniently be composed with the $>\!>\!>$-operator. The list of children *cs* is turned into a new *ListIterator*, with the head of the element in focus. Function g replaces the current list of children with a new list, and is used (in f) to extend the context *fs*.

When constructing a *UniplateNavigator* value, we make a *ListIterator* from a singleton list. This always succeeds (hence the *fromJust*) because the list is non-empty.

```
data Rose a = Rose a [Rose a]
  deriving Eq
leaf :: a → Rose a
leaf a = Rose a []
instance Show a ⇒ Show (Rose a) where
  show (Rose a xs)
    | null xs    = show a
    | otherwise = show a ++ show xs
instance Update Rose where
  update (Rose a xs) = (a, λb → Rose b xs)
instance Uniplate (Rose a) where
  uniplate (Rose a xs) = (xs, Rose a)
```




```
rose :: Rose Int
rose =
  Rose 1 [leaf 2
         , Rose 3 [leaf 4, leaf 5]
         , Rose 6 [leaf 7]]
```

Fig. 3. Rose tree data type with an example tree

```
makeUniplateNavigator :: a → UniplateNavigator a
makeUniplateNavigator = U [] ∘ fromJust ∘ makeListIterator ∘ return
```

When showing a *UniplateNavigator*, we show the entire value, followed by the current value in focus and the location of the focus. We also provide an instance for the *Update* type class.

```
instance (Show a, Uniplate a) ⇒ Show (UniplateNavigator a) where
  show a = show (current (top a)) ++
           " << " ++ show (current a) ++
           " @ "  ++ show (location a) ++ " >>"
instance Update UniplateNavigator where
  update (U fs a) = let (b, g) = update a in (b, U fs ∘ g)
```

4.3 Example

Figure 3 defines the *Rose* data type, together with *Show*, *Update*, and *Uniplate* instances. The right-hand side contains an example value of the *Rose* data type. The following session in ghci illustrates how to navigate in this example tree (where `it` refers to the result of the previous command in ghci).

```
*Main> makeUniplateNavigator rose
1[2,3[4,5],6[7]] << 1[2,3[4,5],6[7]] @ [] >>
*Main> down it
Just 1[2,3[4,5],6[7]] << 2 @ [0] >>
*Main> it >>= right
Just 1[2,3[4,5],6[7]] << 3[4,5] @ [1] >>
*Main> it >>= downLast
Just 1[2,3[4,5],6[7]] << 5 @ [1,1] >>
```

4.4 Adapters for navigators

The adapters defined in Section 3 can also be used for the *Navigator* type class. In the case of the *Mirror* adapter, we choose to swap *left* and *right*, and also *down* and *downLast*. The *Filter* adapter filters sub-trees based on its predicate.

```
  -- basic strategies
rule      :: String → (a → Maybe a) → Strategy a
trans     :: (a → Maybe a) → Strategy a
check     :: (a → Bool) → Strategy a
succeed   :: Strategy a
  -- strategy combinators
(<*>), (<|>), (▷) :: Strategy a → Strategy a → Strategy a
  -- derived combinators
try, repeatS :: Strategy a → Strategy a
try       s = s ▷ succeed
repeatS s = try (s <*> repeatS s)
```

Fig. 4. Interface of the strategy language

```
instance Navigator a ⇒ Navigator (Mirror a) where
  up    = changeM up
  down  = changeM downLast
  left  = changeM right
  right = changeM left
instance Navigator a ⇒ Navigator (Filter a) where
  up    (Filter p a) = fmap (Filter p) (up a)
  down  (Filter p a) = fmap (Filter p) (down a >>= searchWith p right)
  left  (Filter p a) = fmap (Filter p) (searchStep p left a)
  right (Filter p a) = fmap (Filter p) (searchStep p right a)
```

5 Strategies

Strategies [1] are procedures for the step-wise rewriting of a term. Running a strategy on a term gives a list of alternative results ($run :: Strategy\ a \to a \to [a]$). For each result, we can calculate the intermediate steps and inspect which rules were used. These intermediate steps and rules make up a derivation, which shows how the original term is transformed into the final result. Derivations are constructed by stepping through the strategy using the functions *firsts* and *empty*: these functions are inspired by the context-free grammar functions found in parser implementations. We use function *derivations* of type $Show\ a \Rightarrow Strategy\ a \to a \to IO\ ()$ to show all derivations.

Figure 4 shows the interface of the strategy language. Basic strategies are constructed from transformations with a name (function *rule*), or from anonymous transformations that do not show up as intermediate steps (function *trans*). Strategy *check p* only succeeds if the predicate p holds; strategy *succeed* always succeeds. The strategy language offers combinators for sequence (<*>) and choice (<|>). Strategy $s \triangleright t$ denotes left-biased choice and only applies strategy t if s fails. Many more combinators can be derived, such as *try* and *repeatS*, which are both greedy combinators. In this paper we will not consider the interleaving of strategies [10].

For example, consider the strategy *collatz* for the $3n + 1$ conjecture, constructed from two rules that rewrite an integer:

```
ruleEven, ruleOdd :: Strategy Int
ruleEven = rule "even" f where
```

```
  f x | even x      = Just (x `div` 2)
      | otherwise  = Nothing
ruleOdd = rule "odd" f where
  f x | odd x       = Just (3 * x + 1)
      | otherwise  = Nothing
collatz :: Strategy Int
collatz = repeatS (check (>1) <*> (ruleEven <|> ruleOdd))
```

Running *collatz* on integer 10 results in one derivation:

```
10 ⇒{even} 5 ⇒{odd} 16 ⇒{even} 8 ⇒{even} 4 ⇒{even} 2 ⇒{even} 1
```

5.1 Lifting strategies

Because the *Strategy* data type is not a functor, something else is needed for lifting a strategy to another type. A strategy can be applied to a sub-part of some term if we know how to access this sub-part, and how to update it. This functionality is exactly what *UpdateType* provides. Hence, the strategy language can be extended by one more primitive combinator for lifting:

```
liftS :: UpdateType a b → Strategy b → Strategy a
```

Of course, we can use the *Update* type class for lifting a strategy to an instance of this class:

```
liftTo :: Update f ⇒ Strategy a → Strategy (f a)
liftTo = liftS update
```

For the other way around, we also need a constructor function $a \to f\ a$, since such a function is not part of the *Update* type class:

```
liftFrom :: Update f ⇒ (a → f a) → Strategy (f a) → Strategy a
liftFrom f = liftS (λx → (f x, current))
```

With *liftTo* and *liftFrom*, we can also lift unary strategy combinators:

```
liftWith :: Update f ⇒ (a → f a) → (Strategy (f a) → Strategy (f a))
                                 → Strategy a → Strategy a
liftWith f sf = liftFrom f ∘ sf ∘ liftTo
```

6 Traversals

We can now define all kinds of generic traversal combinators using the *Iterator* and *Navigator* type classes for navigation, and the strategy language for composing the navigation functions into traversals.

6.1 Visits for sequences

Suppose that we have a sequence over which we can iterate. We can then apply a strategy to one (arbitrary) element of the sequence, or to the first element of the sequence where the strategy succeeds. We assume that the iterator that is rewritten by the strategy has its focus on the first element of the sequence.

$$
\begin{aligned}
&\mathit{visitOne}, \mathit{visitFirst} :: \mathit{Iterator}\ a \Rightarrow \mathit{Strategy}\ a \to \mathit{Strategy}\ a \\
&\mathit{visitOne}\ \ s = s \mathbin{<|>} (\mathit{trans}\ \mathit{next} \mathbin{<\star>} \mathit{visitOne}\ s) \\
&\mathit{visitFirst}\ s = s \triangleright\ \ (\mathit{trans}\ \mathit{next} \mathbin{<\star>} \mathit{visitFirst}\ s)
\end{aligned}
$$

It is also possible to apply the strategy to more than one element. Combinator *visitAll* applies the strategy to all elements of an iterator, and fails if the strategy fails for some element. Combinator *visitMany* applies the strategy to as many elements as possible. Lastly, *visitSome* applies the strategy to at least one element (and as many more as possible).

$$
\begin{aligned}
&\mathit{visitAll}, \mathit{visitMany}, \mathit{visitSome} :: \mathit{Iterator}\ a \Rightarrow \mathit{Strategy}\ a \to \mathit{Strategy}\ a \\
&\mathit{visitAll}\ \ \ s = s\ \ \ \ \mathbin{<\star>} (\mathit{check}\ \mathit{isFinal} \triangleright (\mathit{trans}\ \mathit{next} \mathbin{<\star>} \mathit{visitAll}\ s)) \\
&\mathit{visitMany}\ s = \mathit{try}\ s \mathbin{<\star>} (\mathit{check}\ \mathit{isFinal} \triangleright (\mathit{trans}\ \mathit{next} \mathbin{<\star>} \mathit{visitMany}\ s)) \\
&\mathit{visitSome}\ s = s\ \ \ \ \mathbin{<\star>} \mathit{try}\ (\mathit{trans}\ \mathit{next} \mathbin{<\star>} \mathit{visitMany}\ s) \\
&\qquad\qquad \triangleright\ \mathit{trans}\ \mathit{next} \mathbin{<\star>} \mathit{visitSome}\ s
\end{aligned}
$$

We illustrate *visitSome* combined with *ruleOdd* for rewriting the odd numbers.

```
let Just ints = makeListIterator [1..5]
*Main> derivations (visitSome (liftTo ruleOdd)) ints
[<<1>>,2,3,4,5]  ⇒{odd}  [<<4>>,2,3,4,5]
                 ⇒{odd}  [4,2,<<10>>,4,5]
                 ⇒{odd}  [4,2,10,4,<<16>>]
```

6.2 One-layer traversals

The *layer* combinator applies a strategy to the left-most child. We first move down, apply the strategy, and go back up again:

$$
\begin{aligned}
&\mathit{layer} :: \mathit{Navigator}\ a \Rightarrow \mathit{Strategy}\ a \to \mathit{Strategy}\ a \\
&\mathit{layer}\ s = \mathit{trans}\ \mathit{down} \mathbin{<\star>} s \mathbin{<\star>} \mathit{trans}\ \mathit{up}
\end{aligned}
$$

The strategy we want to supply to *layer* uses one of the visit functions, specifying how to visit the children of the current node (e.g., visit one child or visit all children). A technical issue that we need to solve first is that the visit functions depend on the *Iterator* interface, whereas *layer* expects a *Navigator*. For this reason we introduce the *Horizontal* adapter, which turns a *Navigator* into an *Iterator* by using *left*/*right* for *previous*/*next*. It should be noted that there are many more meaningful ways to implement the *Iterator* interface for a *Navigator*, such as the pre-order and post-order tree traversals.

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{Horizontal}\ a = \mathit{Horizontal}\ a\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}) \\
&\textbf{instance}\ \mathit{Update}\ \mathit{Horizontal}\ \textbf{where} \\
&\quad \mathit{update}\ (\mathit{Horizontal}\ a) = (a, \mathit{Horizontal}) \\
&\textbf{instance}\ \mathit{Navigator}\ a \Rightarrow \mathit{Iterator}\ (\mathit{Horizontal}\ a)\ \textbf{where} \\
&\quad \mathit{previous} = \mathit{changeM}\ \mathit{left} \\
&\quad \mathit{next}\ \ \ \ = \mathit{changeM}\ \mathit{right}
\end{aligned}
$$

With the *Horizontal* data type, we can now apply a strategy to one layer by lifting the visit function:

$$
\begin{aligned}
&\mathit{layerOne}, \mathit{layerFirst}, \mathit{layerAll} :: \mathit{Navigator}\ a \Rightarrow \mathit{Strategy}\ a \to \mathit{Strategy}\ a \\
&\mathit{layerOne}\ \ = \mathit{layer} \circ \mathit{horizontal}\ \mathit{visitOne} \\
&\mathit{layerFirst} = \mathit{layer} \circ \mathit{horizontal}\ \mathit{visitFirst} \\
&\mathit{layerAll}\ \ = \mathit{layer} \circ \mathit{horizontal}\ \mathit{visitAll} \\
&\mathit{horizontal} = \mathit{liftWith}\ \mathit{Horizontal}
\end{aligned}
$$

6.3 One-pass traversals

With the one-layer traversal combinators it becomes easy to define combinators for one-pass traversals. Top-down traversals first transform the current position before descending into the tree. For a full traversal (which applies a strategy everywhere), we have to check whether we are at a leaf node, in which case *layerAll* would fail. If we are at a leaf node, we do not attempt to visit the next layer. Instead, we are done. In addition to full and once traversals, we also define so-called spine and stop traversals [4].

$$
\begin{aligned}
&\mathit{fulltd}, \mathit{spinetd}, \mathit{stoptd}, \mathit{oncetd} :: \mathit{Navigator}\ a \Rightarrow \mathit{Strategy}\ a \to \mathit{Strategy}\ a \\
&\mathit{fulltd}\quad s = s \mathbin{<\!\star\!>} (\mathit{check}\ \mathit{isLeaf} \triangleright \mathit{layerAll}\quad (\mathit{fulltd}\ s)) \\
&\mathit{spinetd}\ s = s \mathbin{<\!\star\!>} (\mathit{check}\ \mathit{isLeaf} \triangleright \mathit{layerFirst}\ (\mathit{spinetd}\ s)) \\
&\mathit{stoptd}\quad s = s \triangleright \mathit{layerAll}\quad (\mathit{stoptd}\ s) \\
&\mathit{oncetd}\ \ s = s \triangleright \mathit{layerFirst}\ (\mathit{oncetd}\ s)
\end{aligned}
$$

In the same way, we can define bottom-up traversals by swapping the order of strategies. We only define *fullbu* and *oncebu*:

$$
\begin{aligned}
&\mathit{fullbu}, \mathit{oncebu} :: \mathit{Navigator}\ a \Rightarrow \mathit{Strategy}\ a \to \mathit{Strategy}\ a \\
&\mathit{fullbu}\quad s = (\mathit{check}\ \mathit{isLeaf} \triangleright \mathit{layerAll}\ (\mathit{fullbu}\ s)) \mathbin{<\!\star\!>} s \\
&\mathit{oncebu}\ s = \mathit{layerFirst}\ (\mathit{oncebu}\ s) \triangleright s
\end{aligned}
$$

For a right-to-left traversal, we simply mirror the underlying navigator. Likewise, we can filter the traversal with some predicate.

$$
\begin{aligned}
&\mathit{mirrored} = \mathit{liftWith}\ \mathit{Mirror} \\
&\mathit{filtered}\ p = \mathit{liftWith}\ (\mathit{Filter}\ p)
\end{aligned}
$$

The above traversal combinators are rather strict in their order. If a strategy should be applied somewhere and we don't care about the location, then the following combinator can be used:

$$
\begin{aligned}
&\mathit{somewhere} :: \mathit{Navigator}\ a \Rightarrow \mathit{Strategy}\ a \to \mathit{Strategy}\ a \\
&\mathit{somewhere}\ s = s \mathbin{<|>} \mathit{layerOne}\ (\mathit{somewhere}\ s)
\end{aligned}
$$

For example, consider the rose tree in Figure 3. Suppose we define a strategy that applies *ruleEven* or *ruleOdd*, and we construct a full bottom-up traversal, which works from right to left, and which skips the entire sub-tree labeled 3.

$$
\begin{aligned}
&\mathit{evenOdd} = \mathit{ruleEven} \mathbin{<|>} \mathit{ruleOdd} \\
&\mathit{trav}\quad\ \ = \mathit{filtered}\ ((\not\equiv 3) \circ \mathit{current} \circ \mathit{current})\ (\mathit{mirrored}\ \mathit{fullbu})
\end{aligned}
$$

If we use this traversal with the lifted strategy *evenOdd* for a navigator on the rose tree, we can observe that the nodes 7, 6, 2, and 1 are changed, in that order.

```
1[2,3[4,5],6[7]]  ⇒{odd}   1[2,3[4,5],6[22]]
                  ⇒{even}  1[2,3[4,5],3[22]]
                  ⇒{even}  1[1,3[4,5],3[22]]
                  ⇒{odd}   4[1,3[4,5],3[22]]
```

6.4 Fixed-point traversals

One-pass traversals can be turned into traversals with multiple passes, for instance by calculating the fixed-point with the *repeatS* combinator. We define the traversal combinators *innermost* and *outermost*.

$innermost, outermost :: Navigator\ a \Rightarrow Strategy\ a \to Strategy\ a$
$innermost = repeatS \circ oncebu$
$outermost = repeatS \circ oncetd$

These traversals can for instance be used for evaluating lambda expressions. As a final example we show a derivation for $S\ K\ K$ with β-reduction and definitions for the S and K combinators, where we also reduce 'under a lambda'.

$$\begin{array}{lll} S\ K\ K & \Rightarrow\{\texttt{S combinator}\} & (\lambda f \to \lambda g \to \lambda x \to f\ x\ (g\ x))\ K\ K \\ & \Rightarrow\{\beta\texttt{-reduction}\} & (\lambda g \to \lambda x \to K\ x\ (g\ x))\ K \\ & \Rightarrow\{\beta\texttt{-reduction}\} & \lambda x \to K\ x\ (K\ x) \\ & \Rightarrow\{\texttt{K combinator}\} & \lambda x \to (\lambda x \to \lambda y \to x)\ x\ (K\ x) \\ & \Rightarrow\{\beta\texttt{-reduction}\} & \lambda x \to (\lambda y \to x)\ (K\ x) \\ & \Rightarrow\{\beta\texttt{-reduction}\} & \lambda x \to x \end{array}$$

7 Conclusions

We have shown how traversal combinators can be defined based on the *Iterator* and *Navigator* type classes, with zipper-like data structures for keeping a point of focus. These navigation functions can be used in rewriting strategies, making it possible to inspect all intermediate steps of a derivation, and to have multiple of these derivations. Adapters for the *Iterator* and *Navigator* type classes help to support more variations for the traversals. The paper is limited to simple recursive data types, even though the Ideas framework [1] also uses traversals over more complicated, nested data structures. It is future work to investigate how this can be done in a convenient and type-safe way.

References

1. Heeren, B., Jeuring, J., Gerdes, A.: Specifying rewrite strategies for interactive exercises. Mathematics in Computer Science **3** (2010) 349–370
2. Gerdes, A., Jeuring, J., Heeren, B.: An interactive functional programming tutor. In: Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education. ITiCSE '12, New York, ACM (2012) 250–255
3. Visser, E., Benaissa, Z.e.A., Tolmach, A.: Building program optimizers with rewriting strategies. SIGPLAN Not. **34**(1) (1998) 13–26
4. Ren, D., Erwig, M.: A generic recursion toolbox for Haskell or: scrap your boilerplate systematically. In: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell. Haskell '06, New York, ACM (2006) 13–24
5. Huet, G.: The zipper. J. Funct. Program. **7**(5) (1997) 549–554
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston (1995)
7. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. ICFP '00, New York, ACM (2000) 268–279
8. Hinze, R., Jeuring, J.: Generic Haskell: applications. In: In Generic Programming, Advanced Lectures, volume 2793 of LNCS, Springer-Verlag (2003) 57–97
9. Mitchell, N., Runciman, C.: Uniform boilerplate and list processing. In: Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell, ACM (2007) 49–60
10. Heeren, B., Jeuring, J.: Interleaving strategies. In: Proceedings of the 18th Calculemus and 10th international conference on Intelligent computer mathematics. MKM'11, Berlin, Heidelberg, Springer-Verlag (2011) 196–211

Testing proven algorithms

Thomas Arts[1]

thomas.arts@quviq.com
Quviq AB, Gothenburg, Sweden

Abstract. Much of computer science education and research is based on a dogmatic belief that developing correct software should be based on mathematical proof of correctness. When such a proof is hard to provide at once, then one could refine the software proving correctness of each refinement or one could try to modularise the software after a modular proof of correctness. Methods and tools to assist in this task have been developed for decades with impressive results. But despite these methods, tools, and thousands of articles in which algorithms are proven correct, software faults are more rule than exception.
People that do implement correct algorithms know from experience that they find faults in their software during testing. Hence, software testing must be superior over proofs in practice.
By stigmatizing software testing as a technique for the ignorant, the preachers of correctness proofs are doing themselves a disservice. Rather should one offer techniques for testing implementations of correct algorithms. This may increase the adoption of correct algorithms by industry.

1 Introduction

Research in formal methods is progressing and more and more success stories are presented in which critical software components are developed using formal methods. There is a wide variety of formal methods, from type systems, to static analysis tools, model checkers and theorem provers. Each with its own success stories and area of applicability; combining techniques seems even more promising for practical purposes. In particular distributed algorithms "need a formal proof to ensure correctness" [1].

Software developers today have support from formal method tools and the programming languages they use have changed over the years. Using a functional language like Haskell [2] or Erlang [3] simply removes the possibility to introduce certain faults in the developed software. A language like Agda [4] has such an advanced type system that writing a program is similar to producing a proof for it. Then, why do we not write all our programs together with a formal proof of correctness?

Even if the competence of the software developers is no barrier and even if the cost of doing it wrong justifies spending the resources to get it right, writing formal proofs at the same time as writing the program is economically unwise, because the requirements are often unclear and some experimentation is needed to get the requirements right. Prototyping a program and validating it against the problem domain requires a number of iterations, **producing proofs for the prototypes one discards is simply too costly** (cf. [5–9]).

Assume then that we do the prototyping with a combination of languages and techniques to get a number of the non-functional requirements right and get a clearer understanding of the functional requirements. Admittedly there is a temptation to quickly earn back the investment in the prototype by patching and releasing it as a product, but then we do get the kind of software we see in the market, containing an irritating amount of defects.

With precise requirements we should be able to develop software without defects. In fact, for many algorithmic problems there exist books and research papers that describe possible solutions. Software engineering is then finding the right solutions, combine them in a system and write additional code that shuffles around the data or provides an interface in the way the customers like it.

For the benefit of future generations, algorithms are not presented as source code in a specific programming language, but on a higher level of abstraction. This requires at least one translation step towards a programming language when implementing these algorithms. Moreover, there are many different algorithms addressing the same problem. For example, one can easily find 500 different articles on leader-election algorithms all solving the same problem with slightly different assumptions about the context [10, 11]. It is often a **non-trivial task** to (1) **find the right algorithm**, (2) to **adapt that algorithm** to the specific setting at hand. Doing it wrong will introduce a fault in the resulting software.

If we were able to use the proofs to automatically generate test cases for the implementation programming language at hand, then any implementation of the algorithm could quickly be tested and many defects would immediately be revealed. Property Based Testing [12] provides exactly that solution. The properties are obtained from the formal proofs and QuickCheck [13, 14] is used to automatically generate and execute test cases from these properties

In this paper this idea is illustrated by implementing a distributed algorithm in Erlang. We have a network of half a million distributed computers, each hosting a number of game players. In order to reduce the load, the server with the maximum number of playing users should refuse new users access. A biassed search for a correct algorithm has narrowed us to an article[1] by Lentfert and Swierstra [15].

2 Finding the right algorithm

At a first glance, this requirement seems so simple that we hardly realize that there might be need for a sophisticated solution. At a second glance the requirement is completely incorrect and insane: the maximum will be zero from the beginning and we will never allow any new user on any machine. Welcome to reality, this is how requirements are given in industry. The requirement needs to be refined. Instead of using the maximum to block new users, the maximum is input to a decision algorithm that may instruct nodes to reject additional login requests. Given that the majority of the errors will be made in the decision algorithm, we are happy to only have to provide a correct implementation of an algorithm that reports the maximum number of users logged in to any node.

The following paragraph in our chosen article seems in line with our goal:

> Assume that every node in a network has a data-item, called its *input*. Inputs are not necessarily unique elements, and are taken from a totally ordered set. The input of a node may change spontaneously. This section formally derives and verifies a distributed algorithm to maintain the maximum of all inputs in the network. [15]

Each node, or game server, has the number of players as input. Numbers are totally ordered. We know that values are not unique and that they change by login and logout behaviour.

It is tempting to immediately look at the algorithm and implement it, given this seemingly perfect match. However, in a less biassed search we would have selected at least twenty

[1] Our position in industry prevents us access to the published article, but the technical report, with all the proofs can be downloaded for free.

different articles, all solving this problem and we need to inspect the assumptions that these algorithms pose on the environment before we can decide whether we have the right candidate (cf. [10]). Let us have a look at the assumptions in this article.

Node requirements

The first assumption is given in Section 4 of the article:

> If the state of a node a depends on the state of node b ($b \neq a$), the state of b does not depend on the state of a. [15]

What does this mean? Surely, the game servers are reasonably autonomous, but some interaction takes place between them. Since the software on the game servers is identical, this condition can be interpreted by a demand that the state of one server may never depend on the state of the other server, since if so, it also is the other way around. For example, a user playing on node b cannot login to node a. Thus, the state of a depends in that sense on the state of b. However, the state of b is for that particular playing user not depending on the state of a, but it was depending on the state of a when allowing the user to start playing.

Without understanding where in the formal proof this assumption is used, it is extremely hard to see whether our particular situation fulfills this requirement.

Network requirements

In Section 5.1 some further assumptions are made about the network that connects our game servers.

> Every node $a \in N$ corresponds to a processor in the network. Moreover, every node can be uniquely identified .... Every edge $\{a, b\} \in E$ corresponds to a (bidirectional) link between processors a and b. These links are assumed to be fault-free and have a FIFO behaviour. Furthermore, it is assumed that the graph is *connected*. [15]

Most game servers use a multi-core processor; how critical is that in this context? Given that the Erlang virtual machine hides the notion of processor for us and internally handles the multi-core aspect, do we really need to verify this assumption? Every node can be uniquely identified. There is both a notion of IP address as well as that Erlang nodes do have a unique identifier. Thus both on the concrete level as well as on the more abstract level, we do have this uniqueness. What does it mean to have bidirectional fault-free links? An Erlang programmer would immediately respond that the nodes are indeed linked and that the communication is fault-free. However, faults may occur, processors may fall over and networks may be partitioned (contradicting the connected graph assumption). Is the assumption on FIFO links actually a requirement on an underlying assumption that one has communicating channels and that these channels are FIFO?

Here it seems that the algorithm is insufficient for our purpose. Game-servers do fail now and then and the network is not reliable in the sense that we always have a connected graph. But under normal operation, all nodes are up and can communicate with each other. A common way of thinking here is: let us implement the algorithm and try to understand what happens in those cases that these assumptions are violated. We can adapt the algorithm a bit to take care of those situations. This is where software errors are going to be introduced.

Global variables in a distributed system

Something that is always a bit puzzling for the implementer of distributed algorithms is the fact that the proofs have global variables.

> There is a variable $m(G)$; it is intended to hold the maximum of the inputs. [15]

Where is this variable located? We must assume a special node that collects the maximum in which this variable is stored in memory. We are not using a transputer having all nodes read and write in the same memory space. Is this an assumption requiring a global memory?

Specification

The formal specifications of correctness for the algorithm is presented as:

$$(\Diamond\Box(nodesval(G) = \overrightarrow{M})) \Rightarrow (\Diamond\Box(m(G) = \langle \texttt{max}\ \ a : a \in N :: inp(a)\rangle))$$

Is this the property that we want to hold for our software? In text it is explained as:

> Given an indexed set of inputs, eventually the algorithm terminates (and thus "delivers" the maximum input) or the indexed set of inputs changes. And once the algorithm terminates, this remains to hold until at least one input changes. [15]

This is where the article clearly adds value for the software engineer. Here one realizes that once again the requirement hampers, since in a constantly evolving system, 'the maximum number of players' does only exists for a short time interval and the requirement is not explicit in talking about that time interval. One realizes that the algorithm at once has a timing requirement, it should be able to compute this maximum rather fast, such that login and logout is an extremely slow operation compared to the computation of the maximum. With half a million servers, this might be an unrealistic assumption. Alternatively the requirement has to be formulated in another way, since at the moment it is not precisely defined what maximum number of users actually means. Time to talk to the customer and raise the price.

On the other hand, the article defines a more precise notion of termination in Section 5.2.1 where a hierarchy is imposed on nodes (spanning tree) with a termination criterion per node:

> The algorithm has terminated for node a if and only if the algorithm has finished for a and this value of a is known to the father of a. [15]

That means that we need to define a spanning tree over half a million nodes and it is at the moment unclear whether it matters which spanning tree we take. Does a choice for a deep tree versus a broad tree have impact that we need to communicate to the customer?

Reading this article, or any other proposing a solution to a challenge one has, reveals requirements one has not thought about. It is useful to get additional understanding of the problem domain as well as that it raises additional, hard to answer questions. Assumptions made are not clearly enough explained. The chosen article is just an example of many articles we have analyzed over the years where it was unclear what concrete requirements were posted on the underlying hardware, network and communication. Although we are uncertain whether our internet game-servers fulfil all assumptions needed, we would like to proceed, implement the proposed distributed algorithm and test it to see if it fulfils the customer's needs.

3 Implementation

The article [15] argues that the specification above results in two main properties and that after refinement of these properties, the algorithm trivially follows. An ideal case for Property Based Testing using these properties.

Let us implement the algorithm in Erlang and then test that the properties hold by using QuickCheck. The implementation provided by the article is given in UNITY [16], of which the semantics is shortly described in Section 2 of the article we study.

```
Program
  assign
    ⟨ | a, f : a ◁ f ::
      m^f(a) := m^a(a)
      |
      m^a(a) := max {inp(a), ⟨max b : b ◁ a :: m^a(b)⟩}
    ⟩
end
```

Note that this is the algorithm for one node a where the father node is f. The spanning tree is assumed to be given outside the algorithm. We interpret this as the algorithm being correct for any given spanning tree.

We need to interpret this specification as non-experts in UNITY. We choose to represent node a by an Erlang process in an Erlang node named a. The expression $\langle \texttt{max}\ b : b \lhd a :: m^a(b)\rangle$ means that the process on node a computes the maximum of all values it has been provided with from siblings b. If a has no siblings, then value is $-\infty$, according to the semantics. The article explains:

> Uninitialize variables have arbitrary initial values.

which would indicate that the variables $m^a(b)$ and therefore the maximum can start with having any value. We are puzzled by that behaviour.

In parallel with that we receive the values from all b's we can compute the maximum for a by the maximum so far and the total number of players on node a. In parallel with that, we communicate the value to our father. We translate this directly to Erlang without the normal additions we need to do in production code to allow for stopping the processes, hot-code updates, error handling, *etc.* We also change the language used in the article to a nowadays more common English.

```
loop(Parent,Max) ->
  Parent ! {update,Max},
  receive
    login -> loop(Parent,Max+1);
    logout -> loop(Parent,Max-1);
    {update,M} ->
      loop(Parent,max(Max,M))
  end.
```

Lst. 1.1. Erlang translation of UNITY program

Our Erlang nodes are real distributed nodes and they cannot share the same memory. Therefore, we have chosen to communicate via messages. The UNITY $inp(a)$ is the input of the node and is changed by either receiving a `login` message or a `logout` message. These messages are produced by the process handling login and logout events. The process starts

sending its current maximum to the parent. Whenever that maximum changes because of an event, the infinite recursive loop will again forward the message to the parent. The UNITY values $m^a(b)$ are messages from the children to the parent of the form {`update,M`} where `M` is the variable containing the new maximum. We made the choice here to immediately update the parent, even before we have received any change. The alternative is to send as soon as we have received a new value. Yet another optimization would be to send only when the maximum has changed.

In production code one would complicate this with more robustness, making sure that the process does know its children, such that when these crash, the parent can take action. Also, the update message can now come from everywhere, not only from the children. Keeping track of the children and extending the message with a unique child identifier would certainly be done in real code. Moreover, one would most likely extend this program to also communicate the name of the node that reaches the maximum, since that's probably useful information.

On purpose we keep this program as close as possible to the code in the article. The code in the article is correct and therefore this code should be correct. However, an important part is missing, how do we start the code using the spanning tree. There is no UNITY code for that in the article, but without starting this on all nodes, we cannot use the program. Therefore, we must write some more Erlang code. In a distributed system of Erlang nodes, the build-in function `nodes()` returns the set of all nodes the present node is connected to (not including itself). To keep things simple, we assume all these connected nodes to run a game server. We hard-code the spanning tree by a binary tree; starting N nodes is performed by starting the parent with maximum two children 'equally' dividing the remaining $N-1$ nodes:

```
start(Parent,[Node|Nodes]) ->
  {Left,Right} = lists:split(length(Nodes) div 2,Nodes),
  spawn(Node, ?MODULE, init, [Parent,[Left,Right]]).

init(Parent,SubTrees) ->
  [ start(self(),Tree) || Tree<-SubTrees, Tree /= []],
  register(algorithm,self()),
  loop(Parent,0).
```

Lst. 1.2. Starting an Erlang process on each node

We split the tail of a list of nodes in two parts, giving us the children for the head node. We spawn a process on the head node that starts two children provided that there are children in the sub-tree. These started children get the process identifier of the parent (evaluated by `self()`) as input. After that, the head process executes the translation of the UNITY code as provided in Lst. 1.1 with starting maximum zero.

The only noteworthy part of this start-up code is the registration of the process identifier in the `init` function. This makes calling by name possible; we use this to be able to send the main loop the messages `login` and `logout`. These messages come from an external process and either we write a lot of code to communicate the correct process identifier to these processes or we hard code a call to the named process {`algorithm, node()`} `! login` would send a login message to the registered process `algoritm` on the current node.

4 Testing the implementation

Even though the Erlang implementation is obviously correct by just looking at it, we might have introduced an error in this code. After all, we have translated UNITY to Erlang,

interpreted a number of language constructs, violated a number of assumptions made in the article and added some code for starting and for manipulating the input. In reality, the code would not even resemble the Erlang code presented here, but be far more complex, just because standardized patterns would be used. Repeating the proof would require the Erlang Theorem Prover [17], but that seems inefficient use of time for an already proven algorithm. Instead, we want to test the properties provided by the article to ensure ourselves that we made no stupid mistakes[2].

At this moment, the software engineer would like to see properties in the code with some remark like: *check these properties and your implementation is correct.* We extract the two main properties of the article:

1. When every descendant of node a has successfully computed its value, then within finite time the father of a is notified of the maximum value of the inputs at the descendants of a or at least one descendant of a must recompute its value.
2. And when every descendant of a has successfully recomputed its value and the father of a is aware of the value of a, then this continues to hold until at least one descendant of a must recompute its value.

There are many ways of writing these properties in QuickCheck. Most suitable probably would be to use the temporal relations library [18] for specifying these properties. However, we choose to use a simplification of the above properties: If we start the software and do not produce any login or logout, then the parent node should after finite time have computed value zero.

In the activity of proving correctness, one needs to formulate properties general and strong enough to conclude correctness from. In contrast to that, one can very easily find defects by formulating one test case or a simple property that turns out not to hold. Thus, testing against a simpler property is fine, as long as one detects defects and fixes those. If no defects are detected, then one needs to improve testing by a more advanced property.

We test this property by starting our program on a subset of all nodes, such that we test for more than one fixed scenario. In fact we have 10 nodes configured in our test set-up from which we take an arbitrary non-empty subset to start the test with. After starting the programs on the selected nodes, we wait for 1 second (extremely long in our test setting) during which we collect all messages arriving at the root node. The number obtained here should be zero. We also stop the programs that we started at the end of the test.

```
prop_no_action() ->
  ?FORALL(Nodes,non_empty(subset(nodes())),
          begin
            start(self(),Nodes),
            Max = read_last_message(0),
            [ stop(Node) || Node<-Nodes ],
            equals(Max,0)
          end).

read_last_message(Max) ->
  receive
    {update,M} ->
      read_last_message(M)
  after 1000 ->
      Max
```

[2] When implementing this simple program some mistakes were actually made, only revealed by compiling and testing the code.

```
  end.

stop(Node) ->
  {algorithm,Node} ! {stop,self()},
  receive
    {stopped,Node} ->
      ok
  end.
```

Lst. 1.3. QuickCheck property for nodes without activity

Since we run a number of tests after each other, we need to stop the program and restart it! The Erlang primitive to kill processes is a tricky one to use in such a property, since it is an asynchronous operation. We can only be sure we stopped a process before the next test, if we synchronize with it. This requires an addition to the code under test, normally not something to strive after, but in this case, we may need that functionality in any serious implementation of this algorithm anyway. We add the possibility to stop the implementation of the algorithm:

```
loop(Parent,Max) ->
  Parent ! {update,Max},
  receive
    login -> loop(Parent,Max+1);
    logout -> loop(Parent,Max-1);
    {update,M} ->
      loop(Parent,max(Max,M));
    {stop,Pid} ->
      unregister(algorithm),
      Pid ! {stopped,node()}
  end.
```

Lst. 1.4. Erlang translation of UNITY program with stop feature

QuickCheck finds no errors when testing this property, which is re-assuring. Now let us strengthen the property a bit and have some activity that changes the input; that was the whole idea of the algorithm. Again we go for a simple property and generate an arbitrary sequence of login messages for the different nodes. We start the program on all nodes, send login messages, wait a while at the root node and then report the findings. We compute the expected value by counting the occurrences of the each node in the sequence of logins, or zero if the sequence is empty (therewith making the first property a special case of this property).

```
prop_login() ->
  ?FORALL(Nodes,non_empty(subset(nodes())),
     ?FORALL(Seq, list({login,elements(Nodes)}),
       begin
         start(),
         [ {algorithm,Node} ! Action || {Action,Node}<-Seq ],
         Max = read_last_message(0),
         [ stop(Node) || Node<-nodes() ],
         equals(Max,count_max([ Node || {_,Node}<-Seq]))
       end)).
```

Lst. 1.5. QuickCheck property for nodes with login events

Now, this property represents how the implemented algorithm would be used in a real situation. The code is loaded on a number of nodes. Events are spontaneously happening,

since we are working with a large system and have no control over login and logout. We wait long enough to have the algorithm stabilize and nevertheless we get the following QuickCheck counter example:

```
(main@ThomasMacBook)3> eqc:quickcheck(maxnode:prop_login()).
...Failed! After 4 tests.
['3@ThomasMacBook']
[{login,'3@ThomasMacBook'}]
0 /= 1
Shrinking.(1 times)
['1@ThomasMacBook']
[{login,'1@ThomasMacBook'}]
0 /= 1
false
```

With only one node, performing one login, we receive as maximum 0, whereas we really expects 1 in this case. What went wrong? Is the proven algorithm defect after all? No, not really. This is distributed software. We spawn new processes in all existing nodes. In Erlang, the spawn function is non-blocking, it immediately returns a process identifier. That means that we can hit a scenario in which we spawn the first child `1@ThomasMacBook` and before that child has even got the time to register itself, we continue executing the sequence of logins. The first message is sent to a process named `algorithm` on the first node. According to the Erlang semantics [19], messages sent to non-existing processes are just silently removed. Thus, the login never arrives due to a race condition in the usage of the code!

In some sense we have been lucky to find this race condition with so few tests with a normal QuickCheck property. In many circumstances it takes far more to detect these race conditions: from the test program taking control over the scheduler [20, 21], to using model checking for all possible scenarios [22].

The solution is to either wait long enough to spawn these processes on half a million servers or, equally bad, to synchronize the start-up. No matter which solution we choose, we are outside the scope of the article with the correctness proof and therefore, any solution we choose can only result in the necessity to proof correctness of our program ourselves, not being able to build upon the existing knowledge.

5 Conclusion

In this paper it is demonstrated how difficult it can be for a software engineer to find and implement a correct algorithm for a given problem. Even a seemingly simple problem with a seemingly simple solution described in a well written article, is a challenge to translate to a different context.

We want to stress that the problems we describe in this paper are a consequence of the necessary translation needed from theory to practice. We picked a specific article to demonstrate the problems, but it is not due to that specific article that these problems arise. We have similar experience with implementing algorithms from many other articles.

What we argue is that, when implementing an algorithm from an article, a software engineer can benefit from the description of the algorithm itself and from the assumptions made to getter a bet understanding of requirements as well as solution space. However, producing an implementation of correct software from an algorithm that is proven to be correct is non-trivial.

Testing plays a major role in getting from an algorithm to a correct implementation of that algorithm. It took a few iterations before we could come up with the Erlang code presented in this article. In fact, many more iterations than presented. Had we immediately tried to prove all these iterations to be correct, then we would have had to abandon most of these proofs. However, the properties we used for our tests are easier to create and more likely to be re-usable.

Due to the importance of testing, it is beneficial to the community if clear properties are provided in the articles. Properties that can be used in combination with Property Based Testing to automatically generate test cases.

References

1. Lentfert, P., Swierstra, S.: Towards the formal design of self-stabilizing distributed algorithms. STACS 93 (1993) 440–451
2. Jones, S.P.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
3. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
4. Norell, U.: Dependently typed programming in agda. In Koopman, P., Plasmeijer, R., Swierstra, D., eds.: Advanced Functional Programming. Volume 5832 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 230–266
5. Arts, T., Dam, M.: Verifying a distributed database lookup manager written in Erlang. In Wing, J.M., Woodcock, J., Davies, J., eds.: World Congress on Formal Methods. Volume 1708 of Lecture Notes in Computer Science., Springer (1999) 682–700
6. Arts, T., Earle, C.B., Derrick, J.: Development of a verified Erlang program for resource locking. STTT **5**(2-3) (2004) 205–220
7. Arts, T., van Langevelde, I.: Correct performance of transaction capabilities. In: ACSD, IEEE Computer Society (2001) 35–42
8. Arts, T., Claessen, K., Svensson, H.: Semi-formal development of a fault-tolerant leader election protocol in Erlang. In Grabowski, J., Nielsen, B., eds.: FATES. Volume 3395 of Lecture Notes in Computer Science., Springer (2004) 140–154
9. Svensson, H.: Verification of Distributed Erlang Programs using Testing, Model Checking and Theorem Proving. Chalmers University (2008)
10. Arts, T., Claessen, K., Hughes, J., Svensson, H.: Testing implementations of formally verified algorithms. In: Proc. of the fifth conference on Software Engineering Research and Practice in Sweden. (2005)
11. Svensson, H., Arts, T.: A new leader election implementation. In: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang. ERLANG '05, New York, NY, USA, ACM (2005) 35–39
12. Derrick, J., Walkinshaw, N., Arts, T., Earle, C.B., Cesarini, F., Fredlund, L.Å., Gulías, V.M., Hughes, J., Thompson, S.J.: Property-based testing - the protest project. In de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M., eds.: FMCO. Volume 6286 of Lecture Notes in Computer Science., Springer (2009) 250–271
13. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of ACM SIGPLAN International Conference on Functional Programming. (2000) 268–279
14. Arts, T., Hughes, J., Johansson, J., Wiger, U.T.: Testing telecoms software with Quviq QuickCheck. In Feeley, M., Trinder, P.W., eds.: Erlang Workshop, ACM (2006) 2–10
15. Lentfert, P., Swierstra, S.: Towards the formal design of self-stabilizing distributed algorithms. Technical Report RUU-CS 92-25, Utrecht University (1992)
16. Chandy, K.M., Misra, J.: Parallel program design. Reading, MA; Addison-Wesley Pub. Co. Inc. (1989)
17. Fredlund, L.Å., Gurov, D., Noll, T., Dam, M., Arts, T., Chugunov, G.: A tool for verifying software written in erlang. International Journal on Software Tools for Technology Transfer (STTT) **4**(4) (2003) 405–420

18. Hughes, J., Norell, U., Sautret, J.: Using temporal relations to specify and test an instant messaging server. In: 5th Workshop on Automation of Software Test, New York, NY, USA, ACM (2010) 95–102
19. Claessen, K., Svensson, H.: A semantics for distributed erlang. In: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang. ERLANG '05, New York, NY, USA, ACM (2005) 78–87
20. Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.: Finding race conditions in Erlang with QuickCheck and PULSE. In: 14th ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM (2009) 149–160
21. Arts, T., Hughes, J., Norell, U., Smallbone, N., Svensson, H.: Accelerating race condition detection through procrastination. In: Proceedings of the 10th ACM SIGPLAN workshop on Erlang. Erlang '11, New York, NY, USA, ACM (2011) 14–22
22. Fredlund, L.A., Svensson, H.: McErlang: a model checker for a distributed functional programming language. In: 12th ACM SIGPLAN International Conference on Functional Programming. (October 2007)

Accumulating Attributes

for Doaitse Swierstra, on his retirement

Jeremy Gibbons

Department of Computer Science, University of Oxford
http://www.cs.ox.ac.uk/jeremy.gibbons/

Abstract. Doaitse has always been enthusiastic about attribute grammars, seeing them even where most people don't. While I was writing up my DPhil thesis, he explained to me where they were in that too. This paper is by way of belated thanks for that perspective—and as a more general rendering of the observation, with the benefit of hindsight and twenty years of progress.

1 Introduction

I first met Doaitse Swierstra and became exposed to his predilections at the STOP project summer school on Ameland in 1989. For as long as I have known him, Doaitse has been devoted to attribute grammars—in compiler technology, in program design, and simply in structuring his thought processes.

I next visited the Netherlands in the summer of 1991. I was deep in the middle of writing up my DPhil thesis [1] on upwards and downwards accumulations on trees, and during my trip I gave a couple of seminars about my work, including one at Utrecht hosted by Doaitse. As many who have met him will attest, Doaitse is like a Frenchman: whatever you say to him, he translates into his own language, and forthwith it is something entirely different. In this case, he immediately explained to me that my thesis was really about attribute grammars, and not about accumulations at all.

In a sense, he was right. Indeed, his observation grew into the final chapter of my thesis, which attempted to explain the connection. At the time of my visit, I was looking for one more substantive piece of work to balance out what I already had, so this contribution was very welcome. But I'm not sure that I ever thanked him properly for his gift, and this is my opportunity to do so. Besides, I can now tell the story much better than I could twenty-odd years ago: in particular, I can do so datatype-generically.

So, thank you, Doaitse, for your ever fresh way of looking at the world!

2 Origami programming

One of the advances in functional programming since I did my thesis has been a much better understanding of what I have been calling 'datatype-generic' [2] and 'origami' [3] programming—that is, the use of structured recursion operators such as maps, folds, and unfolds, parametrized by the shape of data. These ideas were developing at the time, in the work of Hagino [4] and Malcolm [5], but I for one didn't really appreciate them until some years later. With the wisdom of hindsight, the results in my thesis can be presented in this style too.

We will discuss attribute grammars in terms of labelling every node of a data structure. So for simplicity, in this paper we stick to labelled data structures—every node has precisely one label, of type a, and an f-structure of children:

$$\textbf{data}\ \mathit{Mu}\ f\ a = \mathit{In}\ \{\mathit{root} :: a, \mathit{kids} :: f\ (\mathit{Mu}\ f\ a)\}$$

The datatype $\mathit{Mu}\ f$ supports numerous datatype-generic origami operations, parametrized by the shape function f. Of these, we will only make use in this paper of folds, not unfolds or other more sophisticated origami operators:

$$\begin{aligned}&\mathit{fold} :: \mathit{Functor}\ f \Rightarrow (a \to f\ b \to b) \to \mathit{Mu}\ f\ a \to b\\&\mathit{fold}\ \phi\ t = \phi\ (\mathit{root}\ t)\ (\mathit{fmap}\ (\mathit{fold}\ \phi)\ (\mathit{kids}\ t))\end{aligned}$$

We'll also need ad-hoc datatype genericity; so we define a universe of polynomial functors.

$$\begin{aligned}&\textbf{data}\ \mathit{Unit}\ a &&= \mathit{Unit}\\&\textbf{data}\ \mathit{Id}\ a &&= \mathit{Id}\ a\\&\textbf{data}\ \mathit{Sum}\ f\ g\ a &&= \mathit{Inl}\ (f\ a) \mid \mathit{Inr}\ (g\ a)\\&\textbf{data}\ \mathit{Prod}\ f\ g\ a &&= f\ a :\times: g\ a\end{aligned}$$

Here also is the empty datatype—it will be useful for some constructions, although we won't consider it part of our universe of functors:

$$\textbf{data}\ \mathit{Zero}\ a$$

There are no constructors for *Zero*, so no (proper) values of that type; therefore, if you were somehow to be able to come up a value of type *Zero*, you deserve to be able to turn it into anything you want:

$$\begin{aligned}&\mathit{magic} :: \mathit{Zero}\ a \to b\\&\mathit{magic}\ z = \mathit{seq}\ z\ (\mathit{error}\ \texttt{"It must be magic"})\end{aligned}$$

All these codes represent functors, of course:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ \mathit{Zero}\ \textbf{where}\\&\quad \mathit{fmap}\ f\ z &&= \mathit{magic}\ z\\&\textbf{instance}\ \mathit{Functor}\ \mathit{Unit}\ \textbf{where}\\&\quad \mathit{fmap}\ f\ \mathit{Unit} &&= \mathit{Unit}\\&\textbf{instance}\ \mathit{Functor}\ \mathit{Id}\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathit{Id}\ a) &&= \mathit{Id}\ (f\ a)\\&\textbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (\mathit{Sum}\ f\ g)\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathit{Inl}\ x) &&= \mathit{Inl}\ (\mathit{fmap}\ f\ x)\\&\quad \mathit{fmap}\ f\ (\mathit{Inr}\ y) &&= \mathit{Inr}\ (\mathit{fmap}\ f\ y)\\&\textbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (\mathit{Prod}\ f\ g)\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (x :\times: y) = \mathit{fmap}\ f\ x :\times: \mathit{fmap}\ f\ y\end{aligned}$$

For example, *TreeF* is the code for the shape functor of homogeneous binary trees, and *sum* is an example fold for that datatype:

$$\begin{aligned}&\textbf{type}\ \mathit{TreeF} = \mathit{Sum}\ \mathit{Unit}\ (\mathit{Prod}\ \mathit{Id}\ \mathit{Id})\\&\textbf{type}\ \mathit{Tree} \quad = \mathit{Mu}\ \mathit{TreeF}\\&\mathit{add} :: \mathit{Int} \to \mathit{TreeF}\ \mathit{Int} \to \mathit{Int}\\&\mathit{add}\ m\ (\mathit{Inl}\ \mathit{Unit}) &&= m\\&\mathit{add}\ m\ (\mathit{Inr}\ (\mathit{Id}\ n :\times: \mathit{Id}\ p)) &&= m + n + p\\&\mathit{sum} :: \mathit{Tree}\ \mathit{Int} \to \mathit{Int}\\&\mathit{sum} = \mathit{fold}\ \mathit{add}\end{aligned}$$

For convenience, here are two 'constructors' for *Tree*:

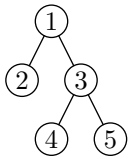


Fig. 1. The tree $t = node\ 1\ (leaf\ 2)\ (node\ 3\ (leaf\ 4)\ (leaf\ 5))$

$$
\begin{aligned}
&leaf :: a \to Tree\ a \\
&leaf\ a = In\ a\ (Inl\ Unit)
\end{aligned}
$$

$$
\begin{aligned}
&node :: a \to Tree\ a \to Tree\ a \to Tree\ a \\
&node\ a\ t\ u = In\ a\ (Inr\ (Id\ t :\times: Id\ u))
\end{aligned}
$$

Figure 1 shows a small tree t, which we will use for examples.

3 Accumulations on lists

My thesis was about upwards and downwards accumulations on trees, which were intended to be analogous to the accumulations (or 'scans') on lists that had proven so fruitful in Bird's work on the Theory of Lists [6–8]. Recall the standard definitions of *tails*, *foldr*, and *scanr* from the Haskell libraries:

$$
\begin{aligned}
&tails :: [a] \to [[a]] \\
&tails\ [\,] = [[\,]] \\
&tails\ x\ = x : tails\ (tail\ x)
\end{aligned}
$$

$$
\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b \\
&foldr\ f\ e\ [\,] \quad\ = e \\
&foldr\ f\ e\ (a : x)\ = f\ a\ (foldr\ f\ e\ x)
\end{aligned}
$$

$$
\begin{aligned}
&scanr :: (a \to b \to b) \to b \to [a] \to [b] \\
&scanr\ f\ e\ [\,] \quad\ = [e] \\
&scanr\ f\ e\ (a : x) = f\ a\ (head\ y) : y\ \textbf{where}\ y = scanr\ f\ e\ x
\end{aligned}
$$

Here, *scanr* computes all the partial results of a fold, from the right:

$$scanr\ (+)\ 0\ [1, 2, 3] = [6, 5, 3, 0]$$

It satisfies a very important 'scan lemma', stating that these partial results are precisely the results of folding each of the tails:

$$scanr\ f\ e = map\ (foldr\ f\ e) \circ tails$$

The scan lemma is crucial in deriving numerous efficient algorithms over lists, not least for the famous 'maximum segment sum' problem [8].

Dually, there are functions that work from the opposite end of the list:

$$
\begin{aligned}
&inits :: [a] \to [[a]] \\
&inits\ [\,] \quad\ = [[\,]] \\
&inits\ (a : x) = [\,] : map\ (a:)\ (inits\ x)
\end{aligned}
$$

$$
\begin{aligned}
&foldl :: (b \to a \to b) \to b \to [a] \to b \\
&foldl\ f\ e\ [\,] \quad\ = e
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldl}\ f\ e\ (a : x)\ = \mathit{foldl}\ f\ (f\ e\ a)\ x\\
&\mathit{scanl} :: (b \to a \to b) \to b \to [a] \to [b]\\
&\mathit{scanl}\ f\ e\ [\,]\ \ \ \ \ \ = [e]\\
&\mathit{scanl}\ f\ e\ (a : x) = e : \mathit{scanl}\ f\ (f\ e\ a)\ x
\end{aligned}
$$

for which

$$\mathit{scanl}\ (+)\ 0\ [1,2,3] = [0,1,3,6]$$

Again, there is a scan lemma:

$$\mathit{scanl}\ f\ e = \mathit{map}\ (\mathit{foldl}\ f\ e) \circ \mathit{inits}$$

With these six functions as my inspiration, the essence of my thesis work was to generalize them to trees of various kinds.

4 Upwards accumulations

Generalizing *tails* and *scanr* is the easier task, because tail segments follow the structure of the datatype, whereas *inits* and *scanl* in some sense go against the grain. The idea is that *tails* labels every node of a list with the tail starting at that position, and *scanr* labels every node with the fold of that tail. My thesis made some ad-hoc generalizations of this idea to various kinds of tree. It wasn't until some years later [9] that a datatype-generic construction was discovered, involving a systematic way of deriving a 'labelled variant' for any datatype, which has (precisely) one label at each node. Happily, in this paper we have already restricted ourselves to such datatypes, so we don't need this construction—the 'labelled variant' of Muf is $\mathit{Mu}\ f$ itself.

Then *subtrees*, the datatype-generic version of *tails*, labels every node of a tree with the subtree rooted at that node. The root label of the output is the whole of the input; and each child in the output is generated from the corresponding child in the input.

$$
\begin{aligned}
&\mathit{subtrees} :: \mathit{Functor}\ f \Rightarrow \mathit{Mu}\ f\ a \to \mathit{Mu}\ f\ (\mathit{Mu}\ f\ a)\\
&\mathit{subtrees} = \mathit{fold}\ \psi\ \mathbf{where}\ \psi\ a\ \mathit{ts} = \mathit{In}\ (\mathit{In}\ a\ (\mathit{fmap}\ \mathit{root}\ \mathit{ts}))\ \mathit{ts}
\end{aligned}
$$

An upwards accumulation is like *subtrees*, except that it folds every tree it generates. It does this as it goes, so it takes no longer to compute than a mere fold of the input tree does; which is to say, it records all the partial results already involved in folding the original tree.

$$
\begin{aligned}
&\mathit{scanu} :: \mathit{Functor}\ f \Rightarrow (a \to f\ b \to b) \to \mathit{Mu}\ f\ a \to \mathit{Mu}\ f\ b\\
&\mathit{scanu}\ \phi = \mathit{fold}\ \psi\ \mathbf{where}\ \psi\ a\ \mathit{ts} = \mathit{In}\ (\phi\ a\ (\mathit{fmap}\ \mathit{root}\ \mathit{ts}))\ \mathit{ts}
\end{aligned}
$$

Note that we again have the all-important scan lemma:

$$\mathit{scanu}\ \phi = \mathit{fmap}\ (\mathit{fold}\ \phi) \circ \mathit{subtrees}$$

Figure 2 shows (a) $\mathit{subtrees}\ t$ and (b) $\mathit{scanu}\ \mathit{add}\ t$, where t is the tree in Figure 1.

5 Derivatives of datatypes

Generalizing *inits* and *scanl* is more difficult. The function *inits* labels every node of a list with its list of predecessors, and the datatype-generic version should label every node of a data structure with the ancestors of that node; but the ancestors form a completely different datatype, of linear shape whatever the branching structure of the original. Similarly, a downwards accumulation *scand* will label every node of a data structure with some function

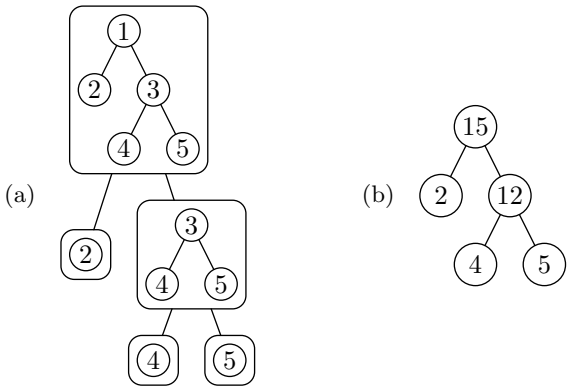


Fig. 2. (a) *subtrees t* and (b) *scanu add t*

of its ancestors—and not just any function, but some kind of fold that will allow us to compute the whole scan in linear time. To capture ancestors, we need to make a detour through derivatives of datatypes [10, 11].

The derivative of a functor f is another functor Δf such that $\Delta f\ a$ is like $f\ a$ but with precisely one a missing: $\Delta f\ a$ is a 'one-hole context' for $f\ a$ with respect to a. We model this idea through a type class *Diff* (of differentiable functors), which has Δ as an associated type synonym:

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Diff}\ f\ \textbf{where}\\ \quad \textbf{type}\ \Delta f :: * \to *\end{array}$$

The class also has two methods, to produce and to consume holes:

$$\begin{array}{l}\mathit{posns} :: f\ a \to f\ (a, \Delta f\ a)\\ \mathit{plug}\ \ :: (a, \Delta f\ a) \to f\ a\end{array}$$

The idea is that *posns* takes a complete piece of data, and labels every element a in it with the one-hole-context for which a completes the original data structure; whereas *plug* takes a one-hole-context and a value to fill that hole, and puts the latter in the former to make a complete piece of data. The two are related by the following two laws (which I believe completely determine the implementation):

$$\begin{array}{l}\mathit{fmap}\ \mathit{fst}\ (\mathit{posns}\ x) \quad = x\\ \mathit{fmap}\ \mathit{plug}\ (\mathit{posns}\ x) = \mathit{fmap}\ (\mathit{const}\ x)\ x\end{array}$$

Informally, the first law states that *posns* really annotates, so that discarding the annotations is a left inverse; and the second that plugging together each pair in the output of *posns* produces many copies of the original data structure, one for each element.

Here are the *Diff* instances for our universe of functors. The constant functor has no elements, so the derivative is the empty type, *posns* has no effect, and you can never have a hole to plug.

$$\begin{array}{l}\textbf{instance}\ \mathit{Diff}\ \ \mathit{Unit}\ \textbf{where}\\ \quad \textbf{type}\ \Delta \mathit{Unit} = \mathit{Zero}\\ \quad \mathit{posns}\ \mathit{Unit} = \mathit{Unit}\\ \quad \mathit{plug}\ (a, z)\ \ = \mathit{magic}\ z\end{array}$$

The identity functor contains precisely one element, so what's left when this is deleted is the unit type:

```
instance Diff Id where
  type ΔId = Unit
  posns (Id a)   = Id (a, Unit)
  plug (a, Unit) = Id a
```

A value of a sum type is either of the left summand or of the right, and in each case a one-hole context is a corresponding one-hole context for that summand; so the two methods simply follow the structure.

```
instance (Diff f, Diff g) ⇒ Diff (Sum f g) where
  type Δ(Sum f g) = Sum (Δf) (Δg)
  posns (Inl x)   = Inl (fmap (λ(a, dx) → (a, Inl dx)) (posns x))
  posns (Inr y)   = Inr (fmap (λ(a, dy) → (a, Inr dy)) (posns y))
  plug (a, Inl x) = Inl (plug (a, x))
  plug (a, Inr y) = Inr (plug (a, y))
```

A value of a product type is a pair, and a one-hole context for the pair is either a one-hole context for the left half, together with an intact right half, or an intact left half and a context for the right half.

```
instance (Diff f, Diff g) ⇒ Diff (Prod f g) where
  type Δ(Prod f g) = Sum (Prod (Δf) g) (Prod f (Δg))
  posns (x :×: y)          = (fmap (λ(a, dx) → (a, Inl (dx :×: y))) (posns x)) :×:
                             (fmap (λ(a, dy) → (a, Inr (x :×: dy))) (posns y))
  plug (a, Inl (dx :×: y)) = plug (a, dx) :×: y
  plug (a, Inr (x :×: dy)) = x :×: plug (a, dy)
```

For example, consider the type *TreeF Int*, whose values are either a unit or a pair of integers. The derivative *ΔTreeF Int* of this type represents data structures with one missing integer. Expanding the definitions from the type class instances, we see that

```
ΔTreeF = Sum Zero (Sum (Prod Unit Id) (Prod Id Unit))
```

The left-hand variant of this sum is void: corresponding *TreeF* values are just a unit, and there is no way for such a value to be missing an integer. The right-hand variant is itself a sum: corresponding *TreeF* values have two integers, so there are two ways for such a value to be missing an integer, and in each case what remains is the other integer.

Here is a piece of data in the right-hand variant of *TreeF Int*:

```
u :: TreeF Int
u = Inr (Id 3 :×: Id 4)
```

Here are two one-hole contexts for u, in each case having the unit value in place of one of the integers:

```
v1, v2 :: ΔTreeF Int
v1 = Inr (Inl (Unit :×: Id 4))
v2 = Inr (Inr (Id 3 :×: Unit))
```

If you plug the correct integer back into each context:

```
u1, u2 :: TreeF Int
u1 = plug (3, v1)
u2 = plug (4, v2)
```

then you get the original data back again: $u = u_1 = u_2$.

5.1 Zippers

Incidentally, derivatives are intimately connected with *zippers* [12], which represent a data structure with a single subterm highlighted as a 'focus'. Concretely, a zipper is a pair. The first component is the subterm in focus. The second component is the remainder of the data structure, expressed as a sequence of layers, like an onion, innermost layer first; each layer is the one-hole context into which the structure inside fits.

$$\textbf{type } \mathit{Zipper}\ f\ a = (\mathit{Mu}\ f\ a, [(a, \Delta f\ (\mathit{Mu}\ f\ a))]) \quad \text{-- innermost layer first}$$

To reconstruct the complete data structure from the zipper, we plug subterms into contexts, from the inside out:

$$\begin{aligned}&\mathit{close} :: \mathit{Diff}\ f \Rightarrow \mathit{Zipper}\ f\ a \to \mathit{Mu}\ f\ a\\ &\mathit{close}\ (x, ds) = \mathit{foldl}\ \mathit{glue}\ x\ ds\ \textbf{where}\ \mathit{glue}\ x\ (a, d) = \mathit{In}\ a\ (\mathit{plug}\ (x, d))\end{aligned}$$

For example, consider a little tree t_1 and two surrounding contexts tc_2, tc_3:

$$\begin{aligned}&t_1 :: \mathit{Tree}\ \mathit{Int}\\ &t_1 = \mathit{leaf}\ 4\end{aligned}$$

$$\begin{aligned}&tc_2, tc_3 :: (\mathit{Int}, \Delta\,\mathit{TreeF}\ (\mathit{Tree}\ \mathit{Int}))\\ &tc_2 = (3, \mathit{Inr}\ (\mathit{Inl}\ (\mathit{Unit} :\times: \mathit{Id}\ (\mathit{leaf}\ 5))))\\ &tc_3 = (1, \mathit{Inr}\ (\mathit{Inr}\ (\mathit{Id}\ (\mathit{leaf}\ 2) :\times: \mathit{Unit})))\end{aligned}$$

The tree in Figure 1 can be reconstructed from these: $t = \mathit{close}\ (t_1, [tc_2, tc_3])$.

6 Downwards accumulations

Now, the ancestors of an element in a data structure form a path to that element, and paths are a projection of zippers. They omit the subterm in focus; they also omit all non-ancestors (siblings, great-aunts, etc) of the focus too, so the type parameter to Δf is the unit type. We call such values 'directions', because they state which direction to take in a parent to get to one of its children.

$$\textbf{type } \mathit{Dir}\ f = \Delta f\ ()$$

For example, the derivative of *TreeF* is a sum type, and so there are different possible directions for each variant. However, as we have already seen, the left-hand summand of $\Delta\,\mathit{TreeF}$ is *Zero*, indicating that there is no direction you can turn to get to a child of a leaf. The right-hand summand is another sum type, and this yields two possible directions to turn for a child of an internal node.

$$\begin{aligned}&\mathit{left}, \mathit{right} :: \mathit{Dir}\ \mathit{TreeF}\\ &\mathit{left}\ \ = \mathit{Inr}\ (\mathit{Inl}\ (\mathit{Unit} :\times: \mathit{Id}\ ()))\\ &\mathit{right} = \mathit{Inr}\ (\mathit{Inr}\ (\mathit{Id}\ () :\times: \mathit{Unit}))\end{aligned}$$

Each node of a $\mathit{Mu}\ f\ a$ data structure has a label of type a and an f-structure of children; so a path to a node (including the root label of the target node) consists of an alternating sequence of elements a and directions $\mathit{Dir}\ f$, with one more element than direction. We represent this sequence innermost-first, so that the path to a child has as a subterm the path to its parent.

$$\textbf{data } \mathit{Path}\ f\ a = \mathit{Start}\ a \mid \mathit{Step}\ (\mathit{Path}\ f\ a)\ (\mathit{Dir}\ f)\ a$$

For example, the path to the node labelled 4 in t starts with a 1, turns right to meet a 3, then turns left to meet a 4, so it is represented by the expression $\mathit{Step}\ (\mathit{Step}\ (\mathit{Start}\ 1)\ \mathit{right}\ 3)\ \mathit{left}\ 4$.

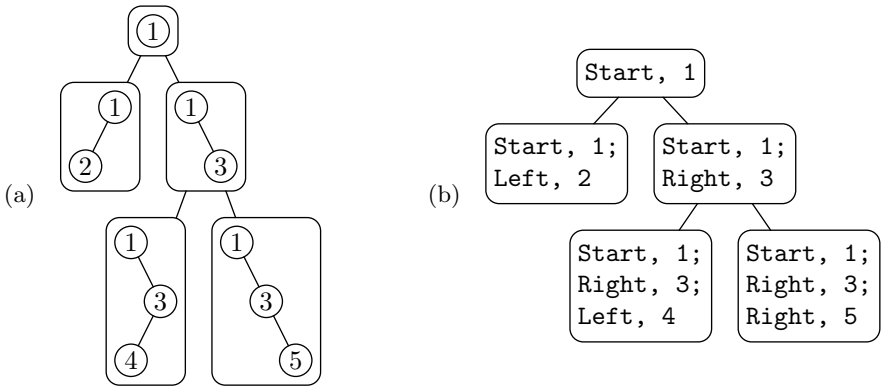


Fig. 3. (a) *paths t* and (b) *routes t* (the latter edited for presentation)

There is, of course, a natural pattern of folds for paths:

$$\begin{aligned}&\mathit{foldPath} :: (b \to \mathit{Dir}\ f \to a \to b) \to (a \to b) \to \mathit{Path}\ f\ a \to b\\&\mathit{foldPath}\ f\ g\ (\mathit{Step}\ p\ d\ a) = f\ (\mathit{foldPath}\ f\ g\ p)\ d\ a\\&\mathit{foldPath}\ f\ g\ (\mathit{Start}\ a) \quad = g\ a\end{aligned}$$

The function *paths* takes a data structure and labels every node with the path from the root to that node:

$$\mathit{paths} :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f)) \Rightarrow \mathit{Mu}\ f\ a \to \mathit{Mu}\ f\ (\mathit{Path}\ f\ a)$$

The definition will have to involve an accumulating parameter: the paths to children depend on surrounding context (the 'path so far') as well as the children themselves. It seems sweetly reasonable to start by unpacking the data structure $\mathit{In}\,a\,ts$ using *posns*, labelling every child t in ts with its one-hole context. We therefore require an auxilliary function *paths′* that should depend on a path so far, initially simply $\mathit{Start}\ a$, and apply to both a child and its one-hole context:

$$\mathit{paths}\ (\mathit{In}\ a\ ts) = \mathit{In}\ p\ (\mathit{fmap}\ (\mathit{paths}'\ p)\ (\mathit{posns}\ ts))\ \textbf{where}\ p = \mathit{Start}\ a$$

$$\begin{aligned}&\mathit{paths}' :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f)) \Rightarrow\\&\qquad \mathit{Path}\ f\ a \to (\mathit{Mu}\ f\ a, \Delta f\ (\mathit{Mu}\ f\ a)) \to \mathit{Mu}\ f\ (\mathit{Path}\ f\ a)\end{aligned}$$

From here, the remainder of the definition is basically driven by the types. We turn the one-hole context z into a direction by discarding siblings, and use that and the node label to extend the path so far for recursive calls.

$$\begin{aligned}&\mathit{paths}'\ p\ (\mathit{In}\ a\ ts, z) = \mathit{In}\ q\ (\mathit{fmap}\ (\mathit{paths}'\ q)\ (\mathit{posns}\ ts))\\&\quad \textbf{where}\ q = \mathit{Step}\ p\ (\mathit{fmap}\ \mathit{bang}\ z)\ a\end{aligned}$$

Here, *bang* is a basic combinator for the unit type:

$$\begin{aligned}&\mathit{bang} :: a \to ()\\&\mathit{bang}\ a = ()\end{aligned}$$

Figure 3(a) shows *paths t*, where t is the tree in Figure 1.

A downwards accumulation is then a fold mapped over the paths:

$$\begin{aligned}&\mathit{scand} :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f)) \Rightarrow\\&\qquad (b \to \mathit{Dir}\ f \to a \to b) \to (a \to b) \to \mathit{Mu}\ f\ a \to \mathit{Mu}\ f\ b\\&\mathit{scand}\ f\ g = \mathit{fmap}\ (\mathit{foldPath}\ f\ g) \circ \mathit{paths}\end{aligned}$$

But because we carefully arranged that the paths to children share a subterm with the path to their parent, we can compute this incrementally in linear time, assuming that the basic operations take constant time. Again, we use an accumulating parameter; but this time, it will be the image under $foldPath\ f\ g$ of the path so far rather than the path itself.

$$scand\ f\ g\ (In\ a\ ts) = In\ b\ (fmap\ (scand'\ f\ g\ b)\ (posns\ ts))\ \textbf{where}\ b = g\ a$$

As with *paths*, the remainder of the definition is type-driven: we turn the one-hole context into a direction by discarding siblings, and use that and the node label to update the accumulating parameter for recursive calls.

$$\begin{aligned}
&scand' :: (Diff\ f, Functor\ (\Delta f)) \Rightarrow \\
&\qquad (b \to Dir\ f \to a \to b) \to (a \to b) \to b \to (Mu\ f\ a, \Delta f\ (Mu\ f\ a)) \to Mu\ f\ b \\
&scand'\ f\ g\ b\ (In\ a\ ts, z) = In\ c\ (fmap\ (scand'\ f\ g\ c)\ (posns\ ts)) \\
&\quad \textbf{where}\ c = f\ b\ (fmap\ bang\ z)\ a
\end{aligned}$$

(In fact, the second argument g of $scand'$ is not used.)

For example, we can produce a little guidebook for a tree, recording in user-friendly format the route to each node in the tree:

$$\begin{aligned}
&routes :: Show\ a \Rightarrow Tree\ a \to Tree\ String \\
&routes = scand\ step\ start \\
&\\
&start\ a = \texttt{"Start, "} +\!\!+ show\ a \\
&step\ s\ (Inl\ z)\ a \qquad\qquad\qquad\qquad = magic\ z \\
&step\ s\ (Inr\ (Inl\ (Unit :\!\times\!: Id\ ()))\ a = s +\!\!+ \texttt{"; Left, "} +\!\!+ show\ a \\
&step\ s\ (Inr\ (Inr\ (Id\ () :\!\times\!: Unit)))\ a = s +\!\!+ \texttt{"; Right, "} +\!\!+ show\ a
\end{aligned}$$

The first clause for *step* is not really needed, because it can never be called; but its presence makes the definition manifestly total. The results are shown in Figure 3(b) (with the strings edited for presentation).

7 Attribute grammars

Having seen datatype-generic definitions of upwards and downwards accumulations, let us now return to the topic of attribute grammars. These were proposed by Knuth [13] as a tool for presenting the semantics of programming languages. They arose as an extension of the 'syntax-directed' compilation techniques of Irons and others in the early sixties [14]. Using these techniques, the parse tree of a program is *decorated* with *attributes*, the decoration attached to an element of the parse tree representing some aspect of the semantics of the subtree rooted there. In Irons' formulation, the attribute attached to an element depends only on the descendants of that element; Knuth showed that although no extra power is gained by doing so, the description of the semantics of a language can be considerably simplified by allowing attributes to depend on other parts of the parse tree as well.

Traditionally, an attribute grammar for a context free language is an extension of the grammar which describes the syntax of that language. Each symbol in the grammar is associated with a number of attributes, and each production in the grammar comes with some rules that give values to some of the attributes attached to symbols appearing in that production, in terms of the values of the other attributes that appear. The attributes are classified into two categories, *inherited* and *synthesized*; inherited attributes are those appearing on the right hand side of the production in which their value is defined, and hence concern the 'children' of the production, whereas synthesized attributes appear on the left, and concern the parents. Irons' syntax-directed translation corresponds to attribute grammars with only

synthesized attributes. Intuitively, inherited attributes carry information into a subtree and synthesized attributes carry it back out again; in Knuth's [15] words, 'inherited attributes are, roughly speaking, those aspects of meaning which come from the context of a phrase, while synthesized attributes are those aspects which are built up from within the phrase.'

Our view of attribute grammars differs somewhat from this traditional view, and follows instead the approach pioneered by Doaitse and his colleagues [16, 17]. We suppose that a tree has been built already, and that the task is simply to evaluate the attributes. That is, attribute grammars are viewed as a means for describing computations over pre-existing trees, rather than in the 'grammar' sense for recognizing or generating those trees in the first place.

We make the simplification, after [18], that every element has exactly one inherited and one synthesized attribute, and that all inherited attributes have the same type i, and all synthesized attributes the same type s. This entails no loss of generality, since attribute types may be product types.

In our datatype-generic formulation, there is just one datatype constructor In, so a single production rule suffices. The production rule takes as input the label (of type a) of a node, the value (of type i) of the sole inherited attribute for that node, and values (collectively of type $f\ s$) for the synthesized attributes of each of the children. It should yield as output the value (of type s) of the sole synthesized attribute of the node, and inherited attributes (collectively of type $f\ i$) for each of the children. We capture this via the following type synonym:

$$\textbf{type}\ \mathit{Rule}\ f\ a\ i\ s = a \to (f\ s, i) \to (s, f\ i)$$

We call a rule $r :: \mathit{Rule}\ f\ a\ i\ s$ 'shape-preserving' if the inherited attributes it generates always match up with the children; that is, for appropriate inputs a, ss, i, if $(s, is) = r\ a\ (ss, i)$ then $\mathit{fmap\ bang\ is} = \mathit{fmap\ bang\ ss}$. We restrict attention in this paper to shape-preserving rules. Of course, 'matching up' will require the ability to zip together f-structures; we will declare a suitable type class *Zippable* with member function *zip* shortly.

7.1 Lazy evaluation and cyclic programs

The usual formulation of attribute grammars is then to compute the synthesized attribute of the root node of a term, given the inherited attribute. Conventionally a lot of effort goes into deducing the dataflow constraints that arise; for example, maybe an inherited attribute of a right child depends on a synthesized attribute of a left sibling, and so a left-to-right traversal is called for. However, as Johnsson [19] observed, in a lazily evaluated language this can all be ignored: the evaluation mechanism automatically works out the appropriate dataflow. So attribute evaluation can be captured as a simple cyclic lazy functional program:

$$\begin{aligned}
&\mathit{eval} :: (\mathit{Functor}\ f, \mathit{Zippable}\ f) \Rightarrow \mathit{Rule}\ f\ a\ i\ s \to (\mathit{Mu}\ f\ a, i) \to s \\
&\mathit{eval}\ r\ (\mathit{In}\ a\ ts, i) = s \\
&\quad \textbf{where} \\
&\qquad (s, is) = r\ a\ (ss, i) \\
&\qquad ss \quad\ = \mathit{fmap}\ (\mathit{eval}\ r)\ (\mathit{zip}\ ts\ is)
\end{aligned}$$

Observe that the inherited attributes is of the children and their synthesized attributes ss are defined using mutual recursion. Statically checking that this recursion is well founded is inherently exponential [20]; indeed, it was one of the first naturally occurring problems to be shown so. In this paper, we assume well-foundedness.

7.2 Matching up

However, one clear necessary condition for well-foundedness is for the shape of *zip ts is* to be determined purely by the shape of *ts*, independent of *is*:

$$fmap\ bang\ (zip\ ts\ is) = fmap\ bang\ (zip\ (fmap\ bang\ ts)\ \bot)$$

This is no hardship, because a given *ts* can be zipped successfully with only one shape of *is*. Operationally, we require a datatype-generic *zip* that is non-strict in its second argument. We introduce a type class of zippable functors:

```
class Zippable f where
  zip    :: f a → f b → f (a, b)
  select :: f a → Δf () → a
```

We have added an operation *select* too, which takes an f-structure of elements and a position in that structure, and selects the appropriate element; we will use this later. All functors in our universe are zippable:

```
instance Zippable Unit where
  zip Unit unit = let Unit = unit in Unit
  select Unit z = magic z

instance Zippable Id where
  zip (Id a) id_b     = let Id b = id_b in Id (a, b)
  select (Id a) Unit = a

instance (Zippable f, Zippable g) ⇒ Zippable (Sum f g) where
  zip (Inl x) inl_y  = let Inl y = inl_y in  Inl (zip x y)
  zip (Inr x) inr_y = let Inr y = inr_y in Inr (zip x y)
  select (Inl x) (Inl d)  = select x d
  select (Inr y) (Inr d) = select y d

instance (Zippable f, Zippable g) ⇒ Zippable (Prod f g) where
  zip (x :×: y) prod_x'_y' = let x' :×: y' = prod_x'_y' in
                                zip x x' :×: zip y y'
  select (x :×: y) (Inl (d :×: y')) = select x d
  select (x :×: y) (Inr (x' :×: d)) = select y d
```

Note the asymmetry in the definitions of *zip*, to ensure non-strictness in the second argument. Of course, *zip* and *select* are partial functions, because the shapes might not match—specifically in the *Sum* case. To be more precise about typing, we could make both methods return a *Maybe* result. However, if we stick to shape-preserving attribute rules, then we will only ever use these two functions on matching shapes.

7.3 Attribute evaluation as a fold

The definition of *eval* above is not obviously in a structured form, since it uses explicit recursion. However, the only use of *ts* is in recursive calls, so it isn't difficult to rearrange the definition into a fold. Of course, children should be processed using different values for the inherited attributes, so it isn't *eval* itself that is a fold. In fact, as many have observed [21–23, 19, 18], it is *curry eval*, computing from a tree a function of type $i \to s$, that is the fold.

```
curryeval :: (Functor f, Zippable f) ⇒ Rule f a i s → Mu f a → i → s
curryeval r = fold φ
```

```
  where φ a fs i = let (s, is) = r a (ss, i)
                       ss      = fmap (uncurry ($)) (zip fs is)
                   in s
```

Conversely, any fold can be formulated as an attribute grammar using only synthesized attributes. From the algebra ϕ for a fold, we can construct an attribute grammar production rule *upRule* ϕ such that *curryeval* (*upRule* ϕ) () = *fold* ϕ, using the unit type for inherited attributes:

```
upRule :: Functor f ⇒ (a → f b → b) → Rule f a () b
upRule φ a (bs, ()) = (φ a bs, fmap bang bs)
```

7.4 Complete attribute evaluation

Attribute evaluation is conventionally understood to mean evaluation of a single attribute, the synthesized attribute of the root of the tree; all the other attributes are 'intermediate results' and are of no further interest. For most applications, and in particular for one-off compilation, this is exactly what is required; once the translation of part of a program has been constructed, the translations of subprograms are no longer needed. However, for some applications we want the intermediate results as well; for example, incremental compilers and structure editors such as the Cornell Synthesizer Generator [24] make use of these intermediate results in order to avoid having to recompile parts of a program that remain unchanged. For such applications, we would like attribute evaluation to return the whole tree of attributes, not just the synthesized attribute of the root.

We have seen that the curried evaluation function *curryeval* is a fold; so *fmap curryeval* ∘ *subtrees*, yielding a tree of inherited-to-synthesized-attribute functions, is an upwards accumulation. This is not quite enough to allow us to compute all the attributes in the tree: given the inherited attribute of the root, we can certainly find the synthesized attribute of the root, but what will the inherited attributes of the children be? We have thrown that information away.

In fact, to support incremental attribute re-evaluation, we should annotate the input tree to record the values of both the inherited and the synthesized attributes at each node. That can be achieved by a slight modification to *curryeval*, reconstructing the input tree as we go, but retaining the inherited attributes throughout. (For completeness, we also preserve the original labels.)

```
annotate :: (Functor f, Zippable f) ⇒ Rule f a i s → Mu f a → i → Mu f (a, i, s)
annotate r = fold (ψ r) where
  ψ r a fs i = In (a, i, s) ts where
    (s, is) = r a (ss, i)
    ts      = fmap (uncurry ($)) (zip fs is)
    ss      = fmap (thd3 ∘ root) ts
```

(This is roughly the concluding construction in [25].)

An illuminating example of an attribute grammar making essential use of both inherited and synthesized attributes is the ranking problem, in which each node is labelled with its position in left-to-right order. This can be expressed as an attribute grammar with one inherited attribute, representing the first index to use, and two synthesized attributes, recording for each node the rank of that node and the size of (that is, the number of elements in) the subtree rooted at that node. For a binary tree, the inherited attribute passed in to a node is propagated to the left child; but the inherited attribute passed to the right child

depends also on the size of the left child. Information flows from left to right, in the same way that it does for a depth-first search. Most of the applications of attribute grammars to programming languages involve dependencies like this, because of the close correspondence between the hierarchical structure of the parse tree and the linear structure of the program text it represents. For our binary tree datatype, we have the following rule:

$$\begin{aligned}
&\mathit{rankRule} :: \mathit{Rule\ TreeF\ a\ Int\ (Int, Int)}\\
&\mathit{rankRule\ a\ (Inl\ Unit}, i) = ((i, 1), \mathit{Inl\ Unit})\\
&\mathit{rankRule\ a\ (Inr\ (Id\ (r_1, s_1)} :\times: \mathit{Id\ (r_2, s_2))}, i)\\
&\qquad\qquad = ((i{+}s_1, 1{+}s_1{+}s_2), \mathit{Inr\ (Id\ i} :\times: \mathit{Id\ (i{+}s_1{+}1)}))
\end{aligned}$$

Then rank ordering is computed by complete attribute evaluation according to this rule, followed by discarding the auxilliary data (the starting index and the size):

$$\begin{aligned}
&\mathit{rank} :: \mathit{Tree\ a} \to \mathit{Tree\ (a, Int)}\\
&\mathit{rank\ t} = \mathit{fmap\ (\lambda(a, i, (r, s)) \to (a, r))\ (annotate\ rankRule\ t\ 0)}
\end{aligned}$$

As another example, consider Bird's 'repmin' problem [26]: replace every element of a tree with the minimum element in that tree. Bird shows a clever circular one-pass solution; as was pointed out soon afterwards, not least (and perhaps first?) by Doaitse himself [27, 19], this circular program precisely corresponds to lazy evaluation of an attribute grammar. Here, the sole inherited attribute is the value with which to replace all labels, and the sole synthesized attribute for a node is the minimum value in the subtree rooted there:

$$\begin{aligned}
&\mathit{repminRule} :: \mathit{Ord\ a} \Rightarrow \mathit{Rule\ TreeF\ a\ a\ a}\\
&\mathit{repminRule\ a\ (Inl\ Unit}, m) = (a, \mathit{Inl\ Unit})\\
&\mathit{repminRule\ a\ (Inr\ (Id\ x} :\times: \mathit{Id\ y)}, m)\\
&\qquad\qquad = (\mathit{min\ a\ (min\ x\ y)}, \mathit{Inr\ (Id\ m} :\times: \mathit{Id\ m)})
\end{aligned}$$

Solving the repmin problem in a single pass then amounts to complete attribute evaluation according to this rule, followed by discarding the original labels and the synthesized attributes—but ensuring that the initial value of the inherited attribute is the synthesized attribute associated with the root:

$$\begin{aligned}
&\mathit{repmin} :: \mathit{Ord\ a} \Rightarrow \mathit{Tree\ a} \to \mathit{Tree\ a}\\
&\mathit{repmin\ t} = \mathbf{let}\ u = \mathit{annotate\ repminRule\ t\ (thd3\ (root\ u))}\ \mathbf{in}\ \mathit{fmap\ snd3\ u}
\end{aligned}$$

(where $\mathit{snd3}\ (a, b, c) = b$ and $\mathit{thd3}\ (a, b, c) = c$). The grammar rule is acylic; it is this main program that ties the cyclic knot—u is defined in terms of itself.

7.5 Accumulations as attribute evaluation

Now, complete attribute evaluation in the sense of the previous section generalizes both upwards and downwards accumulations. As we have seen, the algebra for a fold can be expressed as an attribute grammar rule using only synthesized attributes, and evaluating the fold amounts to computing the synthesized attribute of the root node using that rule. As one might therefore expect, any upwards accumulation can be computed by a complete attribute evaluation using the same attribute rule (and then discarding the original labels and the trivial inherited attribute values):

$$\mathit{scanu\ \phi\ t} = \mathit{fmap\ thd3\ (annotate\ (upRule\ \phi)\ t\ ())}$$

Dually, any downwards accumulation can be computed by a complete attribute evaluation using only inherited attributes, using the following rule:

$$
\begin{aligned}
&\mathit{downRule} :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f)) \Rightarrow \\
&\qquad (b \to \mathit{Dir}\ f \to a \to b) \to (a \to b) \to \mathit{Rule}\ f\ a\ (a \to b)\ () \\
&\mathit{downRule}\ f\ g\ a\ (ss, k) = ((), \mathit{fmap}\ (f\ (k\ a) \circ \mathit{fmap}\ \mathit{bang} \circ \mathit{snd})\ (\mathit{posns}\ ss))
\end{aligned}
$$

Note that the inherited attribute for a node can depend only on context outside the tree rooted there, so in particular cannot depend on the node label. Therefore we make the inherited attribute a function from node label to result, and finish off the computation by applying the inherited attributes to the original labels:

$$
\mathit{scand}\ f\ g\ t = \mathit{fmap}\ (\lambda(a, i, ()) \to i\ a)\ (\mathit{annotate}\ (\mathit{downRule}\ f\ g)\ t\ g)
$$

7.6 Attribute evaluations as accumulation

Clearly, there is a strong analogy between complete attribute evaluation and upwards and downwards accumulations: an upwards accumulation is the complete evaluation of an attribute grammar with only synthesized attributes, and a downwards accumulation is the complete evaluation of a grammar with only inherited attributes.

Conversely, it turns out that any complete attribute evaluation can be expressed as an upwards accumulation followed by a downwards accumulation. The idea is to make a first pass over the tree, labelling every node with (the original node label and) a function from the input inherited attribute to the synthesized attribute of that node together with the inherited attributes for the children; then to finish up by composing all the inherited-to-inherited-attribute functions along each of the paths from the root. The first pass is an upwards accumulation:

$$
\begin{aligned}
&\mathit{collect} :: (\mathit{Functor}\ f, \mathit{Zippable}\ f) \Rightarrow \\
&\qquad \mathit{Rule}\ f\ a\ i\ s \to \mathit{Mu}\ f\ a \to \mathit{Mu}\ f\ (a, i \to (s, f\ i)) \\
&\mathit{collect}\ r = \mathit{scanu}\ (\lambda a\ \mathit{afs} \to (a, \phi\ r\ a\ \mathit{afs}))\ \mathbf{where} \\
&\quad \phi\ r\ a\ \mathit{afs}\ i = (s, \mathit{is}')\ \mathbf{where} \\
&\qquad (s, \mathit{is}) = r\ a\ (\mathit{ss}, i) \\
&\qquad \mathit{fis} \quad = \mathit{zip}\ (\mathit{fmap}\ \mathit{snd}\ \mathit{afs})\ \mathit{is} \\
&\qquad \mathit{ss} \quad = \mathit{fmap}\ (\mathit{fst} \circ \mathit{uncurry}\ (\$))\ \mathit{fis} \\
&\qquad \mathit{is}' \quad = \mathit{fmap}\ \mathit{snd}\ \mathit{fis}
\end{aligned}
$$

The second is a downwards accumulation, followed by a map to discard the children's inherited attributes:

$$
\begin{aligned}
&\mathit{distribute} :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f), \mathit{Zippable}\ f) \Rightarrow \\
&\qquad i \to \mathit{Mu}\ f\ (a, i \to (s, f\ i)) \to \mathit{Mu}\ f\ (a, i, s) \\
&\mathit{distribute}\ i\ t = \mathit{fmap}\ \mathit{tidy}\ (\mathit{scand}\ \mathit{step}\ (\mathit{start}\ i)\ t)\ \mathbf{where} \\
&\quad \mathit{step}\ (\_, \_, (\_, \mathit{is}))\ d\ (a, h) = \mathbf{let}\ i = \mathit{select}\ \mathit{is}\ d\ \mathbf{in}\ (a, i, h\ i) \\
&\quad \mathit{start}\ i\ (a, h) \qquad\qquad\quad = (a, i, h\ i) \\
&\quad \mathit{tidy}\ (a, i, (s, \mathit{is})) \qquad\quad = (a, i, s)
\end{aligned}
$$

Putting them together, we have

$$
\mathit{annotate}\ r\ t\ i = \mathit{distribute}\ i\ (\mathit{collect}\ r\ t)
$$

Incidentally, this pattern of 'collect information upwards through the tree, then redistribute it downwards' is very common; in my thesis [1] and subsequent papers [28, 29] I showed that it also crops up in efficient parallel algorithms for computing prefix sums and in drawing trees.

8 Acknowledgements

The ideas in this paper have been simmering gently at the back of my mind for a long time, and have clearly benefitted from developments from other colleagues (especially Richard Bird and Conor McBride). I am grateful for comments from the Algebra of Programming group at Oxford, for pointing out some errors in drafts. The paper itself was written while I was visiting the National Institute of Informatics in Tokyo, and I would like to thank Zhenjiang Hu for his hospitality.

Last, but not least, I would especially like to thank you, Doaitse, for all the interactions I have had with you over the last twenty years and more: you have always had strong views and strong principles, and have stuck to them tenaciously and with integrity. You have set an uncompromising example for us to follow.

References

1. Gibbons, J.: Algebras for Tree Algorithms. D. Phil. thesis, Programming Research Group, Oxford University (1991) Available as Technical Monograph PRG-94. ISBN 0-902928-72-4.
2. Gibbons, J.: Datatype-generic programming. In Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J., eds.: Spring School on Datatype-Generic Programming. Volume 4719 of Lecture Notes in Computer Science., Springer-Verlag (2007)
3. Gibbons, J.: Origami programming. In Gibbons, J., de Moor, O., eds.: The Fun of Programming. Cornerstones in Computing. Palgrave (2003) 41–60
4. Hagino, T.: A Categorical Programming Language. PhD thesis, Department of Computer Science, University of Edinburgh (1987)
5. Malcolm, G.: Data structures and program transformation. Science of Computer Programming **14** (1990) 255–279
6. Bird, R.S.: An introduction to the theory of lists. In Broy, M., ed.: Logic of Programming and Calculi of Discrete Design. Springer-Verlag (1987) 3–42 Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.
7. Bird, R.S.: Lectures on constructive functional programming. In Broy, M., ed.: Constructive Methods in Computer Science, Springer-Verlag (1988) 151–218 Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.
8. Bird, R.S.: Algebraic identities for program calculation. Computer Journal **32**(2) (1989) 122–126
9. Bird, R., de Moor, O., Hoogendijk, P.: Generic functional programming with types and relations. Journal of Functional Programming **6**(1) (1996) 1–28
10. McBride, C.: The derivative of a regular type is its type of one-hole contexts. `http://strictlypositive.org/calculus/` (2001)
11. McBride, C.: Clowns to the left of me, jokers to the right: Dissecting data structures. In: Principles of Programming Languages. (2008) 287–295
12. Huet, G.: The zipper. Journal of Functional Programming **7**(5) (1997) 549–554
13. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **2**(2) (1968) 127–145
14. Irons, E.T.: A syntax directed compiler for Algol 60. Communications of the ACM **4** (1961) 51–55
15. Knuth, D.E.: Examples of formal semantics. In Engeler, E., ed.: Lecture Notes in Mathematics 188: Symposium on Semantics of Algorithmic Languages, Springer-Verlag (1971) 212–235
16. Swierstra, S.D., Azero Alcocer, P.R., Saraiva, J.: Designing and implementing combinator languages. In Swiestra, S., Henriques, P., Oliveira, J., eds.: Advanced Functional Programming, Third International School. Volume 1608 of Lecture Notes in Computer Science., Springer Verlag (1999) 150–206
17. Centre for Software Technology: Utrecht University Attribute Grammar Compiler (UUAGC). `http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem` (2009)

18. Fokkinga, M., Jeuring, J., Meertens, L., Meijer, E.: A translation from attribute grammars to catamorphisms. The Squiggolist **2**(1) (1991) 20–26
19. Johnsson, T.: Attribute grammars as a functional programming paradigm. In Kahn, G., ed.: LNCS 274: Functional Programming Languages and Computer Architecture, Springer-Verlag (1987) 154–173
20. Jazayeri, M., Ogden, W.F., Rounds, W.C.: The intrinsically exponential complexity of the circularity problem for attribute grammars. Communications of the ACM **18**(12) (1975) 697–706
21. Chirica, L.M., Martin, D.F.: An order-algebraic definition of Knuthian semantics. Mathematical Systems Theory **13**(1) (1979) 1–27
22. Jourdan, M.: Strongly non-circular attribute grammars and their recursive evaluation. In: Symposium on Compiler Construction. (1984) 81–93
23. Katayama, T.: Translation of attribute grammars into procedures. ACM Transactions on Programming Languages and Systems **6**(3) (1984) 345–369
24. Reps, T., Teitelbaum, T.: The Synthesizer Generator—A System for Constructing Language-Based Editors. Springer-Verlag (1989)
25. Jacobs, B., Uustalu, T.: Semantics of grammars and attributes via initiality. In Barendsen, E., Capretta, V., Geuvers, H., Niqui, M., eds.: Reflections on Type Theory, Lambda-Calculus, and the Mind: Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday, Radboud Universiteit Nijmegen (2007) 181–196 `http://www.cs.ru.nl/barendregt60/essays/jacobs_uustalu/art15_jacobs_uustalu.pdf`.
26. Bird, R.S.: Using circular programs to eliminate multiple traversals of data. Acta Informatica **21** (1984) 239–250
27. Kuiper, M.F., Swierstra, S.D.: Using attribute grammars to derive efficient functional programs. In: Computing Science in the Netherlands, Amsterdam, SION (1987) 39–52 Also Utrecht University Technical Report RUU-CS86-16, August 1986.
28. Gibbons, J.: Upwards and downwards accumulations on trees. In Bird, R.S., Morgan, C.C., Woodcock, J.C.P., eds.: Mathematics of Program Construction. Volume 669 of Lecture Notes in Computer Science., Springer-Verlag (1993) 122–138
29. Gibbons, J.: Deriving tidy drawings of trees. Journal of Functional Programming **6**(3) (1996) 535–562

Metaclasses in object oriented programming languages

Piet van Oostrum

piet@vanoostrum.org
Dept. of Computer Science, Utrecht University (Retired)

Abstract. In most object oriented programming languages objects are instances of a class (An exception is for example Javascript). The class defines the behaviour of its instances. If a language would be a pure object oriented language then everything must be an object, i.e. also the classes themselves should be objects. And therefore they should also be instances of another class, which is called their metaclass. The metaclass defines the behaviour of the class, for example the creation of new instances of the class.
In this paper I will give an overview of the possibilities of metaclasses in a number programming languages: Java, Smalltalk and Python. It is not necessary to have in depth knowledge of these programming languages.

1 Introduction

I have worked about 28 years in the Department of Cumputer Science in Utrecht and most of this time I was in Doaitse's group. Somewhere in the 90's[1] we got interested in object oriented programming. So we studied some of this in a small group. It was mostly based on Smalltalk[1]. However we did not have a Smalltalk implementation on our computers so it was mainly theoretical. Later I was asked to give a course on object oriented programming. This was also a more theoretical course on the concepts, not a programming course. The course was based on the well known book by Tim Budd[2]. One concept that attracted me particularly was that of metaclasses. Especially as Smalltalk has quite a nice system of metaclasses.

In this paper I will give an overview of the possibilities of metaclasses in a number of programming languages: Java, Smalltalk and Python.

In Section 2 we start with definitions of the concepts and terms that we will use in the rest of the paper. Then in section 3 we describe the general concept of metaclasses, apart from the implementation in specific programming languages. In Section 4 we look at the metaclass that Java provides. Then in Section 5 we describe the metaclass system of Smalltalk as an example of a theoretically perfect metaclass system. Finally in Section 6 we give some practical applications of metaclasses in Python.

2 Definitions

There are many definitions of "object oriented programming languages", all of which are slightly different. I will use the following definition:

Definition 1. *An* object oriented programming language *is a programming language that obeys the following:*

[1] I am writing this paper in Bolvia, and my archive that could give me some more exact dates is in Utrecht, so I have to make some educated guesses about the dates.

- *(almost) every value is an object*
- *an object consists of a collection of data and corresponding operations that work on these data*
- *every object is an instance of some class*
- *the classes are organized in some hierarchy of super- and subclasses (inheritance)*
- *all operations on an object are executed in some method defined in the class of the object or some superclass thereof*

Notes:

1. There are programming languages that have objects but no classes. The most well known of these in Javascript. I wil not consider these in this paper.
2. The hierarchy may be a tree or forest (single inheritance), where every class has at most one parent (superclass) or a directed acyclic graph (multiple inheritance), where a class may have multiple parents (superclasses).
3. Many object oriented programming languages have some values that are not objects. For example in Java `int`, `float`, `bool`, etc values are not objects. Although these languages are not purely object oriented we still consider them object oriented because they adhere mostly to our definition.
4. In some object oriented languages actions can also be executed outside of methods, for example in Perl and Python you can also use the traditional imperative programming style.
5. In some object oriented programming languages there are things that are not objects and not even values, for example methods in Java. These are not first class citizens, that is they cannot be put in variables, passed as parameters, etc. In Java, however, information about methods can be obtained by means of reflection.

In the following definitions we use lower case variables for objects and uppercase for classes. These definitions are given to take away ambiguities of the terms used.

Definition 2. *An object o is a* direct instance *of a class C iff o has been created by a statement* `new` *C or its equivalent.*
Notation: $C = \text{class}(o)$
Axiom: Each object is a direct instance of exactly one class.

Definition 3. *A class C is a* direct subclass *of a class D iff C has been declared with* `extends` *D or its equivalent.*
Notation: $\text{ds}(C, D)$.

Definition 4. *The relation* subclass *(C is a subclass of D) is the transitive closure of* ds.
Note: with this definition C is a subclass of itself.
Notation: $C \sqsubseteq D$.

Definition 5. *An object o is an instance of a class C iff* $\exists D : D = \text{class}(o) \wedge D \sqsubseteq C$
Notation: $\text{instance}(o, C)$.

2.1 Properties of objects and classes

An object has an *identity*, a *class* and a mapping from identifiers to values: the *instance variables*. There is also a mapping from identifiers to implementations of methods but in the majority of object oriented programming languages this mapping is the same for all direct

instances of the same class. The methods are then defined in the class, and not separately with the instances.

A class has a list of its superclasses (a single class in the case of single inheritance) and it has a mapping of identifiers to the implementation of the methods for its instances. Please note that the method implementations define the behaviour of the instances, not the behaviour of the class.

There may also be mappings from identifiers to static variables and implementations of static methods.

In languages with static type checking the mapping from identifiers to the types of the values of the instance variables and the methods is a characteristic of the class and is the same for all instances of the class.

We leave it open whether the information mentioned above is available at compilation time, execution time or a combination of these. We will also leave interfaces mostly out of the discussion.

3 Metaclasses

When everything in a programming language is an object, ideally also classes should be objects. This would mean that classes would be able to be manipulated at run time, at least to a certain extend. Of course the possibility to manipulate classes at run time gives a lot of possibilities for abuse, and it may impact the efficiency of code generation and run time optimization, but certainly this gives us interesting new possibilities.

As we have stated that every object should be a direct instance of one class, this implies that each class, considered as an object, should also be an instance of some other class. We call the latter class the *metaclass* of the other class.

Definition 6. *A* metaclass *is a class whose (potential) instances are itself classes. The* metaclass *of a class C is the (unique) class M of which C is a direct instance. In this case we will use the notation $M = \text{metaclass}(C)$, but note that this is the same as $M = class(C)$.*

Object oriented programming languages do not have to have metaclasses. For example in C++ classes are not first class objects, therefore it does not have metaclasses.

There is still a lot of freedom in how the metaclasses could be organised. For example each class could have its own metaclass (implying that each metaclass has only one instance), or some classes could share the same metaclass.

As each class defines the behaviour of its instances the metaclass defines the behaviour of the class(es) that are its instances. The behaviour of classes consists mainly of how it creates its instances, and this usually is the main reason to use metaclasses. If you want to define a class that creates its instances in a non standard way then you would have to change the behaviour of that class compared to the "standard" behaviour of classes. In other words you would need to define a special metaclass for this class. Examples would be a singleton class, where only a single instance is ever created, or a class that automatically gives all its instances a special behaviour, for example with regard to logging and/or tracing.

As metaclasses are also classes themselves they will have their own metaclasses. Potentially this leads to an infinite chain of metaclasses. We can break this infinity in a number of ways:

1. Have a "top" metaclass that doesn't have its metaclass. This defies the purity of the metaclass concepts and is therefore undesirable.

2. Have a loop somewhere in the metaclass chain. This could for example be a metaclass that is its own metaclass. Such a construction could probably not be made programmatically in a user's program because you must have the class before you can create an instance, but could be builtin in the language system. This is similar to the builtin class "Object" but then on the metaclass level.
3. Create metaclasses on demand. Then there is potentially an infinite chain of metaclasses but they are constructed lazily. There is, however, not much use for an infinite chain of metaclasses as the need to differentiate the behaviour of the metaclasses at higher levels is almost non-existant. Therefore this solution is also not very useful.

For reasons that are beyond the scope of this paper[3] metaclasses often have the same subclass hierarchy as the classes that are their instances, i.e.

$$\text{metaclass}(C) \sqsubseteq \text{metaclass}(D) \text{ if } C \sqsubseteq D$$

In the following sections we will see how metaclasses are used in Java, Smalltalk and Python.

4 Metaclass in Java

Classes in Java are loaded on demand, on the first use. This is for example when an instance of the class is created or when a static method of the class is called.

In Java there is an object for every class that a program uses. We will call this the *class object*. The class objects are related to the classes that are declared in the program, or more accurately in the collection of files that constitute the program. To a certain extent this collection is dynamic: during run time it can increase. It is not possible to create class objects that do not correspond to classes defined in source files. This would be very hard anyway, because how would you define the behaviour of these classes?

The main uses for the class objects are:

- to obtain information about the class at run time, for example what methods the class has, what the types of the parameters and result of the methods are. This is called reflection or introspection. For this use there are also "pseudo" class objects for the primitive types and `void`, for array classes and interfaces.
- to create instances of the class when the name of the class is only known at run time and not at compile time.
- to invoke methods on an object when the type of the object, the name of the method and/or the types of the method parameters only are known at run time and not at compile time.
- to be used as types in the description of parameters and results of methods and instance variables.

4.1 How to obtain a class object?

There are a number of ways to obtain a class object at run time in a Java program:

1. When you have an object you can get its class object with the call `obj.getClass()`. This corresponds to what we have defined as $C = \text{class}(o)$. This can be useful if the object was declared as a parameter with a superclass or interface as its type and similar situations, where the program wants to have more detailed information about the actual class at run time that the object belongs to.

2. When you have the name of the class at compile time you can use the construction `C.class` where `C` is the name of the class. This is called a *class literal*. It looks like a static variable of the class, but it isn't. It also can't be because `class` is not an identifier but a keyword of the language. This can also be used for primitive types, void, interfaces and array classes. So for example `int.class` is a notation for the class object that describes `int`s, even though `int`s are not objects.
3. In the next section we will see another way to get the class object by means of the metaclass.

The class objects have a collection of methods to obtain information about the class or to create new instances of it. One of these methods is `getName` that returns the name of the class as a string.

The following piece of code

```
Object x = new Object();
System.out.println(x.getClass());
System.out.println(int.class.getName());
```

will print

```
class java.lang.Object
int
```

Of course the second line of our code will implicitly call the `toString` method of the class object (defined in its metaclass of course) that generates the string representation shown.

4.2 The metaclass

In Java there is only one metaclass and its name is `Class`. Because there is only one and it is predefined it is not possible to change the behaviour of classes. For this to be possible it would be necessary to define your own metaclasses. The metaclass defines the behaviour of classes (although in Java most of the behaviour is defined by the compiler) and it contains instance methods that can be used on class objects, like the aforementioned method `getName`. It also defines some static methods that we will look at later.

As a metaclass is also a class and all classes have `Class` as their metaclass, `Class` must be its own metaclass. This is the loop that we mentioned before to prevent an infinite number of metaclasses to appear. You can check this with the following program segment:

```
Object x = new Object();
System.out.println(x.getClass());
System.out.println(x.getClass().getClass());
System.out.println(Class.class==Class.class.getClass();
```

In line 2 we again print the class object of `x` which is the class `Object`, or more accurately `java.lang.Object`. Then we do the same for the class of this class object, in other words its metaclass which is `java.lang.Class`. In the last line we check if the metaclass of `Class` is equal to itself. Remember that `Class.class` is a literal notation for the class object that represents the class `Class`, not for the class of `Class`. The output of the above program fragment is:

```
class java.lang.Object
class java.lang.Class
true
```

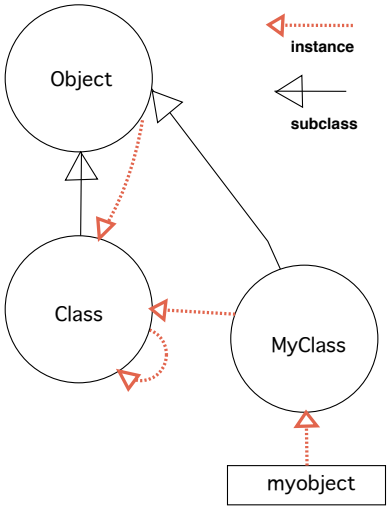


Fig. 1. Single metaclass structure

In Figure 1 we see the structure of the metaclass `Class` with respect to other classes. Each class is a direct instance of `Class`, therefore `Class` is an instance of itself. Also each class is directly or indirectly a subclass of `Object`, the root of the inheritance hierarchy, so `Class` is also a subclass of `Object`. Because `Object` itself is a class, it is also an instance of `Class`. It is the only class that does not have a superclass.

For generic classes and interfaces there is also the generic construction `Class<T>`, which may be necessary to avoid type errors.

4.3 Other methods of Class

The following methods are instance methods for class objects. Because instance methods are defined in the class of the instances, these methods are defined in the class `Class`.

When you have the name of a class at runtime as a string in the variable `name` you can obtain the class object of that class with the method call `Class.forName(name)`. The class (i.e. the implementation of it) will be loaded if this has not yet been done. This will throw an exception if the class cannot be loaded. Of course `forName` is a static method of `Class`.

Here is a complete program that prints the names of the methods of a class when the name of the class is given (in this example of the class `java.lang.Integer`). Of course in this particular example we could have used the class literal instead of `forName`, but that would be less illustrative.

```
class MetaExample {
  public static void printmethods(String classname)
  {
    try {
      Class objClass = Class.forName(classname);
      System.out.println("Classname: "
                         + objClass.getName());
      Method[] methods = objClass.getMethods();
      for (int i = 0; i < methods.length; i++)
```

```
        System.out.println("method: "
                           + methods[i].getName());
    }
      catch (ClassNotFoundException e) {
        System.out.println("No class "+classname);
    }
  }
  public static void main (String args[]) {
    printmethods("java.lang.Integer");
  }
}
```

When you have the class object of a class C you can create an instance of C with the method call `C.newInstance()`. Here C will generally be a variable or expression. Note that you will use this when you don't know at compile time which class it will be, otherwise you would use the compile time construction `new C()` which is equivalent for a class whose name is known at compile time.

`C.isInstance(obj)` is the implementation of the relation isinstance(obj,C) that we defined in Section 2. It is the run time equivalent of the static construction `obj instanceof C`.

`C.isAssignableFrom(D)` is the implementation of the relation $D \sqsubseteq C$ defined in Section 2. It tests if an instance of the class D can be assigned to a variable or parameter of type C, which is equivalent to $D \sqsubseteq C$. See the following program fragment:

```
C x;
D y = new D(...)
...
x = y;
```

The assignment `x = y` is only valid iff $D \sqsubseteq C$.

4.4 Use of Class for reflection

There are many more methods in `Class` that are useful for introspection (also called reflection), but these are beyond the scope of this paper. We have already seen the use of `getMethods` to get a list of the methods of a class. In this section I want to mention a practical application of this. It is actually used in the Java Remote Method Invocation system (RMI) and can be used for similar systems like Corba or SOAP.

In these systems a method can be invoked from a client system to a remote system where the actual object resides. The client system sends all the required data for the method call to the remote system, including the method name, the values and the types of the parameters. The remote system will then use introspection to find the actual method and invoke it.

5 Metaclasses in Smalltalk

Smalltalk is a programming language developed by XEROX PARC labs that is one of the purest object oriented languages in existence. Most modern object oriented concepts come from the programming languages Simula-67 and Smalltalk which could be called the father and mother of object oriented programming. Statically typed languages like C++ and Java have more characteristics from Simula, and dynamic languages like Objective-C, Python

and Ruby are more like Smalltalk. Smalltalk was mainly developed as a research project, but is still used as production environment today[2].

In Smalltalk all values are objects, including 'primitives' like integers and floats. All objects are instances of some class. Methods themselves are not objects in Smalltalk but neither are they values.

Everything in a Smalltalk system can be manipulated on run time, including the definitions of the classes and the compiler. Smalltalk uses a different terminology than modern programming languages. Instead of method calls for example they use the term message sending. It is written without the dot, for example like "`object message`". Also parameters are not given inside parentheses but with keywords like "`paramname:`" Everything in Smalltalk is done with these messages, for example to create a new instance of a class you send the message `new` to the class object. Also arithmetic operators are messages. An expression like `2 + 3` means sending the message "`+`" with parameter `3` to the object `2`. The object sends back the result `5`. These messages have only a single parameter that does not have a keyword name. I will sometimes use the more common term "method call" instead but please note that this is not the official Smalltalk terminology.

Smalltalk is a dynamic language like Python. Type checking is done at run time and not at compile time.

Classes in Smalltalk are created by sending a message `subclass` to the intended superclass. This might come as a surprise because you might have expected that a message similar to `new` would have been sent to the metaclass instead. However, as we will see below, the metaclass does not yet exist when we want to create the class. Ultimately the metaclass will be involved, however.

An example is the creation of the class `Boolean` which is a subclass of the class `Object`. This will not be used in a user program, however, but in the system itself, but the system uses exactly the same method as user programs. The following code is used:

```
Object subclass: #Boolean
       instanceVariableNames: ''
       classVariableNames: ''
```

You see the parameter keywords. There are three parameters: the name of the class (`#` indicates a "symbol", i.e. an interned string), a list of instance variable names, and a list of class variable names. Methods are added later.

I have taken `Boolean` as an example because it has special characteristics with respect to the creation of instances. Booleans are nothing special in Smalltalk; they are defined as normal objects. `Boolean` is an abstract class (i.e. it doesn't have instances itself). It has two subclasses: `True` and `False`. Both of these are singleton classes: they each have only a single instance: `true` and `false` respectively.

`Boolean` redefines the method `new` to generate an error if instances of it are created because you don't want new booleans to be added to the system. Because `new` is a message sent to the class object it is defined in its metaclass. As the behaviour of `Boolean` is different from that of other classes, the metaclass of `Boolean` must be different from the metaclasses of other classes.

[2] If you want to experiment with Smalltalk I advise to download and install Pharo from http://pharobyexample.org/ or http://www.pharo-project.org/. Pharo is a modern implementation of Smalltalk that is actively being developed. It comes with a complete graphical development environment and an elaborate library.

5.1 Metaclasses in Smalltalk

Older versions of Smalltalk had only a single metaclass, as in figure 1. However, this appeared not to be adequate, as we have seen in the example of the class `Boolean`.

In modern Smalltalk each class has its own metaclass, which therefore has a single instance. The metaclass is created by the system at the same time the class is created, and the class is made into an instance of the metaclass. The metaclasses have the same inheritance relation as the classes as mentioned in section 3. On top of the hierarchy is an additional class `Class`. This is an abstract class, i.e. it does not have instances. Therefore this class is not considered to be a metaclass. It does define the common behaviour of the normal (non-meta) classes, which can be overridden for a particular class in its metaclass.

Metaclasses don't have a proper name (identifier) as other classes but they are denoted with a name with a space in it. The 'name' of the metaclass of class `C` is "`C class`". To access the metaclass of `C` you have to send the message `class` to the class `C`, i.e. the construction "`C class`" is used. This looks similar to the 'name' of the metaclass but that is a string, whereas we are talking now about a method call.

In Figure 2 we see the structure of the Smalltalk metaclass system. Two normal classes are shown: `Vehicle`, and its subclass `Car`. We see the class hierarchy with `Object` at the top and the corresponding metaclass hierarchy with the additional class `Class` at the top. `Class` does also have a metaclass, which obviously is `Class class`.

All metaclasses must also have their metaclass. However there is no need to have different behaviour for different metaclasses and therefore there is a single class `Metaclass` of which all metaclasses are instances. This class is not a standard metaclass as it has more than one instance. It defines the behaviour of the metaclasses. `Metaclass` has its own metaclass which is `Metaclass class`. The latter one, being a metaclass is also an instance of `Metaclass`. This completes the loop that prevents an infinite chain of metaclasses. Another loop we find in `Object` → `Object class` → `Class` → `Object`.

In the figure you will find two additional classes, indicated by small circles, one of which is the common superclass of `Class` and `Metaclass`, so this defines the common behaviour of all classes, and its metaclass. Actually the situation is a little bit more complicated (each circle represents two classes) but this is of no interest to our metaclass discussion.

6 Metaclasses in Python

Python is an object oriented programming language in which everything is an object, including 'primitive' types like `int`, `float` and `bool`. Even methods are objects so it is possible to say
`m = obj.meth` (without parentheses) and then later:
`m(par1, par2)` which is equivalent to `obj.meth(par1, par2)`.

Python is a dynamic language, which means that most things occur at run time. So variables have dynamic types, i.e. they can contain objects of different types during a run and the type checking is done at run time. Also definitions of functions, methods and classes, although they look superficially like compile time declarations in other programming languages, are actually executed at run time. This also implies that it is possible to define classes and methods dynamically at run time, and to change the behaviour of classes and objects at run time. It is for example possible to add new methods to a class, remove methods or change the implementation of a method at run time.

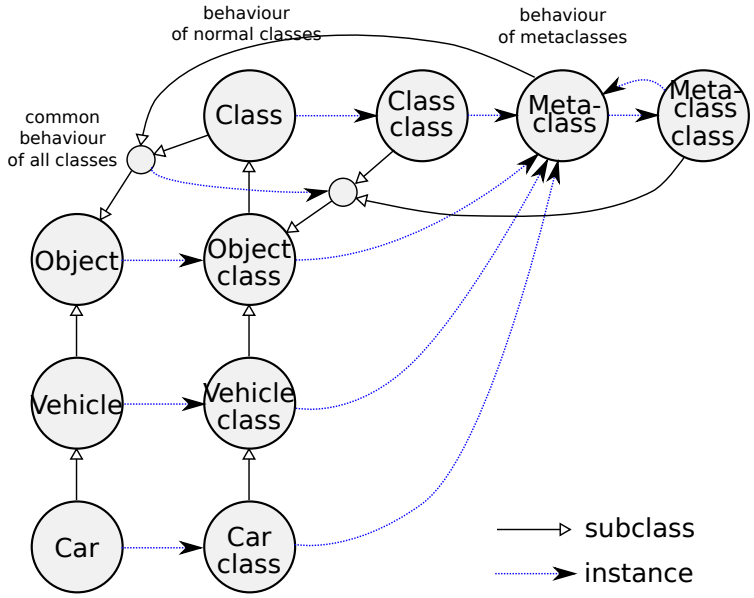


Fig. 2. Smalltalk metaclass structure

Python uses multiple inheritance, i.e. a class can have more than one superclass. The class hierarchy can therefore be a directed acyclic graph. The top of this hierarchy is the class `object`[3].

6.1 Python object and class creation

The class `object` is the only one that can create new objects, by means of its method `__new__`. All other classes either have to inherit `__new__` or redefine it, with a definition that invokes `object.__new__`. Actually object creation in Python is a two step process, the first of which is the call of `__new__`, while the second is the call of the class's `__init__` method. Most classes only define `__init__` and use the standard implementation of `__new__`.

Classes are created by the class `type`. This is the default metaclass. Class creation can be changed by defining your own metaclasses and these should be subclasses of `type` and inherit, or explicitly call the `type.__new__` method. The name `type` may sound strange but classes are types in python, therefore this name was chosen (a class is a type, therefore a class is an instance of `type`).

There are no rules about the inheritance structure of metaclasses like in Smalltalk, except in the case of multiple inheritance, but these are outside the scope of this paper. If you want to create a class with a different metaclass you give the class a variable `__metaclass__` or define this variable before the class definition[4].

6.2 Example

In this section I will give an example. Please note that variables are not declared in Python, and that block structure is by indentation.

[3] In Python 2 there are so called old style classes that do not inherit from object, but these are deprecated. In Python 3 these do no longer exist.

[4] In Python 3 you can give the class definition an extra parameter `metaclass=...`.

```
class MyClass(object):
  variable = "test"

  def __init__(self, value):
    self.val = value

  def method(self, param):
    return self.val + param
```

Some things to note:

- Variables defined in the class definition are "class variables", not instance variables. They belong to the class and are shared by all instances of the class.
- Instance variables can only be created and accessed with the `obj.var` notation, even in methods. As there is no implicit `this` in Python, the object on which a method works must be explicitly given as the first parameter in the method definition. By convention this parameter is usually called `self`. The same applies to the initializer method `__init__`. In the example above `val` is the only instance variable. It is however, possible to create instance variables dynamically, even outside methods, by using the `obj.var` notation.
- This first parameter will not be present in a method call. In fact we could say that it is present at the left of the dot in the `obj.meth(parameters)` notation.
- Superclasses are indicated between parentheses in the class definition.
- Methods (and functions outside classes) are defined with the keyword `def`.

Instances are created by calling the class object with the parameters as defined in the `__init__` method. There is no keyword like `new`. So continuing the above example we can have:

```
myobj = MyClass(3)
print myobj.val # this should print 3
myobj.method(5)
print myobj.val # this should print 8
```

6.3 Method objects

A method can be bound to an object by the notation `obj.methodname`. As we have said before this also is an object. The last example could have been written as:

```
myobj = MyClass(3)
print myobj.val # this should print 3
mymethod = myobj.method
mymethod(5)
print myobj.val # this should print 8
```

with the same effect. Here `mymethod` will receive a method object. This object will actually be a closure consisting of the function object together with the corresponding instance (the one on the left of the dot). In the call this instance will be used as the first parameter `self`.

6.4 Creation of classes

In Python a class is constructed at compile time. We take the previous example augmented with some (not shown) superclass.

```
class MyClass(MySuperClass):
  variable = "test"

  def __init__(self, value):
    self.val = value

  def method(self, param):
    return self.val + param
```

At execution time the system calls the metaclass with 3 parameters:

1. the name of the class as a string
2. a list of the superclasses
3. a dictionary (hash table) with all the definitions inside the class. The name will be used as the key and the contents of the definition as value. In the above example there are 3 items in the dictionary: 'variable' with value "test", '`__init__`' and 'method' both with a function as value.

As we have not specified a metaclass in this example the default metaclass `type` will be used. For the rest there is nothing magical about the class definition. It is just syntactic sugar to make the life of the programmer a little bit easier. But we could as well have called `type` ourself with the proper parameters:

```
def tempfunc1(self, value):
    self.val = value

def tempfunc2(self, param):
    return self.val + param

MyClass = type('MyClass',
               [MySuperClass],
               {'variable': "test",
                '__init__': tempfunc1,
                'method': tempfunc2
               })

# remove the function names (but not the functions)
del tempfunc1, tempfunc2
```

6.5 Python metaclass example

We now give an example of a metaclass that automatically gives its instance classes logging , i.e. each method in the class is provided with logging facilities. We will use a higher order function `logged` for this purpose. This will create a new function that sandwiches the original function between logging calls.

The metaclass has to redefine its `__new__` method (that is used to create its class instances) and this picks up those elements in the dictionary parameter that are methods and

replaces them with a logged version. We use the method name as logging key. We distinguish methods from other items based on the fact that they are *callable*. After replacing these we call the `__new__` method of the default metaclass `type` to do the actual creation of the class.

```
 class LogMeta(type):
    def __new__(cls, name, bases, dic):
        for key, val in dic.items():
            if callable(val):
                dic[key] = logged(key, val)
        return type.__new__(cls, name, bases, dic)

def logged(key, func):
    def newfunc(*args, **kwargs):
        print "Entering", key
        res = func(*args, **kwargs)
        print "Exiting", key
        return res
    return newfunc
```

Now we can use the metaclass:

```
class MySuperClass(object):
    pass # does nothing

class MiClass(MySuperClass):
    __metaclass__ = LogMeta
    variable = "test"

    def __init__(self, value):
      self.val = value

    def method(self, param):
      return self.val + param
```

Now we use this class in an interactive interpreter:

```
In [8]: m = MyClass(3)
Entering __init__
Exiting __init__

In [9]: m.method(5)
Entering method
Exiting method
Out[9]: 8
```

And we see that the methods have been replaced by a logging version of themselves.

Similarly we could define a metaclass to make all method calls mutually exclusive by wrapping calls to a semaphore `acquire` and `release` around each method, making the instances into a monitor like in Java.

This is quite a powerful way of changing the behaviour of a class. You can introduce the logging for example for testing and when you don't need it anymore you just remove the `__metaclass__ = LogMeta` line. As class definitions are executed at run time you could

even make this assignment conditional on some startup or configuration parameter. It would even be possible to disable logging from the whole program by a construction like:

```
if nologging:
  LogMeta = type
```

and leave the rest the same. Of course this would have to be done before any class with logging would have been defined.

More applications for metaclasses are left to the imagination of the reader. In the literature you can find more examples.

7 Conclusion

Metaclasses are a powerful concept for advanced programming. Although they are not something you would use every day, it is very useful to know them and to be able to use them when the need arises.

References

1. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983) Downloadable from http://stephane.ducasse.free.fr/FreeBooks/.
2. Budd, T.: An Introduction to Object-Oriented Programming. Addison-Wesley (1991)
3. Forman, I.R., Danforth, S.: Putting Metaclasses to Work. Addison-Wesley (1999)
4. Nierstrasz, O., Black, A.P., Ducasse, S., Pollet, D.: Pharo by example http://PharoByExample.org.
5. Mertz, D.: A primer on python metaclass programming (2003) http://onlamp.com/pub/a/python/2003/04/17/metaclasses.html.

Doaitse, parsers, theorie en praktijk

Harald Vogt

harald.vogt@omnext.net

Abstract. Over hoe de cirkel rond kwam.

1 Theorie

In 1983 ontmoette ik Doaitse voor het eerst tijdens het college Inleiding Programmeren in Utrecht. Het correct bewijzen van programma's had toen zijn volle aandacht in de met de hand geschreven syllabus. Later kregen we parserbouw en de opdracht om een grote Simpele vertaler aan te passen. De wereld was even gevuld met alleen maar parsers.

Omdat Simpele zaken mij wel aanspreken, ben ik dan ook op het onderwerp Hogere Orde Attributengrammatica's afgestudeerd. Het afstudeeronderwerp had nog zoveel onontdekte gebieden dat een promotie mogelijk was bij Doaitse. De eerste publicatie werd meteen geaccepteerd in Amerika en we hebben een mooie en bijzondere reis gemaakt naar New York, Chicago en Portland. Later volgden meer papers en reizen naar onder andere Ameland, Berlijn, Parijs, Praag en Passau. Het was een leuke en interessante tijd. Minder van wetenschappelijke aard maar zeker in mijn geheugen gegrift blijft de skivakantie met Doaitse, waar ik naast Informatica, ook mijn passie voor skiën heb ontdekt.

2 Theorie en praktijk

Na mijn promotie kwam ik terecht bij het SERC (Software Engineering Research Centre) waar Doaitse in het bestuur zat. Het SERC had als missie om de kloof (parsers!) op het gebied van software engineering tussen Universiteit en bedrijfsleven te verkleinen. We moesten ons geld verdienen uit de markt. En dat hebben we ook geprobeerd met parsers. Toen het millennium probleem eraan kwam dachten we dat het grote parser tijdperk nu werkelijk was aangebroken. De tweecijferige jaarvelden moesten opgespoord en gerepareerd worden. Helaas bleken grote bedrijven niet alleen maar in één taal programma's te schrijven. Ze hadden wel tientallen miljoenen regels in wel vijftig talen. Dus toen hebben we vanwege de tijdsdruk maar niet vijftig parsers gemaakt, maar wel een zeer succesvolle "grep++" ("grep" is het Unix commando waarmee files simpel zonder te parsen doorzocht kunnen worden). Heb deze oplossing ook aan Doaitse moeten vertellen, één van de moeilijkste momenten in mijn leven. Toch gloort er hoop aan de horizon.

3 Praktijk en conclusie

Na SERC kwam Omnext, een bedrijf gespecialiseerd in het transparant maken van software. Er wordt als het ware een röntgenfoto van de software gemaakt zodat inzichtelijk wordt hoe het onder andere met de onderhoudbaarheid is gesteld. En in dit geval gebruiken we wel parsertechnologie en zelfs Hogere Orde Attributengrammatica's.

De cirkel is rond, Doaitse bedankt!

Embedded Attribute Grammars using C++ Templates

Arie Middelkoop

amiddelk@gmail.com

Abstract. Attribute grammars have been implemented as a statically-typed, embedded domain specific language in several programming languages that feature meta programming facilities. C++ templates are considered to be an expressive meta programming language. In this article, we give an outline of such an embedding using C++ templates.

1 Introduction

This article considers what would be Doaitse's next 'best' programming language to embed attribute grammars in, when Haskell [1] is taken aside. Among the list of possibilities, we would normally not consider C++, but there is actually some justification for it:

- The embedding focuses on meta programming, which in case of C++ involve templates, at least when we assume that the embedding is to be statically typed. The template language of C++ is purely functional. This is a must-have for somebody that is such a prominent preacher of functional programming.
- The templates were not intended as a programming language, turning working on the edges of what is possible in practice into a small adventure.
- Their design and implementation in compilers is so poor that somebody should complain to the standardization committees and to compiler implementers to accept improvements.
- When writing a template program, one quickly runs into obscure compiler issues and platform-specific quirks, which is a reason to stay in contact with former colleagues and students.

There exists more serious motivation for embedding attribute grammars in C++. There are many compilers and transformation tools implemented in C++, including the large GCC compiler. A lightweight C++ embedding of attribute grammars can for example be useful in the description of transformation steps of the latter.

It is known for a long time that templates are an expressive meta language, so it is not surprising that an embedding is possible. To do so, we need to:

- Provide an API for the type-level description of an attribute grammar (as a type). In particular, we want to compose such a type out of individual fragments, such as nonterminal, production and attribute declarations (represented as types).
- Derive data types (e.g. classes in C++) for representing the tree as a function of the composed grammar.
- Derive the code for decorating the tree as a function of the composed grammar. Although we can only derive types using templates, this is not a limitation because those types can be classes containing functions. Those functions may call functions of other derived classes, thereby allowing the composition of blocks of code into arbitrary complex algorithms.

As a starting point for Doaitse, we sketch the first step in this article, but first we give an overview of template programming in the next section.

2 Preliminaries: Template Programming

The template language is purely functional [2], evaluated using a call-by-need strategy with maximal sharing, and untyped. The latter makes template programming a rather painful process, and proposals that address this issue (e.g., Concepts) have not been adopted yet. Otherwise, template programming is not unlike programming in any other functional language, albeit with a clumsy syntax, as we show in this section.

A precondition for programming with templates is a basic understanding of C++ type declarations. In particular, the use of typedefs plays a major role, as they encode the concept of let-bindings in templates. A confusing point is the frequent need to prefix type expressions that use the scope resolution operator `::` with the keyword **`typename`** to explicitly state that the identifier that is looked up is a type and not a value or function. As expected, the former is usually the case when writing type level computations. It is also worth noting that, unlike in C, structs are just classes with a default public visibility. In this article, object orientation and information hiding do not play an important role, hence we will mostly use structs for conciser code.

Programming with templates allows one to compute a type as a function of other types, and templates are the means to express those functions. The result of such a 'function' is restricted to class types (i.e. a struct). This restriction is not an obstacle as we see later. We show some templates by example.

The simplest form is a zero-place function, a constant, which is thus actually a conventional type that does not involve templates yet:

```
struct unit;   // optional forward declaration
struct unit {  // actual declaration
  // class declaration: body of the template
};
```

The class declaration can contain fields and methods, but also nested templates, and typedefs. In practice, this means that classes encode the concept of (dependent) records, and the scope resolution operator `::` as the mechanism to dereference or lookup the fields.

As a more complex example, the identity function is expressed as:

```
template<typename x>  // parameter list
struct identity;      // forward declaration

template<typename x>  // parameter list
struct identity {     // actual declaration
  typedef x type;     // field (type)
};
```

The template introduces a parameter `x`, which can be used in type expressions. The typedef binds the type expression `x` to the name `type`. The template can be called in type expressions:

```
identity<int>          // the struct itself
identity<int>::type    // the nested type
```

Thus, we can express other results than structs (e.g. primitive types such as **`int`**) by exposing them as fields. We follow the usual convention to store the actual result of the function as the field name `type`.

A typical template has the following form:

```
template<typename arg1, typename arg2>
```

```
struct my_fun {
  template< /* ... */ >
  struct nested_fun { /* ... */ };

  typedef /* ... type expr ... */ local1;
  typedef /* ... type expr ... */ local2;

  typedef /* ... type expr ... */ type;
}
```

As usual in C++, the scope goes from top to bottom, and forward declarations can be used to introduce recursion. A typical function application has the form of:

```
my_fun< /* type expr */, /* type expr */ >::type
```

When used inside a typedef, we need to make explicit that the name `type` represents the name of a type by prefixing it with the keyword **`typename`**:

```
typedef typename my_fun<int, char>::type result;
```

In general, templates take one or more types as arguments; in the example we showed two.

The syntax of the template language takes a while to getting used to, but in the end it is just functional programming in disguise. Note that currying is not supported this way, meaning that all function applications need to be fully saturated. The Boost MPL library [3] provides a slightly more involved encoding of functions so that currying can be expressed, but we do not need this feature in this article.

Pattern matching on function parameters can be encoded using template specialisation. Noting that the previous type `unit` represents a type level zero, and `identity` a type level successor, we can implement addition of type level natural numbers as follows:

```
template<typename x, typename y>
struct add;  // forward declaration

template<typename x>
struct add<x, unit> {   // y=unit
  typedef x type;
};

template<typename x, typename z>
struct add<x, identity<z> > {  // y=identity<z>
  typedef typename add<identity<x>, z>::type type;
};
```

The template body with the most specific instantiation is chosen, when the choice is unambiguous. Failure to provide such a case usually results in a cryptic error message.

An alternative implementation would wrap the identity around the result instead of around the first argument. The definition as given above is advantageous as it can be written more concisely using inheritance:

```
template<typename x, typename y>
struct add<x, identity<y> > : add<identity<x>, y> {
};
```

This particular construction often shows up in other template programs.

The Boost MPL library [3] provides some common abstractions to make programming with templates more convenient. It provides an if-then-else function, and type level maps with the following API:

```
map<>                                   // empty map
insert<mp,pair<key,value>>::type        // insert on map
has_key<mp,key>::type                   // member testing
fold<mp,state,binfun>::type             // fold function
```

To implement the DSL, we write functions that fold over the structure of the grammar representation, which consists largely of type level maps. It would be possible to express those folds with an attribute grammar, but it should be clear by now that we do not intend to cover meta-meta attribute grammars in this article.

3 Embedding

The API of the embedding consists of functions to construct an attribute grammar description, functions to access the attributes from rules of the grammar, conventions for encoding those rules, and functions to construct the tree and to obtain it synthesized attributes. We discuss each of these parts in this section.

Grammar construction. There are several ways to construct descriptions of attribute grammars, such as oriented per production or per aspect. We do not adopt a particular approach, and rather provide a low-level API upon which a higher-level API can be build.

Each of the functions in Figure 1 incorporate an additional asset (nonterminal, production, attribute, etc.) to a grammar description, and yield the extended grammar description. The starting point is an empty grammar, available as a constant (i.e. a type). The functions are all commutative, i.e. may appear in any order.

The difference between `decl_syn` and `define_syn` is that the former specifies the synthesized attributes of a nonterminal, whereas the latter gives their respective definitions per production of the nonterminal. The same distinction holds for `decl_inh` and `define_inh`, except that these are focused around children, i.e. occurrences of nonterminals in the right-hand side of a production.

The functions of the API take nonterminal, production, child and attribute names as parameters. Names in this setting are constants, i.e. types introduced for the purpose of being a unique identifier. Figure 2 shows the names that we use in the example later in this section.

Rule encoding. Another issue is how to represent the body of rules, which needs to be given as type `rule_def` to the functions `define_inh` and `define_syn`. We encode the body as a static function wrapped in a type. The body needs access to the tree, which we pass as parameter.

Moreover, the type of the tree is derived from the final grammar, not from the intermediate grammar that is available in the call to `define_inh` or `define_syn`. We solve this by an additional level of indirection, where the wrapped type is parametrized with the final grammar, so that a rule definition has the following structure:

```
struct my_rule {
  template<typename grammar_final>
  struct rule {
    typedef typename node_type<grammar_final,
```

```
template<typename name, typename grammar>
struct decl_nont;

template <typename nont_name, typename attr_name,
  typename attr_type, typename grammar>
struct decl_inh;

template <typename nont_name, typename attr_name,
  typename attr_type, typename grammar>
struct decl_syn;

template<typename nont_name, typename prod_name,
  typename term_type, typename grammar>
struct decl_prod;

template<typename nont_name, typename prod_name,
  typename child_name, typename child_nont_name,
  typename grammar>
struct decl_child;

template<typename nont_name, typename prod_name,
  typename attr_name,
  typename rule_def, typename grammar>
struct define_syn;

template<typename nont_name, typename prod_name,
  typename child_name, typename attr_name,
  typename rule_def, typename grammar>
struct define_inh;
```

Fig. 1. API for composing attribute grammar descriptions.

```
      nont_name, prod_name>::type node_type;

    static attr_type evaluate(node_type *node) {
      return /* ... */;
    };
  };
};

define_syn(n_name, p_name, s_name, my_rule, some_g);
```

The names `rule` and `evaluate` cannot be chosen freely, so that when deriving the evaluation code from the final grammar we can use a statement as follows to call the body of the rule:

```
val = rule_def::rule<final_grammar>::evaluate(node);
```

The body of the evaluate function can access the values of attributes via the `node` parameter, using the following API functions in the body of rules:

```
inh_value<node_type, attr_name>::type::get(node)
syn_value<node_type, child_name,
  attr_name>::type::get(node)
```

```
struct n_root {}; struct p_root {}; struct c_root {};

struct n_tree {};
struct p_leaf {}; struct p_bin {};
struct c_left {}; struct c_right {};

struct i_depth {}; struct s_count {};
```

Fig. 2. Names used in the example.

Note that `get` is a (static) function, not a type.

Example. Before continuing, we provide an example to demonstrate the embedding using the functions as given so far. The example is about an attribute grammar for binary trees where a synthesized attribute `s_count` represents the total number of leaf nodes at a depth greater than 2 given initial depth as inherited attribute `i_depth`. The example is separated in two parts: Figure 3 contains the description of the grammar, and Figure 4 and Figure 5 show some of the rules.

We imposed the following conventions to simplify the API: starting nonterminals may only have synthesized attributes, and each production has exactly one terminal, whose values thus have to be available in each node of the tree. Using this convention, inherited attributes of the root can be encoded as terminals of the root node, zero terminals using the type `unit`, and multiple terminals using a structured type. In the example, the starting nonterminal `n_root` does indeed not take any inherited attributes, and all productions have `unit` as type of the terminal.

Tree construction. Figure 6 demonstrates some templates for constructing the tree. We describe these templates in more detail below.

The following two templates are used to obtain types related to a production and nonterminal respectively:

```
node_type<grammar, nont_name, prod_name>::type
tree_type<grammar, nont_name>::type
```

Each `node_type` is a subclass of `tree_type` when given the same nonterminal name. To construct a node of the tree, we use the C++ constructor of the `node_type`, that additionally takes the value of the terminal as parameter. We use the `tree_type` when the distinction between particular node types is irrelevant, which is e.g. the case when we attach a subtree to a node. We use the template `attach` for this purpose:

```
attach<grammar, nont_name, prod_name, child_name
  ::type::set(parent, child);
```

Here, `parent` must be a pointer to a node of type:

```
node_type<grammar, nont_name, prod_name>::type
```

and `child` must be a pointer to any node that is a subclass of:

```
tree_type<grammar, name>::type
```

where `name` is the nonterminal name given to the declaration of `child_name`. In practice, these pointers should be shared pointers of C++, to facilitate automatic garbage collection.

To obtain the synthesized attributes of the root, we use the template:

```
typedef empty_grammar g0;
typedef decl_nont<n_root, g0>::type g1;
typedef decl_nont<n_tree, g1>::type g2;
typedef decl_syn<n_tree, s_count, int,
  g2>::type g3;
typedef decl_inh<n_tree, i_depth, int,
  g3>::type g4;
typedef decl_prod<n_root, p_root,
  unit, g4>::type g5;
typedef decl_prod<n_tree, p_leaf,
  unit, g5>::type g6;
typedef decl_prod<n_tree, p_bin,
  unit, g6>::type g7;
typedef decl_child<n_root, p_root, c_root,
  n_tree, g7>::type g8;
typedef decl_child<n_tree, p_bin, c_left,
  n_tree, g8>::type g9;
typedef decl_child<n_tree, p_bin, c_right,
  n_tree, g9>::type g10;
typedef define_syn<n_tree, p_leaf, s_count,
  r_zero, g10>::type g11;
typedef define_syn<n_tree, p_bin, s_count,
  r_sum, g11>::type g12;
typedef define_inh<n_root, p_root, c_root,
  i_depth, r_init, g12>::type g13;
typedef define_inh<n_tree, p_bin, c_left,
  i_depth, r_down, g13>::type g14;
typedef define_inh<n_tree, p_bin, c_right,
  i_depth, r_down, g14>::type g15;
typedef g15 g_final;
```

Fig. 3. Example: grammar description.

```
result<grammar, nont_name, syn_name>
  ::type::get(root)
```

The function extracted from the type of the template decorates the tree with those attributes necessary to obtain the synthesized attribute `syn_name`.

4 Implementation

We leave the implementation as an exercise to the reader. Instead we give some hints.

Type-level representation. The attribute grammars can be represented as a map of `nont_decl`s, containing two maps of respectively inherited and synthesized `attr_decl`s, and a map of `prod_decl`s, which contain a map of `syn_def`s, and a map of `child_decl`s, each containing a map of `inh_def`s. These types serve as records, which we can express straightforwardly as templates, e.g.:

```
template<typename n, typename iattrs,
  typename sattrs, typename ps>
```

```
struct r_zero {
  template<typename g>
  struct rule {
    typedef typename node_type<g, n_tree, p_leaf>
      ::type t_node;

    int evaluate(t_node *node) {
      return 0;
    }
  };
};
```

Fig. 4. Example: trivial rule.

```
struct nont_decl {
  typedef n nont_name;
  typedef iattrs inhs;
  typedef sattrs syns;
  typedef ps prods;
};

template<typename n, typename t, typename sdefs,
  typename cs>
struct prod_decl {
  typedef n prod_name;
  typedef t term_type;
  typedef sdefs syn_defs;
  typedef cs children;
};
```

Some of the parameters will be maps (e.g. `iattrs`), which is not specified in the code because the template language is untyped.

Inheritance. When using a generator-based approach, it is possible to generate code that is type safe, i.e. does not use any unsafe features nor type casts. This code exploits inheritance (virtual methods) and nested classes to provide interfaces to attributes. For example, a node obtains from its parent an object that provides getter-functions of the node's inherited attributes that are to be computed by the parent. By using such interfaces, a node does not need to know the concrete type of its parent. Similarly, one can envision such a construction between a node and its children.

When using an embedding via templates, however, inheritance is not a viable option. To derive an interface-type containing a list of functions, we would use some nesting of types, either via fields or via inheritance. In the first case, inheritance does not work for fields, and in the latter case inheritance is already in use. Instead, one can label nodes with a runtime tag that specifies to which production the node belongs, and at which child position the node occurs at its parent. The appropriate code can then be looked up in various tables using these identifiers, similar to how virtual methods are implemented using vtables. By inspecting the tag we then discover the concrete type of the node. Note that this requires a type cast to convince the type checker. Fortunately, this cast is in the library code, and we can prove externally that it is always safe.

```
struct r_sum {
  template<typename g>
  struct rule {
    typedef typename node_type<g, n_tree, p_bin>
      ::type t_node;

    int evaluate(node_type<t_node *node) {
      int d = inh_value<t_node, i_depth>
                ::type::get(node);
      int l = syn_value<t_node, c_left, s_count>
                ::type::get(node);
      int r = syn_value<t_node, c_right, s_count>
                ::type::get(node);

      if (d > 2)
        return l + r;
      else
        return 0;
    }
  };
};
```

Fig. 5. Example: complex rule.

Numbering productions. To number the productions as mentioned above, we fold over the nonterminal and production declarations and compute a map that assigns to pairs of nonterminal and production name an increasing number. Figure 7 demonstrates.

5 Conclusion

We showed in this article how attribute grammars can be described using C++ templates. The implementation is up to the reader. As encouragement, we can furthermore say that the reader will easily run into obstacles imposed by the compiler, including slow compilation times and insufficient memory...

References

1. Viera, M., Swierstra, S.D.: Attribute grammar macros. In: SBLP. (2012) 150–164
2. Sinkovics, A.: Nested lambda expressions with let expressions in C++ template metaprograms. ENTCS **279**(3) (2011) 27–40
3. Gurtovoy, A., Abrahams, D.: The boost C++ meta programming library (2002)

```
// allocate the nodes
node_type<g_final, n_root, p_root>
  ::type root(unit);
node_type<g_final, n_tree, p_leaf>
  ::type middle(unit);
node_type<g_final, n_tree, p_bin>
  ::type left, right(unit);

// construct the tree
attach<g_final, n_root, p_root, c_root>
  ::type::set(&root, &middle);
attach<g_final, n_tree, p_bin, c_left>
  ::type::set(&middle, &left);
attach<g_final, n_tree, p_bin, c_right>
  ::type::set(&middle, &right);

// obtain the resulting attribute
int d = result<g_final, n_root, s_depth>
          ::type::get(&root);
```

Fig. 6. Example: tree construction.

```
template<typename grammar>
struct prod_numbers {
  template<typename mp, typename n_pair>
  struct handle_nont {
    typedef n_pair::snd n_decl;

    template<typename mp, typename p_pair>
    struct handle_prod {
      typedef p_pair::snd p_decl;
      typedef pair<n_decl::nont_name,
        p_decl::prod_name> key;
      typedef typename size<mp>::type tag;
      typedef typename insert<mp, key, tag>
        ::type result;
    };

    typedef typename fold<n_decl::prods,
      mp, quote2<handle_prod> >::type type;
  };

  typedef typename fold<grammar, map<>,
    quote2<handle_nont> >::type type;
};
```

Fig. 7. Template that derives a map with production tags.

A note on indirection

Eelco J. Dijkstra

eelco.j.dijkstra@gmail.com

Abstract. Indirection is a powerful mechanism, used in information and communication systems from the lowest hardware level to the highest levels like the web. In this note, we describe some of the principles and uses of this mechanism. As an exercise, we try to design a construction for anonymous and private anonymous web-shopping.

1 Introduction

One of the important lessons I learned from Doaitse concerns the significant role of indirection in Informatics and ICT-based systems. In his usual style he stated that some very large percentage of the ICT-problems could be solved with an extra level of indirection. While the percentage he mentioned probably is an exaggeration, the core of the message remains valid.

Doaitse has spent a large part of his scientific career in the exploitation of indirection, mainly in the context of functional programming. Assuming that this domain is addressed elsewhere, we will focus on a number of complementary aspects and examples.

In this paper, we will give a very short introduction to some of the principles of naming and indirection. After that, as an exercise we try to use these principles in the design of a construction for anonymous shopping on the web.

2 Preliminaries

2.1 Some forms of indirection

By *naming* an object, concrete or abstract, we are able to communicate and to reason (compute) about that object. Naming is used at the lowest hardware level, e.g., in the form of the address of a memory location; and at a very high level, in the form of a URI (web address). Such a name identifies an object or a value. Obtaining the value corresponding to a name is called *dereferencing* the name. Dereferencing a memory address corresponds to fetching the content of the memory location. Dereferencing a URI corresponds to fetching the web page.

Usually, the binding between the name and the object is *fixed*, for the scope in which the name is assumed to be valid. This does not imply that the dereferencing of a name always yields the same value or representation: when the state of the object has been altered, this usually is reflected in its representation.

Next to names for things, we introduce *names for roles*. The binding between a role and the object fulfilling the role usually varies over time. Examples of such role-names are the parameters of a function, and the properties (fields) of an object. The value corresponding to a role-name may be the *name* of an object: then, the role-name provides an extra level of indirection above the object name.

A name can be used in two different ways: it can be used and passed along as a name, and it can be dereferenced. Sometimes, we have a special notation to distinguish these two -

such as a function name versus a function call, or a pointer as pointer versus a dereferenced pointer; in other cases, the context provides the information.

An interface is another example of role-based naming: the roles of the interface are described in a way that decouples its use from its implementation.

Functions, as deferred computation, can be seen as a form of indirection. A function call then is the analogue of a name dereference. Using a function, we can postpone the computation, e.g., to a moment where we actually need the result of the computation.

We may use a named function, with its signature, as an interface separating the use and the implementation of the function. Complex interfaces may be described by a list or table of functions.

Interpretation is a special form of indirection: instead of direct execution, e.g. by the underlying hardware, we use an interpreter to bridge the level of the input "language" and the hardware. Compilation can be considered as an extension of interpretation.

2.2 Some examples of indirection

We present a number of examples of the use of indirection. It is not our intention to be complete: we just want to demonstrate that indirection is used at all kinds of levels, with various intentions.

Pointers or object-references are used for complex and flexible data structures, like trees and graphs. In particular, such data structures may contain shared elements and cycles.

Interfaces demonstrate *separation of concerns* between the use and the implementation of the interface. Changes in the implementation have no or minimal consequences for the use, and vice versa.

Modern computer hardware is based on an instruction set architecture, where the interface of the hardware is specified in the form of an instruction set. This instruction set decouples the hardware-implementation from the software using the hardware. This decoupling allows for variation and evolution of the hardware implementation. At the other side of the interface, the variation and evolution of the software is independent of the hardware.

Virtualization - e.g., in the form of virtual memory, or in the form of virtual machines, is used to achieve flexibility, and to separate different subsystems, users, and application domains. This separation can e.g. be used to increase the security of the different parts: failure, vulnerability and security problems can be contained.

Programming languages, interpretation and compilation is used between the human level, where communication and understanding are central aspects, and the efficient execution at the hardware level.

3 Using indirection for anonymous web shoppping

Currently, virtual shopping on the web is much less anonymous and private than physical shopping. The information that shops and other agents like banks and transport services have about their customers is a potential and sometimes actual intrusion to the privacy of these customers.

We express the information concerning a transaction in a number of relations. We denote a relation with name `rel` as `A rel: B`. The relations that are visible to the shop in the case of a transaction X are:

- `X hasGoods: G` - the combination of goods involved in the transaction;
- `X hasPrice: P` - the price of the goods;

– `X hasCustomer: C` - the customer, usually identified by a name or an e-mail address.
– `X hasBankAccount: B` - the bank account of the customer;
– `X hasAddress: A` - the address where the goods are to be delivered.

Many of these relations involve a *permanent name* or identifier, such as an e-mail address, a physical address, or a bank account. While such a permanent name may be anonymous, it enables tracing the behaviour of a single customer over many transactions. The profile of a user obtained in that way can be used in other interactions with the same user - even if the personal name of the user is not known. Moreover, this profile can also be used to identify the user, and to find his personal name.

In order to increase the anonymity and privacy of a user in such a transaction, we use the following principles:

– instead of permanent names, use *temporary names* where possible. Such a temporary name refers via an extra level of indirection to a permanent name. Dereferencing of the temporary name will only be made possible for a single agent - e.g., by encrypting the information using the public key of that agent, or by using another form of secure exchange with that agent.
– relations of the form `X rel: Y` are split into two (or sometimes three) successive relations: `X rel0: tmp` and `tmp rel1: Y`. The intermediate name `tmp` is a temporary name. These successive relations are known to different agents. These agents are considered to be trusted: they keep their information hidden from the other agents. Splitting a relation may imply the need for an extra intermediate agent, to handle an intermediate step.

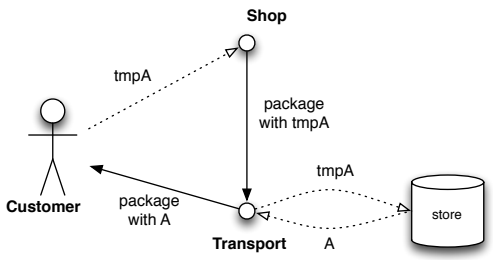


Fig. 1. Customer, Shop and Transport

As an example, we show how the relation `X hasAddress: A` can be hidden from the shop. We split this relation into `X hasAddress0: tmpA` and `tmpA hassAddress1: A`. In the communication between the (anonymous) customer and the shop, the customer provides the shop with the temporary name (address) `tmpA`. As described above, this temporary address cannot be dereferenced by the shop - only by the transport agent. This temporary address is communicated from the shop to the transport agent, e.g, as a label on the package containing the goods. The transport agent dereferences the temporary address, and delivers the package to the customer.

The only transaction information visible to the transport agent is the address of the customer and the shop involved. If a shop has a very broad domain of goods to sell, the information about the shop does not represent significant information. However, if it concerns a small shop, with a very specific product offering, the information identifying the shop

may be significant. In that case, we may need an extra level of indirection to hide the shop information for the transport agent: the relation is split further, and another transport agent is introduced. Then, the first transport agent has information about the shop involved, but not about the customer's address. The second transport agent has information about the customer's address, but not about the shop.

In a similar way, the relations concerning the payment can be hidden for the shop, by splitting these relations, and by introducing one or more extra agents between the customer's bank and the shop.

4 Concluding remarks

Indirection connects and separates at the same time. This connection serves the composition of complex data structures, computation structures, and networks. This separation enables the individual treatment of the parts: we may implement these parts, reason about these, change, adapt, replace, and evolve these parts independently of the rest of the system. Thanks to the use of indirection, the resulting systems are more reliable, secure, private, resilient, adaptable and evolvable.

Derivation of an imperative unification algorithm

Lex Bijlsma

lex.bijlsma@acm.org
Faculteit Informatica, Open Universiteit

Abstract. We present a systematic derivation in the calculational style of an imperative algorithm for unification.

1 Introduction

Unification algorithms [6, 5, 4, 1] are essential to the implementation of logic programming languages. The derivation of unification algorithms by calculational means would seem to be of interest, but is hampered by the style in which the underlying theory is usually presented, both in original papers and in subsequent textbooks. To give the reader a taste of the problems encountered here, I quote from [3] the definition of composition of substitutions.

> **Definition** Let $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ and $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$ be substitutions. Then the *composition* $\theta\sigma$ of θ and σ is the substitution obtained from the set
>
> $$\{u_1/s_1\sigma, \ldots, u_m/s_m\sigma, v_1/t_1, \ldots, v_n/t_n\}$$
>
> by deleting any binding $u_i/s_i\sigma$ for which $u_i = s_i\sigma$ and deleting any binding v_j/t_j for which $v_j \in \{u_1, \ldots, u_m\}$.

From the point of view of program derivation, the problems with such a definition are threefold. In the first place, the composition is defined by an operational description of a method for its determination, not by a closed formula that would lend itself to calculation. Secondly, the number of identifiers, subscripts, and ellipses needed is very high for such a fundamental concept. Finally, in this definition a substitution is not a mapping and the composition of substitutions is not function composition; this deprives us of the possibility to exploit associativity and other properties of function composition.

In this note we follow a different plan of attack. In our approach substitutions are mappings and we do exploit the properties of function composition. It turns out that this decision facilitates reasoning about these subjects to such an extent that it becomes possible to give a calculational derivation of a unification algorithm, as is demonstrated in Section 5.

2 Terms and subterms

We are given a set Var and, for every natural number n, a set Fu_n. The elements of Var are called *variables* and denoted by dummies x and y; the elements of Fu_n are called *n-ary function symbols* and denoted by dummies f and g.

Definition 1 *Set Terms is defined as the smallest solution of equation*

$$T: \quad Var \ \cup \ (\cup n : n \in \mathbf{N} : Fu_n \times T^n) \ \subseteq \ T \quad . \tag{1}$$

□

The smallest solution of (1) exists because the left hand side, considered as a function of T, is continuous (though not disjunctive).

We assume that

$$(\forall T :: Var \ \cap \ (\cup n : n \in \mathbf{N} : Fu_n \times T^n) = \emptyset) \quad , \tag{2}$$

so that the union in (1) is disjoint.

The elements of *Terms* are called *terms*, the elements of $Terms^n$ are called *vectors*. Dummies t, s, r, q will be reserved for terms and dummies v, w for vectors.

For every set T, we identify T^n with the function space $[0\,..\,n) \to T$. Hence, for φ any function defined on terms and v a vector, the notation $\varphi \circ v$ is well-defined and denotes the vector obtained by applying φ to every component of v.

Any subset T of *Terms* satisfying

$$Var \subseteq T \quad , \tag{3}$$

$$(\forall n, f, v : f \in Fu_n : v \in T^n \ \Rightarrow \ (f, v) \in T) \quad , \tag{4}$$

is a solution of (1), hence, by minimality, equals *Terms* itself. As usual, applications of this principle are called proofs by structural induction, and we refer to (3) and (4) respectively as the base and the step of the induction.

Lemma 2 *For* $n \in \mathbf{N}$, $f \in Fu_n$, $v \in Terms^n$, $i \in [0\,..\,n)$ *we have*

$$(f, v) \neq v.i \quad .$$

Proof. By structural induction on $v.i$. The base, $v.i \in Var$, is dealth with by (2). For the step, we write $v.i = (g, w)$ with $g \in Fu_m$, $w \in Terms^m$. Then

$$\begin{array}{ll}
 & (f, v) \neq v.i \\
\equiv & \quad \{\ v.i = (g, w)\ \} \\
 & (f, v) \neq (g, w) \\
\Leftarrow & \quad \{\text{ordered pairs}\} \\
 & v \neq w \\
\Leftarrow & \quad \{\text{Leibniz}\} \\
 & m \neq n \ \vee \ (m = n \ \wedge \ v.i \neq w.i) \\
\equiv & \quad \{\ v.i = (g, w)\ \} \\
 & m \neq n \ \vee \ (m = n \ \wedge \ (g, w) \neq w.i) \\
\equiv & \quad \{\text{induction hypothesis}\} \\
 & m \neq n \ \vee \ (m = n \ \wedge \ true) \\
\equiv & \quad \{\} \\
 & true \quad .
\end{array}$$

□

Definition 3 *Relation* $\prec$ *is defined on Terms by*

$$t \prec x \ \equiv \ false \quad , \tag{5}$$

$$t \prec (f, v) \ \equiv \ (\exists i :: t = v.i \ \vee \ t \prec v.i) \quad , \tag{6}$$

where $x \in Var$, *and* $f \in Fu_n$, $v \in Terms^n$ *for some natural* n. *Dummy* i *ranges over* $[0\,..\,n)$. □

Sometimes it will be advantageous to use the abbreviation

$$t \preceq s \;\equiv\; t = s \;\vee\; t \prec s \quad .$$

This formula is always used implicitly in that its application does not merit a separate line in derivations. When $t \preceq s$, we say that t is a *subterm* of s.

Proposition 4 *Relation $\prec$ is transitive.*

Proof. We prove

$$t \prec s \;\wedge\; s \prec r \;\Rightarrow\; t \prec r$$

by structural induction on r. For the base, $r \in Var$, both sides of the implication are false on account of (5). For the step, we have

$$\begin{array}{ll}
 & t \prec s \;\wedge\; s \prec (f, v) \\
\equiv & \quad \{(6) \text{ with } t := s\,\} \\
 & t \prec s \;\wedge\; (\exists i :: s = v.i \;\vee\; s \prec v.i) \\
\equiv & \quad \{\wedge \text{ over } \exists \text{ and } \vee\text{; term split}\} \\
 & (\exists i :: t \prec s \;\wedge\; s = v.i) \;\vee\; (\exists i :: t \prec s \;\wedge\; s \prec v.i) \\
\Rightarrow & \quad \{\text{Leibniz } \| \text{ induction hypothesis}\} \\
 & (\exists i :: t \prec v.i) \\
\Rightarrow & \quad \{(6)\} \\
 & t \prec (f, v) \quad .
\end{array}$$

□

Proposition 5 *Relation $\prec$ is antireflexive, i.e., for any term t,*

$$t \prec t \;\equiv\; \mathit{false} \quad .$$

Proof. By structural induction on t. The base follows from (5). As for the step,

$$\begin{array}{ll}
 & (f, v) \prec (f, v) \\
\equiv & \quad \{(6) \text{ with } t := (f, v)\,\} \\
 & (\exists i :: (f, v) = v.i \;\vee\; (f, v) \prec v.i) \\
\equiv & \quad \{\text{Lemma 2}\} \\
 & (\exists i :: (f, v) \prec v.i) \\
\equiv & \quad \{(6) \text{ with } t := v.i\,\} \\
 & (\exists i :: v.i \prec (f, v) \;\wedge\; (f, v) \prec v.i) \\
\Rightarrow & \quad \{\text{Proposition 4}\} \\
 & (\exists i :: v.i \prec v.i) \\
\equiv & \quad \{\text{induction hypothesis}\} \\
 & \mathit{false} \quad .
\end{array}$$

□

3 Substitutions

Definition 6 *A substitution is a function φ of type $Terms \to Terms$ that satisfies*

$$\varphi.(f, v) = (f, \varphi \circ v) \tag{7}$$

for $f \in Fu_n$, $v \in Terms^n$. □

The set of all substitutions will be denoted by Sub; dummies φ and ψ are reserved for substitutions. Notice that a substitution is completely determined by the value it takes on the variables.

Proposition 7 *Sub is closed under function composition.*

Proof. For $\varphi \in Sub$, $\psi \in Sub$, $f \in Fu_n$, $v \in Terms^n$ we have

$$
\begin{array}{ll}
 & (\varphi \circ \psi).(f, v) \\
= & \quad \{\text{definition of } \circ\} \\
 & \varphi.(\psi.(f, v)) \\
= & \quad \{(7) \text{ with } \varphi := \psi \text{ , using } \psi \in Sub\} \\
 & \varphi.(f, \psi \circ v) \\
= & \quad \{(7) \text{ with } v := \psi \circ v \text{ , using } \varphi \in Sub\} \\
 & (f, \varphi \circ (\psi \circ v)) \\
= & \quad \{\text{associativity of } \circ\} \\
 & (f, (\varphi \circ \psi) \circ v) \quad ,
\end{array}
$$

so $\varphi \circ \psi \in Sub$. □

Next we show that substitutions are monotonic with respect to relation $\prec$ from the preceding section.

Proposition 8 *For substitution φ and terms t and s ,*

$$t \prec s \;\Rightarrow\; \varphi.t \prec \varphi.s \quad .$$

Proof. By structural induction on s . In the base, $s \in Var$, the antecedent is false by (5). For the step, we reason as follows:

$$
\begin{array}{ll}
 & \varphi.t \prec \varphi.(f, v) \\
\equiv & \quad \{(7)\} \\
 & \varphi.t \prec (f, \varphi \circ v) \\
\equiv & \quad \{(6) \text{ with } t, v := \varphi.t,\ \varphi \circ v\} \\
 & (\exists i :: \varphi.t = \varphi.(v.i) \ \vee\ \varphi.t \prec \varphi.(v.i)) \\
\Leftarrow & \quad \{\text{Leibniz } \| \text{ induction hypothesis}\} \\
 & (\exists i :: t = v.i \ \vee\ t \prec v.i) \\
\equiv & \quad \{(6)\} \\
 & t \prec (f, v) \quad .
\end{array}
$$

□

Combination of Propositions 5 and 8 yields the following result:

Corollary 9 *For substitution φ and terms t and s ,*

$$t \prec s \;\Rightarrow\; \varphi.t \neq \varphi.s \quad .$$

□

Now we introduce a special class of substitutions we shall be particularly interested in, namely those involving only one variable.

Definition 10 *For variable x and term t , we denote by $(x \leftarrow t)$ the unique substitution satisfying, for $y \in Var$,*

$$(x \leftarrow t).y = \begin{cases} t & \text{if } y = x \quad , \\ y & \text{if } y \neq x \quad . \end{cases} \tag{8}$$

□

Here unicity is because a substitution is determined by its values on *Var*.

Lemma 11 $\neg(x \preceq s) \;\Rightarrow\; (x \leftarrow t).s = s$.

Proof. By structural induction on s. For the base, let s be a variable y. Then

$$
\begin{array}{ll}
 & (x \leftarrow t).y = y \\
\Leftarrow & \quad \{(8)\} \\
 & y \neq x \\
\equiv & \quad \{(5)\} \\
 & \neg(x \preceq y) \quad .
\end{array}
$$

For the step, we have

$$
\begin{array}{ll}
 & (x \leftarrow t).(f, v) = (f, v) \\
\equiv & \quad \{(7)\} \\
 & (f, (x \leftarrow t) \circ v) = (f, v) \\
\equiv & \quad \{\text{equality of ordered pairs}\} \\
 & (x \leftarrow t) \circ v = v \\
\equiv & \quad \{\text{equality of vectors}\} \\
 & (\forall i :: (x \leftarrow t).(v.i) = v.i) \\
\Leftarrow & \quad \{\text{induction hypothesis}\} \\
 & (\forall i :: \neg(x \preceq v.i)) \\
\equiv & \quad \{\text{de Morgan; (6)}\} \\
 & \neg(x \prec (f, v)) \\
\equiv & \quad \{(2)\} \\
 & \neg(x \preceq (f, v)) \quad .
\end{array}
$$

□

Combination of (8) with $y := x$ and Lemma 11 with $s := t$ yields

Corollary 12 $\neg(x \prec t) \;\Rightarrow\; (x \leftarrow t).t = t$. □

Proposition 13 $\varphi.x = \varphi.t \;\equiv\; \varphi = \varphi \circ (x \leftarrow t)$.

Proof of LHS⇒ RHS We prove

$$(\forall s :: \varphi.s = (\varphi \circ (x \leftarrow t)).s)$$

by structural induction on s. The base is split into two cases: first, for s a variable, say y, distinct from x we have

$$
\begin{array}{ll}
 & (\varphi \circ (x \leftarrow t)).y \\
= & \quad \{\text{function composition}\} \\
 & \varphi.((x \leftarrow t).y) \\
= & \quad \{(8), \text{using } y \neq x\} \\
 & \varphi.y \quad ,
\end{array}
$$

whereas

$$
\begin{array}{ll}
 & (\varphi \circ (x \leftarrow t)).x \\
= & \quad \{\text{function composition}\} \\
 & \varphi.((x \leftarrow t).x) \\
= & \quad \{(8)\} \\
 & \varphi.t \\
= & \quad \{\text{LHS}\} \\
 & \varphi.x \quad .
\end{array}
$$

For the step, we observe

$$
\begin{array}{ll}
 & (\varphi \circ (x \leftarrow t)).(f, v) \\
\equiv & \quad \{(7)\} \\
 & (f, \varphi \circ (x \leftarrow t) \circ v) \\
\equiv & \quad \{\text{induction hypothesis}\} \\
 & (f, \varphi \circ v) \\
\equiv & \quad \{(7)\} \\
 & \varphi.(f, v) \quad .
\end{array}
$$

Proof of RHS⇒ LHS

$$
\begin{array}{ll}
 & \varphi.x = \varphi.t \\
\equiv & \quad \{(8)\} \\
 & \varphi.x = (\varphi \circ (x \leftarrow t)).x \\
\Leftarrow & \quad \{\text{function application}\} \\
 & \varphi = \varphi \circ (x \leftarrow t) \quad .
\end{array}
$$

□

As a specialization of Proposition 13 for the case $\varphi = (x \leftarrow t)$ we have

Proposition 14 *Let x be a variable and t a term such that $\neg(x \prec t)$ holds. Then $(x \leftarrow t)$ is idempotent.*

Proof.

$$
\begin{array}{ll}
 & (x \leftarrow t) \circ (x \leftarrow t) = (x \leftarrow t) \\
\equiv & \quad \{\text{Proposition 13 with } \varphi := (x \leftarrow t)\,\} \\
 & (x \leftarrow t).x = (x \leftarrow t).t \\
\equiv & \quad \{(8) \parallel \text{Corollary 12, using } \neg(x \prec t)\,\} \\
 & t = t \\
\equiv & \quad \{\} \\
 & \mathit{true} \quad .
\end{array}
$$

□

4 Unification

Before embarking on a formal definition of unification, we introduce two notations. The first one is what functional programmers would call the *map* function: for any set A and any mapping F defined on A, we put

$$F * A = \{a : a \in A : F.a\} \quad .$$

The operator $*$ is given a binding power higher than that of set union, but lower than that of function application. For A a set of sets, we write $(F*) * A$ as $F ** A$. Of the properties of $*$ we mention

$$F * \{a\} = \{F.a\} \quad , \tag{9}$$

$$F * (A \cup B) = F * A \cup F * B \quad , \tag{10}$$

$$(F*) \circ (G*) = ((F \circ G)*) \quad . \tag{11}$$

The second notation we introduce is just an ad hoc abbreviation: for any set A we put

$$[A] \;\equiv\; (\forall a, b : a \in A \;\wedge\; b \in A : a = b) \quad .$$

Thus, $[A]$ signifies that A is either empty or a singleton. One immediately obtains

$$[A] \;\wedge\; B \subseteq A \;\;\Rightarrow\;\; [B] \quad . \tag{12}$$

Now we are ready to define unification.

Definition 15 *For $\varphi \in Sub$, $T \subseteq Terms$ we say that φ unifies T if*

$$[\varphi * T]$$

holds. □

Proposition 16 *Let T be a subset of Terms, $t \in T$ and $x \in T \cap Var$. Then*

$$[\varphi * (T \cup \{x\})] \;\Rightarrow\; \neg(x \prec t) \;\wedge\; \varphi = \varphi \circ (x \leftarrow t) \quad .$$

Proof.

$$\begin{array}{ll} & [\varphi * (T \cup \{x\})] \\ \equiv & \quad \{(10)\} \\ & [\varphi * T \;\cup\; \varphi * \{x\}] \\ \equiv & \quad \{(9)\} \\ & [\varphi * T \;\cup\; \{\varphi.x\}] \\ \Rightarrow & \quad \{(12), \text{ using } \varphi.t \in \varphi * T\,\} \\ & [\{\varphi.x,\, \varphi.t\}] \\ \equiv & \quad \{\text{definition of } *\,\} \\ & \varphi.x = \varphi.t \\ \Rightarrow & \quad \{\text{Corollary 9} \;\|\; \text{Proposition 13}\} \\ & \neg(x \prec t) \;\wedge\; \varphi = \varphi \circ (x \leftarrow t) \quad . \end{array}$$

□

For the next lemma, we need some notation. For $T \subseteq Terms \setminus Var$ we define

$$Func.T = \{f, v : (f, v) \in T : f\} \quad . \tag{13}$$

For $T \subseteq Fu_n \times Terms^n$ we define

$$Comp.T = \{i : 0 \leq i < n : \{f, v : (f, v) \in T : v.i\}\} \quad . \tag{14}$$

Finally, for $\mathcal{S}$ a set of set of terms, we define

$$\varphi \;\underline{\mathsf{un}}\; \mathcal{S} \;\equiv\; (\forall S : S \in \mathcal{S} : [\varphi * S]) \quad . \tag{15}$$

Lemma 17 *For $T \subseteq Terms \setminus Var$, we have*

$$[\varphi * T] \;\equiv\; [Func.T] \;\wedge\; \varphi \;\underline{\mathsf{un}}\; Comp.T \quad .$$

Proof.

$$
\begin{array}{ll}
 & [\varphi * T] \\
\equiv & \quad \{\ T \subseteq \mathit{Terms} \setminus \mathit{Var}\ \} \\
 & [\{f, v : (f, v) \in T : \varphi.(f, v)\}] \\
\equiv & \quad \{(7)\} \\
 & [\{f, v : (f, v) \in T : (f, \varphi \circ v)\}] \\
\equiv & \quad \{\text{equality of ordered pairs}\} \\
 & [\{f, v : (f, v) \in T : f\}] \ \wedge \ [\{f, v : (f, v) \in T : \varphi \circ v\}] \\
\equiv & \quad \{(13)\} \\
 & [\mathit{Func}.T] \ \wedge \ [\{f, v : (f, v) \in T : \varphi \circ v\}] \\
\equiv & \quad \{\text{equality of vectors of the same dimension}\} \\
 & [\mathit{Func}.T] \ \wedge \ (\forall i :: [\{f, v : (f, v) \in T : \varphi.(v.i)\}]) \\
\equiv & \quad \{(14)\} \\
 & [\mathit{Func}.T] \ \wedge \ (\forall S : S \in \mathit{Comp}.T : [\varphi * S]) \\
\equiv & \quad \{(15)\} \\
 & [\mathit{Func}.T] \ \wedge \ \varphi \ \underline{\mathsf{un}} \ \mathit{Comp}.T \quad .
\end{array}
$$

□

5 An algorithm

In this section, we consider the following programming problem: given is a set R of terms, and we are to compute the boolean value of

$$(\exists \varphi :: [\varphi * R]) \quad .$$

Because of the appearance of the term $\varphi \ \underline{\mathsf{un}} \ \mathit{Comp}.T$ in the right hand side of Lemma 17, we see that it is not wise to study unification of one set only. Therefore we introduce the following predicate: for $\mathcal{S}$ a set of sets of terms, we put

$$\mathit{Unif}.\mathcal{S} \ \equiv \ (\exists \varphi :: \varphi \ \underline{\mathsf{un}} \ \mathcal{S}) \quad . \tag{16}$$

The programming problem can now be phrased as the computation of $\mathit{Unif}.\{R\}$. Getting recurrence relations for *Unif* is easy once we have Proposition 16 and Lemma 17. Here is the translation of Proposition 16 in terms of *Unif*:

Lemma 18 *Let $\mathcal{S}$ be a set of sets of terms. Let T be an element of $\mathcal{S}$, with $t \in T$ and $x \in T \cap \mathit{Var}$. Then*

$$\mathit{Unif}.\mathcal{S} \ \equiv \ \neg(x \prec t) \ \wedge \ \mathit{Unif}.((x \leftarrow t) ** \mathcal{S}) \quad .$$

Proof. With φ' short for $\varphi \circ (x \leftarrow t)$, we have

$$
\begin{array}{ll}
 & \mathit{Unif}.\mathcal{S} \\
\equiv & \quad \{(16)\} \\
 & (\exists \varphi :: \varphi \ \underline{\mathsf{un}} \ \mathcal{S}) \\
\equiv & \quad \{(15);\ \text{instantiation}\ \ S := T\ \} \\
 & (\exists \varphi :: \varphi \ \underline{\mathsf{un}} \ \mathcal{S} \ \wedge \ [\varphi * T]) \\
\equiv & \quad \{\text{Proposition 16}\} \\
 & (\exists \varphi :: \varphi \ \underline{\mathsf{un}} \ \mathcal{S} \ \wedge \ [\varphi * T] \ \wedge \ \varphi = \varphi' \ \wedge \ \neg(x \prec t)) \\
\equiv & \quad \{\ \wedge\ \text{over}\ \exists\ \} \\
 & \neg(x \prec t) \ \wedge \ (\exists \varphi :: \varphi \ \underline{\mathsf{un}} \ \mathcal{S} \ \wedge \ [\varphi * T] \ \wedge \ \varphi = \varphi') \\
\equiv & \quad \{(15);\ \text{instantiation}\ \ S := T\ \} \\
 & \neg(x \prec t) \ \wedge \ (\exists \varphi :: \varphi \ \underline{\mathsf{un}} \ \mathcal{S} \ \wedge \ \varphi = \varphi') \quad .
\end{array}
$$

Under assumption of $\neg(x \prec t)$, the existential quantification in the last line of the derivation may be estimated as follows:

$$
\begin{array}{ll}
 & (\exists \varphi :: \varphi \ \underline{\mathsf{un}}\ \mathcal{S} \ \wedge \ \varphi = \varphi') \\
\Rightarrow & \quad \{\text{Leibniz}\} \\
 & (\exists \varphi :: \varphi' \ \underline{\mathsf{un}}\ \mathcal{S}) \\
\equiv & \quad \{\text{Corollary 14, using } \neg(x \prec t)\} \\
 & (\exists \varphi :: \varphi' \ \underline{\mathsf{un}}\ \mathcal{S} \ \wedge \ \varphi' = \varphi' \circ (x \leftarrow t)) \\
\Rightarrow & \quad \{\text{dummy transformation } \varphi := \varphi'\} \\
 & (\exists \varphi :: \varphi \ \underline{\mathsf{un}}\ \mathcal{S} \ \wedge \ \varphi = \varphi') \quad .
\end{array}
$$

This shows, by mutual implication, that

$$(\exists \varphi :: \varphi \ \underline{\mathsf{un}}\ \mathcal{S} \ \wedge \ \varphi = \varphi') \ \equiv \ (\exists \varphi :: \varphi' \ \underline{\mathsf{un}}\ \mathcal{S}) \quad . \tag{17}$$

So we get

$$
\begin{array}{ll}
 & (\exists \varphi :: \varphi \ \underline{\mathsf{un}}\ \mathcal{S} \ \wedge \ \varphi = \varphi') \\
\equiv & \quad \{(17)\} \\
 & (\exists \varphi :: \varphi' \ \underline{\mathsf{un}}\ \mathcal{S}) \\
\equiv & \quad \{(15)\} \\
 & (\exists \varphi :: (\forall S : S \in \mathcal{S} : [\varphi' * S])) \\
\equiv & \quad \{(11);\ \text{dummy transformation } S := (x \leftarrow t) * S\} \\
 & (\exists \varphi :: (\forall S : S \in (x \leftarrow t) ** \mathcal{S} : [\varphi * S])) \\
\equiv & \quad \{(15);\ (16)\} \\
 & \mathit{Unif}.((x \leftarrow t) ** \mathcal{S}) \quad .
\end{array}
$$

Together with the result of the first derivation, this proves the lemma. □

And here is the translation of Lemma 17 in terms of *Unif*:

Lemma 19 *For $T \in \mathcal{S}$ such that $T \subseteq \mathit{Terms} \setminus \mathit{Var}$, we have*

$$\mathit{Unif}.\mathcal{S} \ \equiv \ [\mathit{Func}.T] \ \wedge \ \mathit{Unif}.(\mathcal{S} \setminus \{T\} \cup \mathit{Comp}.T) \quad .$$

Proof.

$$
\begin{array}{ll}
 & \mathit{Unif}.\mathcal{S} \\
\equiv & \quad \{(16)\} \\
 & (\exists \varphi :: \varphi \ \underline{\mathsf{un}}\ \mathcal{S}) \\
\equiv & \quad \{(15);\ \text{split off } S = T\} \\
 & (\exists \varphi :: \varphi \ \underline{\mathsf{un}}\ \mathcal{S} \setminus \{T\} \ \wedge \ [\varphi * T]) \\
\equiv & \quad \{\text{Lemma 17}\} \\
 & (\exists \varphi :: \varphi \ \underline{\mathsf{un}}\ \mathcal{S} \setminus \{T\} \ \wedge \ [\mathit{Func}.T] \ \wedge \ \varphi \ \underline{\mathsf{un}}\ \mathit{Comp}.T) \\
\equiv & \quad \{\wedge \text{ over } \exists\text{; joining the ranges in (15)}\} \\
 & [\mathit{Func}.T] \ \wedge \ (\exists \varphi :: \varphi \ \underline{\mathsf{un}}\ \mathcal{S} \setminus \{T\} \cup \mathit{Comp}.T) \\
\equiv & \quad \{(16)\} \\
 & [\mathit{Func}.T] \ \wedge \ \mathit{Unif}.(\mathcal{S} \setminus \{T\} \cup \mathit{Comp}.T) \quad .
\end{array}
$$

□

Now an application of the 'tail invariance' program scheme [2] immediately leads to the following program:

$$
\begin{array}{l}
|[\ \textbf{var}\ \mathcal{S} :\ \mathcal{P}.(\mathcal{P}.\mathit{Terms})\ ; \\
\quad b, \mathcal{S} := \mathit{true},\ \{R\}
\end{array}
$$

$$
\begin{array}{l}
\quad \{\text{invariant } P\colon\ \mathit{Unif}.\{R\} \ \equiv\ b \ \wedge\ \mathit{Unif}.\mathcal{S}\ \} \\
;\ \ \textbf{do}\ b \ \wedge\ (\exists T : T \in \mathcal{S} : \neg[T]) \ \rightarrow \\
\qquad\quad T\colon\ T \in \mathcal{S} \ \wedge\ \neg[T] \\
\qquad ;\ \textbf{if}\ T \cap \mathit{Var} = \emptyset \ \rightarrow \quad \textbf{if}\ [\mathit{Func}.T] \quad \rightarrow\ \mathcal{S} := \mathcal{S} \setminus \{T\} \cup \mathit{Comp}.T \\
\qquad\qquad\qquad\qquad\qquad\quad [\!]\ \neg[\mathit{Func}.T] \ \rightarrow\ b := \mathit{false} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{fi} \\
\qquad\qquad\qquad\qquad\qquad\quad \{\, P\,, \text{by lemma 19}\} \\
\qquad\quad [\!]\ \ T \cap \mathit{Var} \neq \emptyset \ \rightarrow \quad x\colon\ x \in T \cap \mathit{Var} \\
\qquad\qquad\qquad\qquad\qquad ;\ t\colon\ t \in T \setminus \{x\} \\
\qquad\qquad\qquad\qquad\qquad ;\ \textbf{if}\ \neg(x \prec t) \ \rightarrow\ \mathcal{S} := (x \leftarrow t) ** \mathcal{S} \\
\qquad\qquad\qquad\qquad\qquad\quad [\!]\ x \prec t \qquad \rightarrow\ b := \mathit{false} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{fi} \\
\qquad\qquad\qquad\qquad\qquad\quad \{\, P\,, \text{by lemma 18}\} \\
\qquad\quad \textbf{fi} \\
\quad\ \textbf{od} \\
\quad\ \{\, P \ \wedge\ (\neg b \ \vee\ (\forall T : T \in \mathcal{S} : [T]))\ \} \\
\quad\ \{\, P \ \wedge\ \mathit{Unif}.\mathcal{S}\ \} \\
]|| \\
\{\, b \ \equiv\ \mathit{Unif}.\{R\}\ \}
\end{array}
$$

provided the repetition terminates. That will be proved in the next section.

6 Termination

Informally, an application of the second alternative in the repetition will reduce the number of different variables occurring in $\mathcal{S}$ (variable x is eliminated). In order to make this idea precise, it will be advantageous to use the abbreviation

$$t \preceq s \ \equiv\ t = s \ \vee\ t \prec s \ ,$$

which we shall do implicitly.

Lemma 20 *For variables x, y and terms t, s such that $\neg(x \preceq t)$,*

$$y \preceq (x \leftarrow t).s \ \Rightarrow\ y \neq x \ \wedge\ (y \preceq s \ \vee\ y \preceq t) \ .$$

Proof. By structural induction on s. The base, $s \in \mathit{Var}$, is split into the cases $s = x$ and $s \neq x$. First,

$$
\begin{array}{ll}
 & y \preceq (x \leftarrow t).x \\
\equiv & \quad \{(8)\} \\
 & y \preceq t \\
\equiv & \quad \{\, \neg(x \preceq t)\,\} \\
 & y \neq x \ \wedge\ y \preceq t \\
\Rightarrow & \quad \{\} \\
 & y \neq x \ \wedge\ (y \preceq x \ \vee\ y \preceq t) \ .
\end{array}
$$

Next, for z any variable distinct from x,

$$
\begin{array}{ll}
 & y \preceq (x \leftarrow t).z \\
\equiv & \quad \{(8)\} \\
 & y \preceq z
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{\ \neg(x \prec z)\ \text{because of (5) and}\ z \neq x\ \} \\
 & y \neq x\ \wedge\ y \preceq z \\
\Rightarrow & \{\} \\
 & y \neq x\ \wedge\ (y \preceq z\ \vee\ y \preceq t)\ \ .
\end{array}
$$

For the step, we take $s = (f, v)$ and observe

$$
\begin{array}{ll}
 & y \preceq (x \leftarrow t).(f, v) \\
\equiv & \{(7)\} \\
 & y \preceq (f, (x \leftarrow t) \circ v) \\
\equiv & \{(2)\} \\
 & y \prec (f, (x \leftarrow t) \circ v) \\
\equiv & \{(6)\} \\
 & (\exists i :: y \preceq (x \leftarrow t).(v.i)) \\
\Rightarrow & \{\text{induction hypothesis}\} \\
 & (\exists i :: y \neq x\ \wedge\ (y \preceq v.i\ \vee\ y \preceq t)) \\
\equiv & \{\text{distribution}\} \\
 & y \neq x\ \wedge\ ((\exists i :: y \preceq v.i)\ \vee\ y \preceq t) \\
\Rightarrow & \{(6)\} \\
 & y \neq x\ \wedge\ (y \preceq (f, v)\ \vee\ y \preceq t)\ \ .
\end{array}
$$

□

For r a term and $\mathcal{S} \in \mathcal{P}.(\mathcal{P}.Terms)$, we define

$$
r\ \underline{\mathsf{in}}\ \mathcal{S}\ \ \equiv\ \ (\exists s, S : s \in S \in \mathcal{S} : r \preceq s)\ \ . \tag{18}
$$

We now investigate the behaviour of $y\ \underline{\mathsf{in}}\ \mathcal{S}$ under the assignments occurring in our algorithm.

Lemma 21 *Let x, y be variables and t a term satisfying $t \in T \in \mathcal{S}$ and $\neg(x \preceq t)$. Then*

$$
y\ \underline{\mathsf{in}}\ (x \leftarrow t) **\mathcal{S}\ \ \Rightarrow\ \ y \neq x\ \wedge\ y\ \underline{\mathsf{in}}\ \mathcal{S}\ \ .
$$

Proof.

$$
\begin{array}{ll}
 & y\ \underline{\mathsf{in}}\ (x \leftarrow t) **\mathcal{S} \\
\equiv & \{(18)\} \\
 & (\exists s, S : s \in S \in (x \leftarrow t) **\mathcal{S} : y \preceq s) \\
\equiv & \{\text{dummy transformation}\ S := (x \leftarrow t) * S\ \} \\
 & (\exists s, S : s \in (x \leftarrow t) * S\ \wedge\ S \in \mathcal{S} : y \preceq s) \\
\equiv & \{\text{dummy transformation}\ s := (x \leftarrow t).s\ \} \\
 & (\exists s, S : s \in S \in \mathcal{S} : y \preceq (x \leftarrow t).s) \\
\Rightarrow & \{\text{Lemma 20}\} \\
 & (\exists s, S : s \in S \in \mathcal{S} : y \neq x\ \wedge\ (y \preceq s\ \vee\ y \preceq t)) \\
\equiv & \{\text{distribution}\} \\
 & y \neq x\ \wedge\ ((\exists s, S : s \in S \in \mathcal{S} : y \preceq s)\ \vee\ y \preceq t) \\
\equiv & \{\text{instantiation}\ s, S := t, T\text{, using}\ t \in T \in \mathcal{S}\ \} \\
 & y \neq x\ \wedge\ (\exists s, S : s \in S \in \mathcal{S} : y \preceq s) \\
\equiv & \{(18)\} \\
 & y \neq x\ \wedge\ y\ \underline{\mathsf{in}}\ \mathcal{S}\ \ .
\end{array}
$$

□

If we now define

$$
F0.\mathcal{S}\ =\ (\#y :: y\ \underline{\mathsf{in}}\ \mathcal{S})\ \ , \tag{19}
$$

we may draw the following conclusion:

Corollary 22 *For x a variable and t a term satisfying $\{x,t\} \subseteq T \in \mathcal{S}$ and $\neg(x \preceq t)$,*

$$F0.((x \leftarrow t) ** \mathcal{S}) < F0.\mathcal{S} \quad .$$

□

Hence, as far as the second alternative in the repetition body is concerned, $F0.\mathcal{S}$ is a suitable variant function. We have still to investigate its behaviour in the first alternative.

Lemma 23 *For $T \in \mathcal{S}$ with $T \cap Var = \emptyset \ \wedge\ [Func.T]$ and any r,*

$$r \ \underline{\text{in}}\ Comp.T \ \Rightarrow\ r \ \underline{\text{in}}\ \mathcal{S} \quad .$$

Proof.

$$\begin{array}{ll} & r \ \underline{\text{in}}\ Comp.T \\ \equiv & \quad \{(18)\} \\ & (\exists s, S : s \in S \in Comp.T : r \preceq s) \\ \equiv & \quad \{(14)\} \\ & (\exists f, v, i : (f,v) \in T : r \preceq v.i) \\ \Rightarrow & \quad \{(6)\} \\ & (\exists f, v : (f,v) \in T : r \preceq (f,v)) \\ \Rightarrow & \quad \{\, T \in \mathcal{S} \,\} \\ & (\exists s, S : s \in S \in \mathcal{S} : r \preceq s) \\ \equiv & \quad \{(18)\} \\ & r \ \underline{\text{in}}\ \mathcal{S} \quad . \end{array}$$

□

Corollary 24 *For $T \in \mathcal{S}$ with $T \cap Var = \emptyset \ \wedge\ [Func.T]$,*

$$F0.(\mathcal{S} \setminus \{T\} \cup Comp.T) \leq F0.\mathcal{S} \quad .$$

□

We cannot prove that $F0.\mathcal{S}$ is decreased by the first alternative, but at least Corollary 24 shows that it is not increased. So $F0.\mathcal{S}$ is a suitable first component of a tuple to be used as variant function. To find the other components, we must look at quantities decreased by the first alternative.

Informally, such an application reduces the number of occurrences of function symbols (an occurrence of f is eliminated). In order to make this idea precise, we define a function that counts the number of occurrences of function symbols in a term.

Definition 25 *Function Nof is defined on Terms by*

$$Nof.x = 0 \quad , \tag{20}$$

$$Nof.(f,v) = 1 + (\Sigma\, i :: Nof.(v.i)) \quad , \tag{21}$$

where $x \in Var$, and $f \in Fu_n$, $v \in Terms^n$ for some natural n. Dummy i ranges over $[0\,..\,n)$. □

For $F1$ defined by

$$F1.\mathcal{S} \ =\ (\Sigma\, s, S : s \in S \in \mathcal{S} : Nof.s) \tag{22}$$

we have the following result.

Lemma 26 *For T with $T \neq \emptyset \ \wedge\ T \cap Var = \emptyset \ \wedge\ [Func.T]$ we have*

$$F1.\{T\} > F1.(Comp.T) \quad .$$

Proof.

$$
\begin{array}{ll}
 & F1.\{T\} \\
= & \quad \{(22)\} \\
 & (\Sigma\, s : s \in T : Nof.s) \\
= & \quad \{\, T \cap Var = \emptyset \,\} \\
 & (\Sigma\, f, v : (f, v) \in T : Nof.(f, v)) \\
= & \quad \{(21)\} \\
 & (\Sigma\, f, v : (f, v) \in T : 1 + (\Sigma\, i :: Nof.(v.i))) \\
> & \quad \{\, T \neq \emptyset \,\} \\
 & (\Sigma\, f, v : (f, v) \in T : (\Sigma\, i :: Nof.(v.i))) \\
= & \quad \{\, [Func.T] \text{ , hence the range of } i \text{ is the same for all } f, v \,\} \\
 & (\Sigma\, i, f, v : (f, v) \in T : Nof.(v.i)) \\
= & \quad \{(14)\} \\
 & (\Sigma\, s, S : s \in S \in Comp.T : Nof.s) \\
= & \quad \{(22)\} \\
 & F1.(Comp.T) \quad .
\end{array}
$$

□

Corollary 27 *For* $T \in \mathcal{S}$ *with* $T \neq \emptyset \;\wedge\; T \cap Var = \emptyset \;\wedge\; [Func.T]$ *we have*

$$F1.(\mathcal{S} \setminus \{T\} \cup Comp.T) < F1.\mathcal{S} \quad .$$

□

This proves that a suitable variant function for our algorithm is

$$(F0.\mathcal{S},\ F1.\mathcal{S}) \quad ,$$

with the components defined by (19) and (22) respectively.

Acknowledgement Thanks are due to Rob Hoogerwoord for detailed comments on a previous version of this note that improved the presentation and simplified both Proposition 13 and the variant function.

References

1. Atze Dijkstra, Arie Middelkoop, and S. Doaitse Swierstra, *Efficient functional unification and substitution*. Technical Report UU-CS-2008-027. Institute of Information and Computing Sciences, Utrecht University, 2008.
2. A. Kaldewaij, *Programming: the derivation of algorithms*. Prentice-Hall, New York, 1990.
3. J.W. Lloyd, *Foundations of logic programming*, 2nd ed. Springer-Verlag, Berlin, 1987.
4. A. Martelli and U. Montanari, 'An efficient unification algorithm'. *ACM Trans. Prog. Lang. Syst.* **4** (1982), 258–282.
5. M.S. Paterson and M.N. Wegman, 'Linear unification'. *J. Comp. Syst. Sci.* **16** (1978), 158–167.
6. J.A. Robinson, 'A machine-oriented logic based on the resolution principle'. *J. ACM* **12** (1965), 23–41.

The Mismatch between Computing Science and Corporate / Government IT

Nico Verwer

nverwer@gmail.com

Abstract. For many years, CIO's, human resource managers and other people from the world of corporate and government IT have complained about the disconnect between what students learn at universities, and what is required to function well in a business environment. This perceived mismatch is usually blamed on the curriculum that is offered by computing science faculties being 'too theoretical' in nature. This paper, based on personal experience of the past 20 years, looks at the reasons why the mismatch still exists. An argument is made that if more people who enter 'real-world' IT would have a stronger theoretical background, we might see far fewer failed projects.

1 IT in the "real world"

After finishing my Ph.D. research under the invaluable supervision of Doaitse, in 1993, I decided to pursue a career outside the academia. Since then I have worked on many projects, which had as one of their aims the production of useful software. Back in the early 1990's, most people I worked for or with had an education in computing science or physics, or at least some engineering background. This was true for software developers (who were still called "programmers"), designers, architects (a function title that was still associated with constructing buildings), and even most project managers.

In 20 years, the number of people involved in software-related projects without any background in science or engineering has steadily increased. They carry function titles like project leader, program manager, business architect, enterprise architect, project architect, solutions architect, functional designer, requirements analyst, etcetera. In one project for a large Dutch government organization, that I was part of, there were some 30 'designers' of various types, several architects, project managers, and 15 to 20 software engineers. My contribution as a 'software designer' was, that I actually spoke with the software engineers (who had to build the product based on numerous vague, ambiguous and conflicting designs) and found out that what they were doing had little to do with what was discussed in the meetings of the designers. This project never met any of its planned objectives or estimates.

Of course, big projects are hard to estimate and plan. But many smaller projects do not do any better. Typically, project managers use "reverse planning":

1. Put every feature and task that the customer organization could imagine in a row in a big Excel spreadsheet.
2. Colour the background of some cells in each row (the number of cells being the relative time spent), so that the order suggests dependencies between features and tasks.
3. Calculate the number of months the project can run within the budget, which was estimated way too low from the beginning, in order to acquire the project.
4. Distribute the months across the header columns, so that the last feature is finished within the allotted time.

Most architects see no problem with a planning like this, especially when the waterfall methodology is used and their tasks are planned at the beginning of the spreadsheet. When software engineers start whining about the planning, they are told that they'd better spend their time writing code.

Since I became a software engineer 20 years ago, I have moved on, and my title is now "software and data architect, designer, engineer". Recently, a manager of the company that employs me, told me that my "profile does not fit in the company portfolio". This did not mean that the company has a too small wallet. Two years ago, management has decided to focus on "requirements and architecture", and building software is not an activity they want to be associated with. The main reason is that customers are willing to pay much more for architects, analysts and managers, so profit margins are higher. Besides, constructing software is associated with failure; When a software engineers makes a mistake, it leads to a badly or non-functioning product, whereas the architect's Powerpoint designs never crash.

The undervaluation of constructing software has led to a situation where most software engineers have little or no theoretical background, and can only blindly translate vague specifications into lots of programming code. This situation has become worse by the introduction of "enterprise frameworks" such as J2EE, which require programmers to write lots of mind-numbing boilerplate code.

2 Should the gap be bridged?

As far back as the 1980's, companies (and some government agencies) were complaining that CS departments did not deliver the kind of people they needed. Apparently, graduates have too much theoretical knowledge, and are not able to communicate with people within the company. Indeed, mathematical skills and the ability to analyse a problem before writing code do not match well with the functions in software projects, as described above.

Companies and government agencies have tried to bridge the gap (sometimes called 'alignment') between IT and 'the business' by creating new functions, such as the CIO. This has led to a new administrative layer of people with good 'soft skills'. They get along very well with other managers, and are good career-makers, but I have never met one with an understanding of technical issues. Usually they 'standardize' IT, by stating, for example, that "we will only use a Windows architecture".[1]

Universities have responded by creating new studies like "business informatics". It is useful to have project leaders and business architects who have an education other than MBA, and graduates from these studies can probably communicate better with both management and 'technical' people.

The question remains what problem is solved, by better communication between, or alignment of business and IT? Surely, it is beneficial if someone can elucidate what functional requirements IT projects should satisfy. But in most projects, even the future users of software do not know in advance what they need, let alone their managers.

In one project, a colleague and I worked without a project manager, and without an advance planning. Every week, a group of future users would have a look at what we had made, and comment on it.[2] The resulting product was very different from the initial assignment we had, but users were enthusiastic and used the software before it was officially in the production phase. I believe that part of the success of this project was that we had a moderate, not overly ambitious scope, and that my colleague and I fulfilled all functions (he is also a "software and data architect, designer, engineer"). Whenever we designed something

[1] Again, this has nothing to do with the construction of buildings.

[2] This is akin to the Scrum methodology, with an extremely short iteration period.

in the role of 'architect', we would care about what we would do with that the next moment in the role of 'engineer'.

Another success factor in this project was that we analysed problems, and made an effort to understand what was going on. When the users had left our room, we would speak about things like transitive closures of certain relations, computational complexity, and other stuff that we were taught at university many years ago.

In my current project, there are more people without any 'theoretical' background. Part of the project is about how versions of laws develop over time. What particular version of a law is relevant does not only depend on the required validity date, but also on the observation date (because changes can be made retroactively). I have analysed this, and shown that in the context of laws, time is fundamentally two-dimensional. Yet nobody in the project accepts this, because "it can't be that difficult." Neither a moderate amount of algebra, nor actual examples will convince them otherwise.

In this project, the complexity of two-dimensional time should probably be hidden from most users of the software that is built. But it is essential that the people working on the technical side of the business-IT gap understand this complexity, and reflect it in their data models and software. The refusal to do this has led to the loss of information about the history of some laws, and denying this because it is "theoretical" will not get back that information.

If the goal of an IT project is to deliver software that enables users to execute their tasks, *and* that functions correctly, more, not less people with 'theoretical' knowledge and analytical skills are needed. The gap between business and IT should not be bridged by making technical people become more like managers, focussing on their 'soft skills'.

Still, too many software projects fail. This problem will not be solved by having more *xyz*-architects, spreadsheet managers and Powerpoint designers. Good communication between prospective users and engineers is vital, but both need to have a thorough understanding of their respective domains. This is why I hope that the work that Doaitse has pursued during his tenure at the Department of Information and Computing Sciences will be continued, and more young people with a 'too theoretical' education will enter 'real-world' IT.

Anecdotarium

Alexey Rodriguez Yakushev

alexey@vectorfabrics.com
Vector Fabrics B.V.

Abstract. The author recounts a few anecdotes that shed some light on Doaitse's relationship with the Software Technology PhD students, in particular with the author.

1 The Doaitse Cup

During the last year of my PhD, Doaitse bought an espresso coffee maker machine. It was one of those machines that, at a button press, would grind coffee grains and make a fresh and steaming cup of espresso coffee. Even now, writing these memories down, I yearn for such a cup of coffee!

Of course, the computer science department already had a communal scheme for brewing coffee. However, I and other colleagues I spoke to considered this coffee not quite tasty (to put it mildly).

Doaitse would enjoy drinking his cup of espresso. But you have to realize that he bought this coffee machine (at his expense) to share it with the Software Technology staff, in particular (I think) to share it with the PhD students that used to work there.

This is a typical generosity gesture from Doaitse. He usually helps newly arrived PhD students with issues such as finding accommodations, solve transportion problems by lending or giving away bikes, providing student jobs, facilitating access to laptops, and all sorts of other things. In addition, every now and again he would also give treats that had no utility other than making the PhD students happy. Buying the coffee machine was such an example, but I also remember borrowing his apartment in the beautiful island of Ameland to spend a great weekend there.

At the time that this story happened, we, the PhD students, were quite obsessed with foosball. Foosball is a competitive game for four persons that simulates a game of football. To play it, you need a table of foosball, which is a table with a small football field painted on it and two teams of eleven players which can be controlled by handles at the side of the table.

The foosball competitions after lunch had become quite a tradition. The coffee machine made it even more memorable. We would each get a cup of espresso after lunch and then walk to the foosball table to initiate the competition. We were quite grateful about the coffee machine and the working environment at the department. So we thought it would be only natural to pay credit to our sponsor (in addition to all the academic work we did of course!) by naming our games *The Doaitse Cup*, which would be the short version of *The Doaitse Foosball Cup*.

2 Read this book over the weekend

As I said before, Doaitse was always willing to help out students with their problems. I was impressed by his patience since I am sure that us, the foreign PhD students, must have been

a constant source of problems (think of immigration paperwork, accommodations, cultural differences and so on).

All this for the students did not mean that Doaitse was not a demanding supervisor. On the contrary, he had very high expectations of the performance of all the group's students. He expected them to work very hard and to produce good and beautiful results.

Sometimes he would reveal his expectations in subtle ways. For example, I remember that right after finishing my master thesis, he suggested I submit a paper to a conference that had a deadline only a week away. Back then I was still slow at writing papers but I wasn't aware of it, so I happily accepted the challenge. I worked very hard that week, sometimes continuing late in the night up to the early hours of the next day. On such days Doaitse would come to my office to monitor my progress. He would be satisfied by my tired looks and he would observe that if I looked tired I must have been working very hard. Seeing that students were making a good effort made Doaitse happy. I think I must have made him happy a few other times during my PhD, since my most productive hours were always in the evenings. I must confess though that a handful other occasions my tired looks might have been a false alarm since they were due to a late dinner with friends or a party.

There is another time that clearly stands out in my memory about how Doaitse subtly proposed a challenge. This was back at the time when I arrived to the Netherlands. Doaitse drove from Utrecht to Eindhoven (where I was doing my internship at the time) and drove me a bit around the city and gave me a bike so that I could more easily move around. At this occasion Doaitse also brought the book on Principles of Program Analysis by Nielson, Nielson and Hankin. He suggested that I could read this book over the weekend. I happily accepted the challenge. Later in the evening I realized how difficult it would be to read the whole thing in the weekend (I did not finish it). Nevertheless, I did manage to learn a new mathematical framework and applications of it during that weekend.

3 Lasting influence

I am writing this little collection of anecdotes using a Mac computer. As you know, Doaitse uses the Mac platform for all his work. The fact that I also use it is just one of the influences I have had from Doaitse over the years. He has strong opinions and clear convictions on research interests, ways of working, and life in general.

Probably the first influence I had from Doaitse was when I took the course on advanced functional programming during my bachelor studies in Bolivia. We studied multiple designs for parser combinators, all of which were, of course, based on Doaitse's work. The influence was indirect but it was strong and lasting, it was the first time I was exposed to the style of designing APIs using a combinator style.

Some years later, when I was working on my master thesis, I got interested in the modular construction of compilers. Of course, it was Doaitse who got me interested in the topic. So, for my master thesis, I worked on a technique to modularly compose attribute grammars to achieve reuse of compiler analyses.

To conclude, I can say that his influence is still lasting to this day. It is not unfrequent when I run into a software design or algorithmic problem and I see that the problem can be thought in terms of an attribute grammar formalism. Doaitse's guidance, support and advice has been very important in my career. I hope Doaitse enjoys the few anecdotes that I collected here as much as I enjoyed them when they took place.

Mechanized Metatheory for a λ-Calculus with Trust Types

Rodrigo Ribeiro, Carlos Camarão, and Lucília Figueiredo

[1] Instituto de Ciências Exatas, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais. {rribeiro,camarao}@dcc.ufmg.br.
[2] Instituto de Ciências Exatas e Biológicas, Departamento de Computação, Universidade Federal de Ouro Preto. lucilia@iceb.ufop.br

Abstract. This work is in honour to a most trustful friend, Professor Doaitse Swierstra, whose brilliant ideas, enthusiasm, sense of humour and kind heart will be for us always an inspiration and a model.

As computer programs become increasingly complex, techniques for ensuring trustworthiness of information manipulated by them become critical. In this work, we use the Coq proof assistant to formalize a λ-calculus with trust types, originally formulated by Ørbæk and Palsberg. We give formal proofs of type soundness, erasure and simulation theorems and also prove decidability of the typing problem. As a result of our formalization a certified type checker is derived.

1 Introduction

Ensuring security of information manipulated by computer systems is a long-standing and increasingly important problem. There is little assurance that current computer systems keep data integrity and traditional (theoretical and practical) approaches to express and enforce security properties are, in general, unsatisfactory [1, 2].

One of such traditional approaches to protect data confidentiality is access control: privileges are required to access files or objects containing confidential data. Information release is restricted according to some policy. Access control checks can restrict release but not propagation of information. Once information is released, a program can transmit it in some form and, since it is not feasible to suppose that all programs in a system are trustworthy, we cannot ensure that confidentiality is maintained. In order to guarantee that information is used only in accordance with relevant policies, it is necessary to analyze how information flows within the program. Since modern computing systems are complex artefacts, a form of automating such analysis is required [1].

A promising approach has been recently developed, which consists on the use of type systems in order to control information flow in software [1]. In programming languages with *security types*, variables and expressions types have annotations that indicate policies to be ensured by the compiler on uses of such data. This approach has the following benefits: 1) since these policies are checked at compile-time, there is no run-time overhead; 2) once security policies are expressed by a type system, standard techniques for guaranteeing type system soundness can be used to certify that security policies are enforced in an end-to-end way in the whole program.

However, proofs of programming language formalisms (e.g. type systems and semantics) are usually long and error prone. In order to give more reliability to these proofs, programming language researchers have been developing, in recent years, a large number of works devoted to machine assisted proofs [3, 4, 5, 6].

In this work, we provide a formalization of a variant of λ-calculus with trust types, as proposed by Ørbæk and Palsberg [7], using the Coq proof assistant [8]. Specifically, our contribution is to provide a machine checked proof of:

1. type soundness, using a standard small-step call-by-value semantics;
2. erasure and simulation theorems [7, Sections 3.3 and 3.4];
3. decidability of type checking. From this proof we extract a certified type checker for the language.

The developed formalization is axiom free and has approximately 1400 lines of code. This makes it impossible to present here all details of the work. We only sketch the main proofs and some function definitions are omited for brevity, when they are trivial. The Coq source code of this work is available at trust-calculus github repository.

The rest of this paper is organized as follows. Section 2 briefly reviews the syntax and defines a small-step semantics for the λ-calculus with trust types. Section 3 presents the non-syntax directed type system for the λ-calculus with trust, as proposed in [7], and proves its type soundness property. We also define a syntax directed version of this original type system and prove soundness and completeness between these two versions. We also prove that the typing problem for this calculus is decidable. Section 5 presents related work and Section 6 concludes.

2 λ-Calculus with Trust Types

This section reviews some motivations for the use of trust types and gives definitions of the syntax and semantics of the trust λ-calculus, which differ from the original definitions in [7] as follows: 1) we use a small-step call-by-value semantics, and 2) without loss of generality, we consider only one base type: `bool`. Extensions to include other type constructors are straightforward.

2.1 Motivations

Data manipulated by computer programs can be classified as *trusted* or *untrusted*. Trusted data come from trusted sources, like company databases, program constants, cryptographically verified network data etc. All other data are considered untrusted [7].

Trust analysis is especially important in web applications, where user input data can be used to exploit security vulnerabilities, using attacks such as cross-site scripting (XSS). XSS attacks can occur when a user is able to "dump" HTML text in a dynamicaly generated page [9]. Through this vulnerability, it is possible to inject JavaScript code to steal cookies, in order to acquire session privileges. Such a threat occurs due to a lack of verification on input data, since, ideally, HTML code cannot be considered as valid input.

In order to avoid such invalid inputs, one can insert checks that ensure data trustworthiness. But, how can we guarantee that all paths, in which probably untrusted information flows, pass all required checks? The solution proposed by Ørbæk and Palsberg [7] is to use a type system to track the flow of untrusted data in a program.

The language considered is a λ-calculus with additional constructs to check if some piece of data can be trusted and mark data as trusted or untrusted. If e is some program expression, then *trust* e indicates that the result of e can be trusted. Dually, *distrust* e indicates that the result of e cannot be trusted and *check* e indicates that e must be trustworthy. Well-typed programs do not have any sub-expression *check* e where e has an untrusted type.

2.2 Syntax of Types and Terms

The type syntax is given in Figure 1, in which we also provide the meta-variable usage. It is exactly the type syntax of the simply typed λ-calculus with boolean constants, except that each type t has a trust annotation to specify if t-values can be trusted or not. The translation of the type syntax to a Coq inductive type is straightforward, and is also presented in Figure 1.

$u ::= \texttt{tr} \mid \texttt{dis}$
$\tau ::= t^u$
$t ::= \texttt{bool} \mid \tau \to \tau$

```
Inductive trustty : Type :=
   | tr : trustty
   | dis : trustty.

Inductive ty : Type :=
   | ty_bool : trustty -> ty
   | ty_arrow : ty -> ty -> trustty -> ty.
```

Fig. 1. Syntax of trust types

The syntax of terms consists of boolean constants, variables, abstractions and applications, and the three additional constructs to deal with trust types, explained previously. Figure 2 defines the syntax of terms and the corresponding Coq data type.

$e ::= x$
$\mid \lambda x : \tau . e$
$\mid e\, e$
$\mid \texttt{true}$
$\mid \texttt{false}$
$\mid \texttt{trust}\ e$
$\mid \texttt{distrust}\ e$
$\mid \texttt{check}\ e$

```
Inductive term : Type :=
   | tm_var : id -> term
   | tm_lam : id -> ty -> term -> term
   | tm_app : term -> term -> term
   | tm_true : term
   | tm_false : term
   | tm_trust : term -> term
   | tm_distrust : term -> term
   | tm_check : term -> term.
```

Fig. 2. Syntax of terms

The syntax of types and terms used in our formalization is identical to [7], except that we require type annotations in every λ-abstraction. We restrict ourselves to type annotated

λ-terms, since our main interest is the development of a correct type checker for this language. Allowing non-annotated λ-abstractions characterizes a type inference problem that would require a formalization of a unification algorithm. The formalization of a unification algorithm has been studied elsewhere [10, 11]. We let a formalization of the type inference problem for this trust-calculus for future work.

The `id` type, used in the definition of `term`, represents a generic identifier with a decidable function for testing equality and its simple definition is omitted, to avoid unnecessary distraction.

2.3 Small-Step Operational Semantics

In order to prove type soundness, we follow the standard approach of using a small-step operational semantics for proving progress and preservation theorems [12]. This differs from the approach adopted in [7], where the semantics of the trust λ-calculus is formalized using a reduction semantics, with no predefined order of evaluation, and the Church-Rosser property and a Subject Reduction Theorem are proved [13, 14].

Let us firstly define the notion of *value*, i.e. a term that cannot be further reduced according to the intended semantics. We distinguish two kinds of values: primitive values and untrusted values. Primitive values (represented by meta-variable v) are boolean constants and λ-abstractions. An untrusted value (represented by meta-variable u) is a term of the form (*distrust* v), where v is a primitive value. Untrusted values arise as normal forms of terms that do not have any *check* construct.

The definition of values is given in Figure 3. Corresponding Coq definitions for values are straightforward predicate definitions over `term`.

$v ::= \mathit{true}$
  $\mid \mathit{false}$
  $\mid \lambda x : \tau.e$
$u ::= \mathit{distrust}\ v$

```
Inductive prim_value : term -> Prop :=
   | v_true : prim_value tm_true
   | v_false : prim_value tm_false.
   | v_abs : forall x T e,
                prim_value (tm_abs x T e).

Inductive untrusted_value : term -> Prop :=
   | u_dist : forall v, prim_value v ->
            untrusted_value (tm_distrust v)
```

Fig. 3. Definition of Values

The small-step semantics of the trust λ-calculus is an extension of the standard call-by-value semantics for the simply typed λ-calculus. The required extensions deal with trust specific constructs (terms `trust`, `distrust` and `check`). As usual, semantics for λ-calculi rely on substitution. For any e_1, e_2 and x, we define $[x \mapsto e_1]\, e_2$ to be the result of substituting every *free* occurrence of variable x in e_2[3], that follows the standard definition of capture free substitution [13, 14].

[3] The notion of free and bound variables is well-known. See e.g. [14], section 1B.

The Coq function presented in Figure 4 encodes term substitution. The function `subst` replaces every free occurrence of `x` in `t'` for `t`. It is straightforwardly defined by structural recursion over `t'`. In `tm_var` and `tm_abs` cases we have to check whether `x` is equal to the current variable.

```
Fixpoint subst(x : id)(t t' : term) : term:=
  match t' with
    | tm_var i =
        if beq_id x i then t else t'
    | tm_app l r =>
        tm_app (subst x t l) (subst x t r)
    | tm_abs i T t1 => tm_abs i T
        (if beq_id x i then t1
            else (subst x t t1))
    | tm_trust t1 =>
        tm_trust (subst x t t1)
    | tm_distrust t1 =>
        tm_distrust (subst x t t1)
    | tm_check t1 =>
        tm_check (subst x t t1)
    | tm_true        => tm_true
    | tm_false       => tm_false
  end.
```

Fig. 4. Coq function for term substitution.

Figure 5 presents the small-step operational semantics. Most of its rules are standard, but some deserve attention. Rules $\texttt{Trust}_c$, $\texttt{Distrust}_c$, $\texttt{Distrust}_{ca1}$, $\texttt{Distrust}_{ca2}$, $\texttt{Trust}_v$ and $\texttt{Check}_v$ are rules for eliminating redundant uses of trust related constructs. For example, rule $\texttt{Distrust}_c$ specifies that distrusting a value twice is the same as distrusting it once. The other contraction rules have similar meanings.

We denote by $\rightarrow^*$ the reflexive, transitive closure of the small-step semantics. If a term e is not a value (primitive or untrusted), and e cannot be further reduced according to the rules of the small-step semantics, let's say that e is *stuck*. An example of an stuck term is *check*(*distrust true*); since *check* only reduces trusted values, this term does not reduce to any other term and it is not a primitive or untrusted value.

The main purpose of the type system is to rule out all programs that contain stuck expressions such as *check*(*distrust* t), for some term t.

The following lemma states the property that the proposed semantics is deterministic.

Lemma 1 (Determinism of small-step semantics) *For any* e_1, e_2 *and* e_3*, if* $e_1 \rightarrow e_2$ *and* $e_1 \rightarrow e_3$ *then* $e_2 = e_3$*.*

Proof. Induction over the derivation of $e_1 \rightarrow e_2$ and case analysis on the last rule used to conclude $e_1 \rightarrow e_3$.

3 Type System

The type system proposed in [7] is based on the Curry version of the simply-typed λ-calculus. Since our main interest is the development of a certified type-checker and proofs about the

type system, we use a variation of a Church like type system for the simply typed λ-calculus. The type system is defined in Figure 7, as a set of rules for deriving judgements $\Gamma \vdash e : \tau$, meaning that term e has type τ, in typing context Γ (which contains type assumptions for the free variables in e).

Notation $\Gamma, x : \tau$ is the standard notation for extending typing context Γ with a new assumption, after deleting from Γ any type assumption for x; and we let $\Gamma(x) = \tau$ if $x : \tau \in \Gamma$. Typing contexts are represented in Coq by lists of pairs of identifiers and types. Definitions of typing contexts, functions and properties over them (and their corresponding lemmas) are straightforward.

Trust annotations in types are subjected to a subtyping relation $s \preceq s'$, meaning that trust type s is a subtype of s', which is defined as the reflexive relation such that `trust` $\preceq$ `distrust` (the only non-reflexive element of relation $\preceq$ is `trust` $\preceq$ `distrust`).

Using the ordering relation over trust types, we can define a subtyping relation over types. For any types τ and τ', we say that $\tau \leq \tau'$ iff: 1) $\tau = \texttt{bool}^s$, $\tau' = \texttt{bool}^{s'}$ and $s \preceq s'$; 2) $\tau = \tau_1 \to \tau_2$, $\tau' = \tau_1' \to \tau_2'$, $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$. This subtyping relation is defined in Figure 6.

The meaning of the typing rules for boolean constants, variables, subtyping and abstractions is standard. Constants and functions written by the programmer are considered as trusted, following [7]. The rules `T-Trust` and `T-Distrust` "cast" the trust type of an expression to `trust` and `untrust` respectively and rule `T-Check` checks whether an expression has a trusted type. In rule `T-App`, the annotated type of the actual argument is required to match the annotated type of the formal argument. This includes trustworthiness. The trust of the result of an application is the least upper bound of the trust of that function result type and the trust of the function type itself. We let $s \vee s'$ denote the least upper bound between the trust types s and s'.

In order to define the Coq inductive predicate for the typing relation, we need a function to compute the least upper bound of a pair of trust types. The definitions of the least upper bound and trust type update functions are given in Figure 9. The function `lub_trustty` has a straightforward definition and `update_trusty` receives as parameters a type $\tau = t^s$ and a trust annotation s' and updates the trust annotation on type τ to $s \vee s'$.

We can now proceed to prove that the type system enjoys the type soundness property. In order to do this, we need to prove some lemmas about the typing relation, namely: inversion lemmas for the typing relation and canonical forms lemmas [12]. We will not state each one of these "infrastructure" lemmas here, but only sketch the key ones.

Theorem 1 (Progress) *If $\Gamma \vdash e : \tau$, then either e is a value, or it is an untrusted value, or there exists some term e' such that $e \to e'$.*

Proof. Induction over the derivation of $\Gamma \vdash e : \tau$ using canonical form lemmas.

Lemma 2 (Substitution lemma) *If $\Gamma, x : \tau' \vdash e : \tau$ and e' is such that $\Gamma \vdash e' : \tau'$, then $\Gamma \vdash [x \mapsto e']\, e : \tau$.*

Proof. Induction over the structure of e using the corresponding inversion lemma for the typing relation in each case.

Theorem 2 (Preservation) *If $\Gamma \vdash e : \tau$ and $e \to e'$, then $\Gamma \vdash e' : \tau'$, for some τ' such that $\tau' \leq \tau$.*

Proof. Induction over the derivation of $\Gamma \vdash e : \tau$ and case analysis over the last rule used to conclude $e \to e'$, using Lemma 2.

Corollary 1 (Type Soundness) *If $\Gamma \vdash e : \tau$ and $e \rightarrow^* e'$, then e' is not stuck (i.e., e' is not of the form* `check` *e'', where e'' has an untrusted type).*

Proof. Induction over $e \rightarrow^* e'$ using Theorems 1 and 2.

3.1 Syntax Directed Type System

The type system presented in Figure 7 has the drawback of allowing applications of rule `T-Sub` at any place in the type derivation for some expression e. This makes this set of rules not immediately suitable for implementation. In this section, we develop a syntax-directed version of the type system for the trust λ-calculus and prove its soundness and completeness with respect to the original type system.

The syntax directed type system is presented in Figure 8, as a set of rules for deriving judgements of the form $\Gamma \vdash^D e : \tau$. The rules are almost the same as the ones in Figure 7, except for the application rule, that now includes, as a premise, a test of the subtyping relation $\tau' \leq_D \tau$, which represents a function that is true if and only if $\tau' \leq \tau$ holds. Termination, soundness and completeness of the subtyping test function follows the approach in [12] and their proofs are straightforward.

The next theorems state soundness and completeness of the syntax directed type system, and their proofs are in the companion Coq scripts.

Theorem 3 (Soundness) *If $\Gamma \vdash^D e : \tau$, then $\Gamma \vdash e : \tau$.*

Proof. Induction on the derivation of $\Gamma \vdash^D e : \tau$.

Theorem 4 (Completeness) *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash^D e : \tau'$ for some τ' such that $\tau' \leq \tau$.*

Proof. Induction on the derivation of $\Gamma \vdash e : \tau$.

Finally, we prove that the typing problem for the trust λ-calculus is decidable, that is, we prove that, given a typing context Γ and term e, it is decidable whether there exists a type τ such that $\Gamma \vdash^D e : \tau$. Due to the constructive nature of this proof, a certified algorithm for type checking an expression can be extracted from it. This theorem is stated as the following piece of Coq source code.

```
Theorem typecheck_dec :
   forall(e : term) (ctx : context),
     {t | has_type_alg ctx e t} +
      {forall t, ~ has_type_alg ctx e t}.
```

Predicate `has_type_alg` represents the syntax directed type system of Figure 8. Intuitively, this theorem means that either there exists a type t such that `has_type_alg ctx e t` is provable or there is no such type `t`.

4 Erasure and Simulation

As pointed out in [7], the type system for the λ-calculus with trust types is just a restriction of the classic (in our formalization) Church type system for λ-calculus. This notion is formalized by an erasure function that converts terms, types and contexts from the trust calculus to bare λ-calculus.

Intuitively, the erasure function removes trust annotations from types, as well as trust constructs from terms. These functions are given in Figure10.

Following [7], we write these erasure functions using notation $|\phi|$, where ϕ is used as a term, type or context.

Lemma 3 (Lemma 12 of [7]) *For any trust types τ and τ' such that $\tau \leq \tau'$ we have that $|\tau| = |\tau'|$.*

Proof. Induction over the derivation of $\tau \leq \tau'$.

The relationship between the trust calculus and λ-calculus is stated by the next theorem, where the judgement $\Gamma \vdash^C e : t$ denotes the Church style type system for the λ-calculus presented in Figure 11. The proof of this theorem uses some lemmas relating erasure and operations over typing contexts and types. These lemmas are necessary just for "lifting" the erasure functions. Since these lemmas are simple consequences of these function definitions, they are omitted here.

Theorem 5 (Erasure) *If $\Gamma \vdash e : \tau$, then we have that $|\Gamma| \vdash^C |e| : |\tau|$.*

Proof. Induction over $\Gamma \vdash e : \tau$.

For any well typed term, we can erase all `trust`, `distrust` and `check` constructs and evaluate the resulting term using a standard semantics of λ-calculus. In practice, this means that after type-checking a term, we can erase all trust related constructs and evaluate the term without any performance penalties [7]. This fact is expressed by the following theorem.

Theorem 6 (Simulation) *If $\Gamma \vdash e : \tau$ and $|e| \rightarrow^* e'$ then there exists e_1 such that $e \rightarrow^* e_1$ and $|e_1| = e'$.*

Proof. Induction over e.

5 Related Work

Language support. The use of language based techniques for protecting information has as it most prominent example the security mechanism implemented by the Java run-time environment, which defines a set of security policies for applets [15, 16].

Recently, an extension of Haskell was designed to deal with some language features that can be used to bypass the type system, referential transparency and module encapsulation [17]. The approach used by Safe Haskell is to classify modules and packages as safe, trusted and unsafe based on its source code or in compiler pragmas that can be used to declare a possibly unsafe module as trustworthy. The Safe Haskell extension is available in the GHC compiler version 7.2 [18]. The authors used it to implement a web-based version of a Haskell interpreter, but no formal description of the safety inference process was given.

Type systems for security. Volpano et. al. was the first to use type systems to enforce security policies by a compiler [19]. They defined the lattice based analysis proposed by Denning in [20] as a type system for a prototypical imperative language with first order procedures. Their type system relies on polymorphism, thus allowing that commands and expression types depend on the context in which they occur. Another proposal for a type system for ensuring security was described in [21], where a type system for a purely functional language was given. The JFlow type system [22] is used in a language that extends Java with security types. A production compiler for this language is available [23] and was used in the development of a secure voting system [24].

Barthe et. al. [25] describe a security type system for a low level language with jumps and calls and prove that information flow types are preserved by the compilation. A mechanized proof of Barthe's work was given in [26], using the Coq proof assistant.

As pointed out by Ørbæk and Palsberg in [7], security analysis focus on avoiding that classified information leaks out of a system to unprivileged users. The formalized type system ensures that untrustworthy information does not flow *into* the system. So, a trust type system can be seen as the "dual" of security type systems.

Use of Proof Assistants. The use of proof assistants for mechanizing programming language metatheory has been the subject of extensive research in several directions [3, 4, 27, 6, 28, 29, 30]. Successful applications of proof assistants in programming languages are the Compcert verified C compiler [31] and formalizations of Java virtual machines [32].

6 Conclusion

We presented an axiom-free, fully constructive Coq formalization of λ-calculus with trust types. The use of a small-step semantics, instead of a reduction semantics, and a Church-style, instead of a Curry-style, type system are the major differences between the present work and its original formulation. This allowed us to give concise proofs of type soundness, erasure and simulation theorems.

We also presented a syntax directed formulation of the original type system, that is sound and complete with respect to the former. Decidability of type checking is proved using the syntax directed version and a correct type checker can be extracted from this proof.

The complete formalization has near 1,400 lines of Coq code and can be found at trust-calculus github repository.

References

[1] Sabelfeld, A., Myers, A.C.: Language-based information-flow security. Selected Areas in Communications, IEEE Journal on **21**(1) (January 2003) 5–19

[2] Volpano, D., Smith, G.: 48. In: A type-based approach to program security. Volume 1214 of Lecture Notes in Computer Science. Springer-Verlag Berlin / Heidelberg, Berlin/Heidelberg (1997) 607–621

[3] Aydemir, e.a.: Engineering formal metatheory. In Necula, G.C., Wadler, P., eds.: POPL, ACM (2008) 3–15

[4] Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning **43**(3) (2009) 263–288

[5] Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4) (2009) 363–446

[6] Chlipala, A.: A certified type-preserving compiler from lambda calculus to assembly language. In: PLDI'07: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation. (June 2007)

[7] Ørbæk, P., Palsberg, J.: Trust in the lambda-calculus. J. Funct. Program. **7**(6) (1997) 557–591

[8] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)

[9] Rimsa, A., d'Amorim, M., Pereira, F.M.Q.a.: Tainted flow analysis on e-ssa-form programs. In: Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software. CC'11/ETAPS'11, Berlin, Heidelberg, Springer-Verlag (2011) 124–143

[10] Kothari, S., Caldwell, J.: A machine checked model of idempotent mgu axioms for lists of equational constraints. In Fernandez, M., ed.: UNIF. Volume 42 of EPTCS. (2010) 24–38

[11] McBride, C.: First-order unification by structural recursion. J. Funct. Program. **13**(6) (2003) 1061–1075

[12] Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)

[13] Barendrecht, H.P.: The Lambda Calculus: its Syntax and Semantics. Volume 103 of Studies in Logic and the Foundations of Mathematics. Elsevier (1984)
[14] Hindley, J.R., Seldin, J.P.: Lambda-Calculus and Combinators: An Introduction. 2 edn. Cambridge University Press, New York, NY, USA (2008)
[15] Lindholm, T., Yellin, F.: Java Virtual Machine Specification. 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
[16] Wallach, D.S., Appel, A.W., Felten, E.W.: Safkasi: a security mechanism for language-based systems. ACM Trans. Softw. Eng. Methodol. **9**(4) (October 2000) 341–378
[17] Terei, D., Mazires, D., Marlow, S., Peyton Jones, S.: Safe Haskell. In: Haskell '12: Proceedings of the Fifth ACM SIGPLAN Symposium on Haskell, ACM (2012)
[18] S. P. Jones and others: GHC — The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/` (1998)
[19] Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2-3) (January 1996) 167–187
[20] Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5) (May 1976) 236–243
[21] Heintze, N., Riecke, J.G.: The slam calculus: programming with secrecy and integrity. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '98, New York, NY, USA, ACM (1998) 365–377
[22] Myers, A.C.: Jflow: practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '99, New York, NY, USA, ACM (1999) 228–241
[23] Andrew Myers and others: Jif Compiler. `http://www.cs.cornell.edu/jif/` (1998)
[24] Clarkson, M.R., Chong, S., Myers, A.C.: Civitas: Toward a secure voting system. Security and Privacy, IEEE Symposium on **0** (2008) 354–368
[25] Barthe, G., Rezk, T., Basu, A.: Security types preserving compilation. Comput. Lang. Syst. Struct. **33**(2) (July 2007) 35–59
[26] Kammüller, F.: Formalizing non-interference for a simple bytecode language in coq. Formal Asp. Comput. **20**(3) (2008) 259–275
[27] Chlipala, A.: Static checking of dynamically-varying security policies in database-backed applications. In: OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation. (October 2010)
[28] Charguéraud, A.: The Locally Nameless Representation. Journal of Automated Reasoning (May 2011) 1–46
[29] Koprowski, A.: Coq formalization of the higher-order recursive path ordering. Applicable Algebra in Engineering, Communication and Computing (AAECC) **20**(5-6) (2009) 379–425
[30] Wisnesky, R., Malecha, G., Morrisett, G.: Certified web services in Ynot. In: 5th International Workshop on Automated Specification and Verification of Web Systems. (July 2009)
[31] Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7) (2009) 107–115
[32] Barthe, G., Dufay, G., Jakubiec, L., Sousa, S.a.M.d.: A formal correspondence between offensive and defensive javacard virtual machines. In: Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation. VMCAI '02, London, UK, UK, Springer-Verlag (2002) 32–45

$$\frac{}{(\lambda\, x : \tau . e_1)\, v_2 \;\to\; [x \;\mapsto\; v_2]\, e_1} \;\text{(App)}$$

$$\frac{}{(\lambda\, x : \tau . e_1)\, u_2 \;\to\; [x \;\mapsto\; u_2]\, e_1} \;\text{(App}_u\text{)}$$

$$\frac{e_1 \;\to\; e_1'}{e_1\; e_2 \;\to\; e_1'\; e_2} \;\text{(App}_1\text{)}$$

$$\frac{e_2 \;\to\; e_2'}{v_1\; e_2 \;\to\; v_1\; e_2'} \;\text{(App}_2\text{)}$$

$$\frac{e_2 \;\to\; e_2'}{u_1\; e_2 \;\to\; u_1\; e_2'} \;\text{(App}_{2u}\text{)}$$

$$\frac{e \;\to\; e'}{\texttt{trust}\; e \;\to\; \texttt{trust}\; e'} \;\text{(Trust}_1\text{)}$$

$$\frac{e \;\to\; e'}{\texttt{distrust}\; e \;\to\; \texttt{distrust}\; e'} \;\text{(Distrust}_1\text{)}$$

$$\frac{e \;\to\; e'}{\texttt{check}\; e \;\to\; \texttt{check}\; e'} \;\text{(Check}_1\text{)}$$

$$\frac{}{\texttt{trust}(\texttt{distrust}\; v) \;\to\; \texttt{trust}\; v} \;\text{(Trust}_c\text{)}$$

$$\frac{}{\texttt{distrust}(\texttt{distrust}\; v) \;\to\; \texttt{distrust}\; v} \;\text{(Distrust}_c\text{)}$$

$$\frac{}{(\texttt{distrust}\ (\lambda x : \tau.\, e))\, v \;\to\; \texttt{distrust}\ ([x \mapsto v]\, e)} \;\text{Distrust}_{ca1}$$

$$\frac{}{(\texttt{distrust}\ (\lambda x : \tau.\, e))\, u \;\to\; \texttt{distrust}\ ([x \mapsto u]\, e)} \;\text{Distrust}_{ca2}$$

$$\frac{}{\texttt{trust}\; v \;\to\; v} \;\text{(Trust}_v\text{)}$$

$$\frac{}{\texttt{check}\; v \;\to\; v} \;\text{(Check}_v\text{)}$$

Fig. 5. Small-step Operational Semantics

$$\frac{s \preceq s'}{\texttt{bool}^{s} \leq \texttt{bool}^{s'}} \;\text{(S-Bool)}$$

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'} \;\text{(S-Arrow)}$$

$$\frac{}{\tau \leq \tau} \;\text{(S-Refl)}$$

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \;\text{(S-Trans)}$$

Fig. 6. Subtyping Relation

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}^{tr}} \ (\texttt{T-True})$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}^{tr}} \ (\texttt{T-False})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \ (\texttt{T-Var})$$

$$\frac{\Gamma \vdash e_1 : (\tau \to t^{s'})^s \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : t^{(s' \vee s)}} \ (\texttt{T-App})$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : (\tau \to \tau')^{tr}} \ (\texttt{T-Abs})$$

$$\frac{\Gamma \vdash e : t^s}{\Gamma \vdash \texttt{trust}\ e : t^{tr}} \ (\texttt{T-Trust})$$

$$\frac{\Gamma \vdash e : t^s}{\Gamma \vdash \texttt{distrust}\ e : t^{dis}} \ (\texttt{T-Distrust})$$

$$\frac{\Gamma \vdash e : t^{tr}}{\Gamma \vdash \texttt{check}\ e : t^{tr}} \ (\texttt{T-Check})$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \ (\texttt{T-Sub})$$

Fig. 7. Type System for λ-calculus with Trust Types

$$\frac{}{\Gamma \vdash^D \texttt{true} : \texttt{bool}^{tr}} \ (\texttt{D-True})$$

$$\frac{}{\Gamma \vdash^D \texttt{false} : \texttt{bool}^{tr}} \ (\texttt{D-False})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash^D x : \tau} \ (\texttt{D-Var})$$

$$\frac{\Gamma \vdash^D e_1 : (\tau \to t^{s'})^s \quad \Gamma \vdash^D e_2 : \tau' \quad \tau' \leq_D \tau}{\Gamma \vdash^D e_1\ e_2 : t^{(s' \vee s)}} \ (\texttt{D-App})$$

$$\frac{\Gamma, x : \tau \vdash^D e : \tau'}{\Gamma \vdash^D \lambda x : \tau. e : (\tau \to \tau')^{tr}} \ (\texttt{D-Abs})$$

$$\frac{\Gamma \vdash^D e : t^u}{\Gamma \vdash^D \texttt{trust}\ e : t^{tr}} \ (\texttt{D-Trust})$$

$$\frac{\Gamma \vdash^D e : t^u}{\Gamma \vdash^D \texttt{distrust}\ e : t^{dis}} \ (\texttt{D-Distrust})$$

$$\frac{\Gamma \vdash^D e : t^{tr}}{\Gamma \vdash^D \texttt{check}\ e : t^{tr}} \ (\texttt{D-Check})$$

Fig. 8. Syntax Directed Type System for λ-calculus with Trust Types

```
Definition
   lubtrustty (x y : trustty):trustty :=
     match x with
      | Tr    => y
      | Dis  => Untrust
     end.

Definition
   updatetrustty
      (t : ty) (s : trustty) : ty :=
        match t with
          | ty_bool s' =>
            ty_bool (lub_trustty s s')
          | arrow l r s' =>
            arrow l r (lub_trustty s s')
        end.
```

Fig. 9. Functions for least upper bound over trust types

```
Fixpoint erase_ty (t : ty) : stlc_ty :=
  match t with
    | ty_bool _ => stlc_bool
    | arrow l r _ => stlc_arrow (erase_ty l)
                                (erase_ty r)
  end.

Fixpoint erase_term (t : term) : stlc_term :=
  match t  with
    | tm_false => stlc_false
    | tm_true => stlc_true
    | tm_var i => stlc_var i
    | tm_app l r => stlc_app (erase_term l)
                             (erase_term r)
    | tm_abs i T t
        => stlc_abs i (erase_ty T)
                      (erase_term t)
    | tm_trust t => erase_term t
    | tm_distrust t => erase_term t
    | tm_check t => erase_term t
  end.

Definition erase_context (ctx : context) :=
  map (fun p => match p with
                  | (i,t) => (i, erase_ty t)
                end) ctx.
```

Fig. 10. Erasure Functions

$$\frac{}{\Gamma \vdash^C true : bool}\ \text{(TC-True)} \qquad \frac{}{\Gamma \vdash^C false : bool}\ \text{(TC-False)}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash^C x : \tau}\ \text{(TC-Var)} \qquad \frac{\Gamma, x : \tau \vdash^C e : \tau'}{\Gamma \vdash^C \lambda x : \tau . e : \tau \to \tau'}\ \text{(TC-Abs)}$$

$$\frac{\Gamma \vdash^C e_1 : \tau \to \tau' \quad \Gamma \vdash^C e_2 : \tau}{\Gamma \vdash^C e_1\, e_2 : \tau'}\ \text{TC-App}$$

Fig. 11. Church-Style Type System for λ-calculus

Herinneringen aan SDS

John-Jules Meyer

Doaitse jaagt mensen vaak schrik aan. Zijn scherpe verstand gecombineerd met een scherpe tong zal daar debet aan zijn. Ik waardeer die kwaliteit wel in hem. Sterker nog, ik zal zijn ongezouten mening over van alles en nog wat, maar zeker ook over het wel en wee van de universiteit / faculteit / departement behoorlijk missen. Doaitse zegt niet religieus te zijn maar is vaak Roomser dan de Paus, in zijn geval misschien adequater geformuleerd als gereformeerder dan een dominee...

Alhoewel Doaitse naar mijn bescheiden mening nonstandaard opvattingen heeft over logica, wat wel eens aanieiding gaf voor pittige discussies (vooral in de tijd dat we naast elkaar een kantoor hadden, wat trouwens ook zijn trouwe hulp met Mac-zaken betekende), hebben we eigenlijk nooit in de twintig jaar dat we samen in het departement zaten echte ruzie gehad. Een onenigheid over verschillen en overeenkomsten tussen software technologie en programmatuurkunde, onze respectievelijke leeropdrachten, leverde een gezamenlijke promovendus op, de getalenteerde Tanja Vos, die is gepromoveerd op semi-automatische (mbv HOL) correctheid van gedistribueerde algoritmen [3]. Dus alhoewel we zeker verschilden, hadden we ook gezamenlijke interessen en overeenkomsten. (Een andere overeenkomst is dat we, in absolute waarde, beiden behoorlijk fortuinlijk zijn... ;-))

Ik herinner me dat hij als lid van mijn sollicitatiecommissie vroeg naar mijn mening over het bekende UNITY-boek van Chandy & Misra [2]. Ik zei zoiets als dat ik het een mooi boek vond en dat ik er wel uit wilde gaan les geven. Dat is er nooit van gekomen, maar alhoewel ik tegenwoordig veel meer in de AI zit dan in de concurrency is het grappig dat UNITY wel nog steeds af en toe een rol speelt in ons werk, bijv. in het werk van onze oud-promovenda 'Lara' [1]. En in de laatste paper van Max Knobbout [4] wordt het gebruikt als een basis voor een kader voor multi-agent systemen, speltheorie en mechanism design.

Doaitse zei me ooit dat "agents zijn ding niet waren, maar dat het wel wetenschap was." Dat is het grootste compliment dat ik uit de mond van Doaitse kon horen...! Doaitse, het ga je goed. Ik zal je missen!

References

1. L. Astefanoaei, F. S. de Boer, M. Dastani & J.-J. Ch.Meyer, A Weakest Precondition Calculus for BUnity, accepted for Science of Computer Programming.
2. K.M. Chandy & J. Misra, Parallel Program Design, A Foundation, Addison-Wesley, Reading (MA),1988.
3. T. Vos, UNITY in Diversity, a Stratified Approach to the Verification of Distributed Algorithms, PhD Thesis, Universiteit Utrecht, 2000
4. M. Knobbout, M. Dastani & J.-J. Ch. Meyer, Multi-Agent Systems, Norms and Mechanism Design: A Logical Approach, submitted.

Type Checking by Domain Analysis in Ampersand

Stef Joosten[1,2]

[1] Open University of the Netherlands.
Postbus 2960.
6401 DL HEERLEN
[2] Ordina NV
stef.joosten@ou.nl

Abstract. This paper proposes a type system for relation algebras. It uses domain analysis to derive types and prove type correctness. The algorithm features overloading and specialization, which makes it suitable for practice. It has been used in a language called Ampersand, resulting in the use of relation algebras as a programming language for information systems.
The purpose of this paper is to document the idea that type checking can be done by a domain analysis on relation terms. We illustrate this by showing how the domain analysis of the relation algebra used in Ampersand provides the information needed to do type checking. As a result, type inference is no longer needed as an algorithm for type checking. Instead a graph algorithm is used.

1 Acknowledgement

I have always admired Doaitse's thesis [13] about Lawine, which opened my eyes to type checking. Back then, Doaitse showed that many runtime errors can be detected at compile time. This made the type checker for Lawine something of an early warning system. I hope that my contribution to the Liber Amicorum appeals to Doaitse's love of type systems.

The story starts with a present that I received for my birthday from my son, Sebastiaan, on April 12th, 2012. He had written a toy type checker for relation algebras. He gave a demo, showing me how to solve a problem we were having in the Ampersand compiler. Sebastiaan's prototype is readily executable in GHCi and is printed in the appendix of this contribution. So type checking by domain analysis is really his idea. My contribution was to develop it further, so that it works properly in the Ampersand compiler, which is written in Haskell. It took me until April 12th, 2013 to complete that work, not without the necessary contributions of Sebastiaan. The Ampersand compiler also owes to Doaitse Swierstra, because it still uses Doaitse's parser combinators rather than parsec. Which brings me back to Doaitse, whose work I appreciate so much for being useful in practice. The Ampersand compiler is vivid proof for that. So how could I say no to the request for a contribution to Doatse's Liber Amicorum? Let this contribution be my way to say: Doaitse, thank you for your contributions and lessons I have learned from your work!

2 Introduction

Ampersand [7] is a formal language, meant to describe business processes by means of rules. It is used by requirements engineers, who formalize an "agreed language" and express business requirements in that language. Ampersand uses a heterogeneous relation algebra [6] for that purpose. This approach defines business process(es) by constraints. Each script written in Ampersand is in essence a theory in relation algebra.

A wealth of theory has been developed for binary relations (Tarski [14], Schmidt and Ströhlein [10], Freyd and Scedrov [3], Backhouse et al [1], Maddux [6], etc), both in the homogeneous and heterogeneous versions. This theory has many nice applications (Brink et al [2]). To this already impressive number of applications, Ampersand adds another: a new way to design information systems by means of business rules. Ampersand means to unleash the manipulative power of relation algebra for constructing information systems that support business processes. This construction is done by means of a compiler, which translates an Ampersand script into an information system. The generated system consists of a MySQL database and PHP web services. The compiler can be used to specify real life systems and has already been used successfully in industry as well as in education.

Ampersand is a (syntactically sugared) implementation of relation algebra. Like other relational specification languages such as Alloy [5] and Z [12, 15], Ampersand uses rules, defined by relation algebra terms, as constraints. Yet, Ampersand's computational model differs from the conventional relational theory of computing [9]. Ampersand does not specify a computer program by constraining the relation between its input and output. Rules in Ampersand are interpreted as constraints on a data space. That data space reflects the state of a business process. It may be perceived as that part of corporate data, which is needed in the daily operation of business processes in organisations. We have built a compiler which translates these rules into working software to support that business process.

Thus, Ampersand is an approach to model an organization as a collection of rules, pretty much in accordance with the ideas of the Business Rules Approach [4]. The tools that support Ampersand generate overviews, functional specifications, and working software applications. By that nature, Ampersand relates to:

- business rules approaches e.g. RuleSpeak, SBVR.
  These approaches share with Ampersand the intention to grasp business rules in conjunction with their meaning. They differ from Ampersand in that Ampersand is an instance of heterogeneous relation algebra [6].
- system specification techniques on formal rules e.g. Alloy and Z.
  These approaches share with Ampersand their use of relation algebra. Ampersand differs from these approaches in its interpretation of the algebra. Ampersand interprets a theory in the algebra as a set of constraints on data, whereas Alloy, Z, and the mathematics of rule complexes use the algebra to constrain program specifications.
- organization engineering e.g. DEMO. This approach shares with Ampersand the intention to define business processes, in a way that is directly useful for producing an information system that supports that process. Ampersand features a generator that actually produces such an information system.
- model-driven engineering e.g. MDA with UML [8]. Model driven approaches (e.g. [11]) have the generation of software in common with Ampersand. Most approaches differ from Ampersand with respect to the modelling activity. Ampersand generates conceptual models, process models and data models, whereas most Model driven approaches take such models as their input.

This paper is about a specific component of Ampersand: the type system. Ampersand perceives the heterogeneous version of relation algebra as a formal language with a type system. The language insists that every relation be declared explicitly, e.g.

$$ssn : Person \times SocialSecurityNumber$$
$$currentAddress : Person \times Adress$$

In order to interpret a term such as $ssn^{\smile}; currentAddress$, a typechecker verifies that the types of both relations allow this term to be implemented in software.

The requirements for this type checker are:

- A type correct program is a representable heterogeneous relational algebra, so the result can be implemented in a database system.
- The type system respects the Tarski axioms as much as possible, so the user can calculate with terms as in relation calculus.
- The type system supports generalization and overloading, in order to facilitate combining scripts of different authors into one.

Using the conventional style of type checking, in which type inferencing is done on the structure of terms, the proof of soundness and completeness was complicated due to (necessary) features such as specialization. Even small parts of that proof became too complicated to be convincing. The authors have replaced type inferencing by an algorithm that does domain analysis. This analysis itself provides the proof of type correctness (or incorrectness, for that matter). If all terms have precisely one type, the type checker can prove that by information provided by the domain analysis. In this paper we shall give examples of such proofs in the form of diagrams.

The discussion starts by positioning the language Ampersand as a language for specifying information systems. Section 3 introduces the abstract syntax of Ampersand. The semantics are introduced in Section 4. The idea of type checking by domain analysis is introduced in the next section, immediately followed by the algorithm.

3 Definitions

This section defines the Ampersand language for the sole purpose of understanding the type system.

Atoms are values that have no internal structure. They are the units of data that are stored in an information system. By convention throughout the remainder of this paper, variables a, b, c, and d are used to represent *atoms*. The set of all atoms is called $\mathbb{A}$.

From a business perspective, atoms are used to represent concrete items of the world, such as `Peter`, `1`, or `the king of France`. From a formal perspective, each atom is an instance of a *concept*.

Concepts are names we use to classify atoms in a meaningful way. For example, you might choose to classify `Peter` as a person, and 074238991 as a telephone number. We will use variables A, B, C, D to represent concepts. The set of all concepts is called $\mathbb{C}$.

Definition 1 (instance).
For each atom a and concept A, the expression $a \in A$ means that atom a is an instance of concept A.

In the syntax of Ampersand, concepts form a separate syntactic category, allowing a parser to recognize them as concepts.

Ampersand uses relations to represent sets of facts (i.e. statements that are true in a business context), which can be stored and maintained as data in a computer. As data changes over time, so do the contents of these relations. The representation of the data in the relations in Ampersand is governed by the constraints. When users (or computers) change the actual content of relations, some of these rules may be violated and the system must be brought back in a state where these constraints are satisfied. This particular use explains why Ampersand uses one specific interpretation of relation algebra (Definition 7).

Relations are used to represent (binary) facts, i.e. basic sentences which are true in a business context $\mathcal{C}$. Basic sentences are terms of the form $a\ r\ b$, in which a and b are atoms and r is a relation. If that sentence is true in a business context $\mathcal{C}$, we write

$$\mathcal{C} \vdash a\ r\ b \tag{1}$$

or just $a\ r\ b$ if there is only one context.

Definition 2 (relation declaration). *The declaration of $A\ r\ B$ in an Ampersand script states that A and B exist as concepts and there exists a relation r between them.*

The declaration of $A\ r\ B$ in context $\mathcal{C}$ makes any expression $a\ r\ b$ meaningful in $\mathcal{C}$, i.e. true or false in that context.

Definition 3 (generalization). *The declaration of $A \preceq B$ (pronounce: A is a B) in an Ampersand script states that any instance of A is an instance of B as well.*

Generalization is needed to allow statements such as: "An Employee is a Person that ....".

Definition 4 (relation declarations).
The set of relations in context $\mathcal{C}$ is a finite set $\mathbb{D}_{\mathcal{C}}$ that is defined by:

1. *if $A\ r\ B$ is declared in the Ampersand script, then $A\ r\ B \in \mathbb{D}_{\mathcal{C}}$*
2. *for every $A, B \in \mathbb{C}$: $\mathbb{I}_A, \mathbb{V}_{A*B} \in \mathbb{D}_{\mathcal{C}}$*
3. *for every $a, b \in A$: $a = b$ is equivalent to $a\ \mathbb{I}_A\ b\ \in\ \mathbb{D}_{\mathcal{C}}$*
4. *for every $a \in A, b \in B$: $a\ \mathbb{V}_{A*B}\ b\ \in\ \mathbb{D}_{\mathcal{C}}$*

The relation $\mathbb{I}_A$ is called the *identity relation* of concept A. The relation $\mathbb{V}_{A*B}$ is called the *universal relation* over concepts A and B. If there is but one context, we write $\mathbb{D}$ rather than $\mathbb{D}_{\mathcal{C}}$.

The set of relation terms, $\mathbb{R}$, is defined by Definition 5. By convention throughout the remainder of this paper, variables r, s, t, p, and q are elements of a set $\mathbb{R}$, which contains all relations and relation terms. Relation terms can be combined by operators to form new relation terms.

Definition 5 (relation terms).
Let $r, s \in \mathbb{R}$. The set of relation terms, $\mathbb{R}$, is the smallest set that satisfies

5. *$r \in \mathbb{R}$ for every $(A\ r\ B) \in \mathbb{D}$*
6. *$r_{[A,B]} \in \mathbb{R}$ for every $(A\ r\ B) \in \mathbb{D}$*
7. *$(r \cup s) \in \mathbb{R}$*
8. *$(r \cap s) \in \mathbb{R}$*
9. *$(r - s) \in \mathbb{R}$*
10. *$(r; s) \in \mathbb{R}$*
11. *$r^{\smile} \in \mathbb{R}$*

Definition 6 (rule terms).
Let $r, s \in \mathbb{R}$. The set of rule terms, $\mathbb{U}$, is defined by.

12. *$(RULE\ r \subseteq s) \in \mathbb{U}$*
13. *$(RULE\ r \equiv s) \in \mathbb{U}$*

These definitions introduce the binary operators $\cup$, $\cap$, $-$, and ;, the unary operator $^{\smile}$, two relation constants $\mathbb{I}$ and $\mathbb{V}$, and two ways of expressing rules by means of relation terms. This constitutes the algebra that Ampersand uses to express business rules.

4 Semantics

The meaning of relation terms in Ampersand is defined by an interpretation function $\mathfrak{I}_{\mathcal{C}}$ (definition 7). It maps each relation term to a set of facts that are true within context $\mathcal{C}$. Let $r \in \mathbb{R}$, and $a \in A$ and $b \in B$, then

$$\mathcal{C} \vdash a\ r\ b \quad \Leftrightarrow \quad \langle a, b\rangle \in \mathfrak{I}_{\mathcal{C}}(r)$$

In the sequel, we restrict the situation to a single context $\mathcal{C}$, so we avoid specifying context $\mathcal{C}$:

$$a\ r\ b \quad \Leftrightarrow \quad \langle a, b\rangle \in \mathfrak{I}(r) \tag{2}$$

Let A and B be finite sets of atoms, then $\mathfrak{I}$ maps all terms to the set of facts for which that term stands.

Definition 7 (interpretation of relation terms).
For every $a, b, c \in \mathbb{A}$, $A, B \in \mathbb{C}$ *and* $r, s \in \mathbb{R}$

$$\begin{array}{lr}
\textit{union:} & \mathfrak{I}(r \cup s) = \{\langle a, b\rangle |\ \langle a, b\rangle \in \mathfrak{I}(r)\ \textit{or}\ \langle a, b\rangle \in \mathfrak{I}(s)\} \\
\textit{intersection:} & \mathfrak{I}(r \cap s) = \{\langle a, b\rangle |\ \langle a, b\rangle \in \mathfrak{I}(r)\ \textit{and}\ \langle a, b\rangle \in \mathfrak{I}(s)\} \\
\textit{difference:} & \mathfrak{I}(r - s) = \{\langle a, b\rangle |\ \langle a, b\rangle \in \mathfrak{I}(r)\ \textit{and}\ \langle a, b\rangle \notin \mathfrak{I}(s)\} \\
\textit{composition:} & \mathfrak{I}(r; s) = \{\langle a, c\rangle |\ \textit{for some } b,\ \langle a, b\rangle \in \mathfrak{I}(r)\ \textit{and}\ \langle b, c\rangle \in \mathfrak{I}(s)\} \\
\textit{converse:} & \mathfrak{I}(r^{\smile}) = \{\langle b, a\rangle |\ \langle a, b\rangle \in \mathfrak{I}(r)\} \\
\textit{type cast:} & \mathfrak{I}(r_{A*B}) = \{\langle a, b\rangle |\ a\ r\ b\ \wedge\ a \in A\ \wedge\ b \in B\} \\
\textit{relation:} & \mathfrak{I}(r) = \{\langle a, b\rangle |\ a\ r\ b\} \\
\textit{identity:} & \mathfrak{I}(I_A) = \{\langle a, a\rangle |\ a \in A\} \\
\textit{full relation:} & \mathfrak{I}(V_{A*B}) = \{\langle a, b\rangle |\ a \in A, b \in B\}
\end{array}$$

Our type system is based on domain analysis. For that reason we need definitions for the domain and codomain of terms. For every relation r, there exist two sets of atoms, $dom(r)$ and $cod(r)$.

Definition 8 (domain and codomain).
For every $a, b \in \mathbb{A}$ *and* $r \in \mathbb{R}$

$$\mathrm{dom}(r) = \{a | (a, b) \in \mathfrak{I}(r)\} \tag{3}$$
$$\mathrm{cod}(r) = \{b | (a, b) \in \mathfrak{I}(r)\} \tag{4}$$

Note that $cod(r) = dom(r^{\smile})$. The atoms of a concept A are defined by

Definition 9 (population).

$$\mathrm{pop}(A) \ = \ \{a | a \in A\}$$

This definition implies that a concept may be perceived as a set of atoms.

Definition 10 (rule). *A rule is a term* r *that is true in every situation. So for each atom* a *and* b*:*

$$a\ r\ b$$

5 Typing

Suppose that for every relation r precisely two concepts exist, $src(r)$ and $trg(r)$, for which

$$a\ r\ b \quad \Leftrightarrow \quad a \in src(r)\ \wedge\ b \in trg(r) \tag{5}$$

then we can define the type of a relation term r by:

Definition 11 (type of relation terms). *The* type *of a relation term r is the pair* $\langle \mathrm{src}(r), \mathrm{trg}(r) \rangle$.

The task of the type system is to ensure that $src(r)$ and $trg(r)$ are uniquely defined for each relation term r. This task becomes slightly more complicated due to generalization, which is specified in an Ampersand script by means of $\preceq$. Let $\sqsubseteq$ be the transitive, reflexive closure of $\preceq$. The Ampersand script must ensure that $\sqsubseteq$ is antisymmetric by avoiding cycles in the structure of $\preceq$. The compiler will produce error messages when an Ampersand script contains such cycles. As a result of being transitive, reflexive, and antisymmetric, $\sqsubseteq$ is a partial order of concepts. By adding $\top$ (pronounced: anything) and $\bot$ (pronounced: nothing), $\sqsubseteq$ becomes a lattice. That allows us to define the operators 'join' ($\sqcup$) as the least upper bound of $\sqsubseteq$ and 'meet' ($\sqcap$) as its greatest lower bound.

The lattice $\sqsubseteq$ is extended to types in a straightforward manner:

$$\langle A, B \rangle \sqsubseteq \langle C, D \rangle \quad \Leftrightarrow \quad A \sqsubseteq C\ \wedge\ B \sqsubseteq D \tag{6}$$

This allows us to define the types of terms as follows:

Definition 12 (type of relation terms).
For every $A, B \in \mathbb{C}$ and $r, s \in \mathbb{R}$

$$\begin{aligned}
\mathfrak{T}(r \cup s) &= \mathfrak{T}(r) \sqcup \mathfrak{T}(s) \\
\mathfrak{T}(r \cap s) &= \mathfrak{T}(r) \sqcap \mathfrak{T}(s) \\
\mathfrak{T}(r - s) &= \mathfrak{T}(r) \\
\mathfrak{T}(r; s) &= \langle \mathrm{src}(r), \mathrm{trg}(s) \rangle \\
\mathfrak{T}(r^{\smile}) &= \langle \mathrm{trg}(r), \mathrm{src}(r) \rangle \\
\mathfrak{I}(r_{A*B}) &= \langle A, B \rangle \\
\mathfrak{T}(r) &= \langle \mathrm{src}(r), \mathrm{trg}(r) \rangle \\
\mathfrak{T}(I_A) &= \langle A, A \rangle \\
\mathfrak{T}(V_{A*B}) &= \langle A, B \rangle
\end{aligned}$$

In order to define the difference between a correct and an incorrect script, Ampersand requires that every atom in a relation is an instance of a concept. Because all type checking is done at compile time, the type checker must prove that without knowing which atoms exist.

For example, if $dom(r) = A$ and $dom(s) = B$, then $dom(r \cap s) = A \sqcap B$. If $A \sqcap B = \bot$ there is no concept of which the atoms are an instance. So the type checker complains and Ampersand produces type errors only. This can be resolved in the script by defining a concept C to accommodate such atoms. If, for example, the script would state that $C \preceq A$ and $C \preceq B$, the type system would deduce $dom(r \cap s) = C$ and the type error would be resolved.

A mistake can originate from an expression with $\cup$ or $\cap$, whose type is defined in terms of join and meet. It can also come from a composition $r; s$, if $cod(s) \sqcap dom(r)$ turns out to be $\bot$. Also, a mistake can occur in a type cast r_{A*B}, if there is no unique $C\ r\ D$ that satisfies $\langle A, B \rangle \sqsubseteq \langle C, D \rangle$.

Ampersand uses Definition 12 as a specification of a type correct script. If the script satisfies these rules, and each term has precisely one type, the script is approved. Each approved script can be processed further. If not, the mistakes are reported and no further processing takes place. In the sequel, we present an algorithm to perform this task.

6 Domain analysis

Let us introduce the idea of domain analysis by means of an example. Consider the Ampersand script of Listing 1.1. This script introduces three relations, `r[A*C]`, `s[A*B]`, and `t[B*C]`

Listing 1.1. A type correct Ampersand script

```
1  CONTEXT Test
2
3  PATTERN Test1
4  RELATION r[A*C]
5  RELATION s[A*B]
6  RELATION t[B*C]
7  RULE r = s;t
8  ENDPATTERN
9
10 ENDCONTEXT
```

and one rule: "`r = s;t`". This rule has the following terms: "`r`", "`s;t`", "`s`", and "`t`".

The type system analyses this script by constructing the type graph of Figure 1. Vertices (ellipses) in this diagram represent sets of atoms. If two terms share the same set of atoms, they are located in the same vertex. For example, $cod(r)$ on line 7:6 of the script is in the same vertex as $cod(r_{A*C})$ on line 4:1, which means that they have the same set of atoms. An edge (arrow) means that the type system has detected a subset relation between two vertices. Figure 1 shows, for instance, an arrow from $dom(r)$ on line 7:6 to $pop(A)$, meaning that $dom(r)$ on line 7:6 is a subset of $pop(A)$. Similarly, the type graph shows that $cod(r)$ on line 7:6 is a subset of $pop(C)$. In this way, the type graph shows that the type of term r on line 7:6 is $\langle A, C\rangle$. If a script is correct, the type graph provides the information to establish the types of all terms.

The type graph can be used to make proofs of type correctness. For instance, figure 1 supports the following two derivations:

$$
\begin{array}{lll}
 & dom(s;t) & \text{(line 7:11)} \\
= & & \\
 & dom(s) & \text{(line 7:10)} \\
= & & \\
 & dom(s_{A*B}) & \text{(line 5:1)} \\
\subseteq & & \\
 & pop(A) &
\end{array}
\qquad
\begin{array}{lll}
 & cod(s;t) & \text{(line 7:11)} \\
= & & \\
 & cod(t) & \text{(line 7:12)} \\
= & & \\
 & cod(t_{B*C}) & \text{(line 6:1)} \\
\subseteq & & \\
 & pop(C) &
\end{array}
$$

Together these two derivations prove that term $s;t$ has type $\langle A, C\rangle$. The question is: why make such proofs in the first place? The graph itself contains all proofs that are relevant in the compactness of a single diagram.

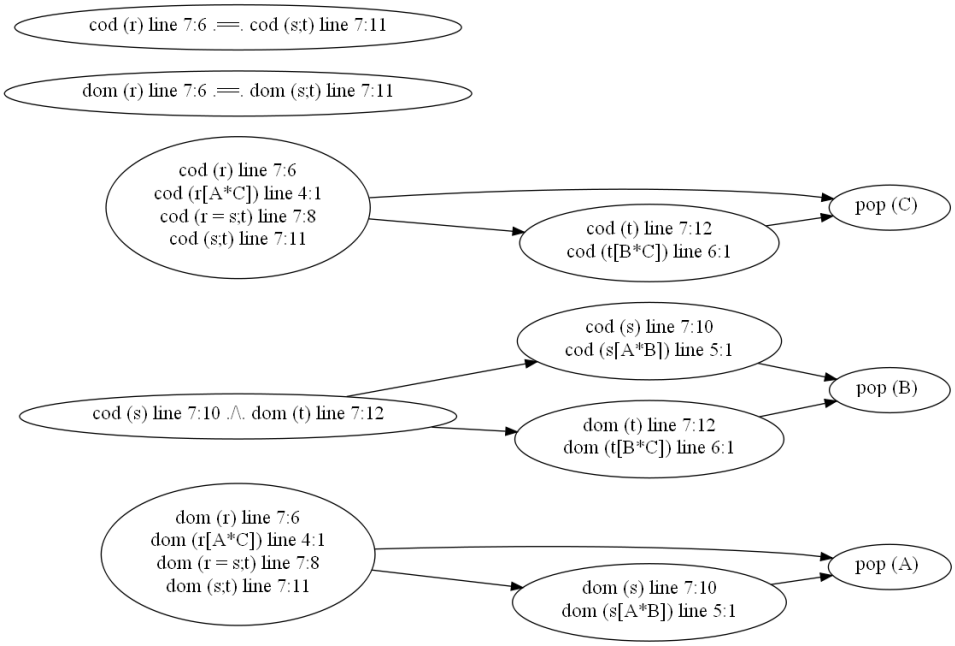


Fig. 1. Type graph for listing 1.1

So what if there are mistakes in the script? Consider a script with a type error in listing 1.2. The type graph of that script is represented in figure 2. There are vertices in this

Listing 1.2. A type incorrect Ampersand script

```
1  CONTEXT Test
2
3  PATTERN Test2
4  RELATION r[A*C]
5  RELATION s[B*A]
6  RELATION t[B*C]
7  RULE r = s;t
8  ENDPATTERN
9
10 ENDCONTEXT
```

graph that have paths to two different concepts, A and B. From the domain analysis, the type checker can prove that the domain of these terms is a subset of A, but a subset of B as well. That violates the rule that each term must have precisely one type. The type checker will reject this script with the following error message.

```
line 7:8, file "try2.adl"
    Ambiguous equation  r = s;t
```

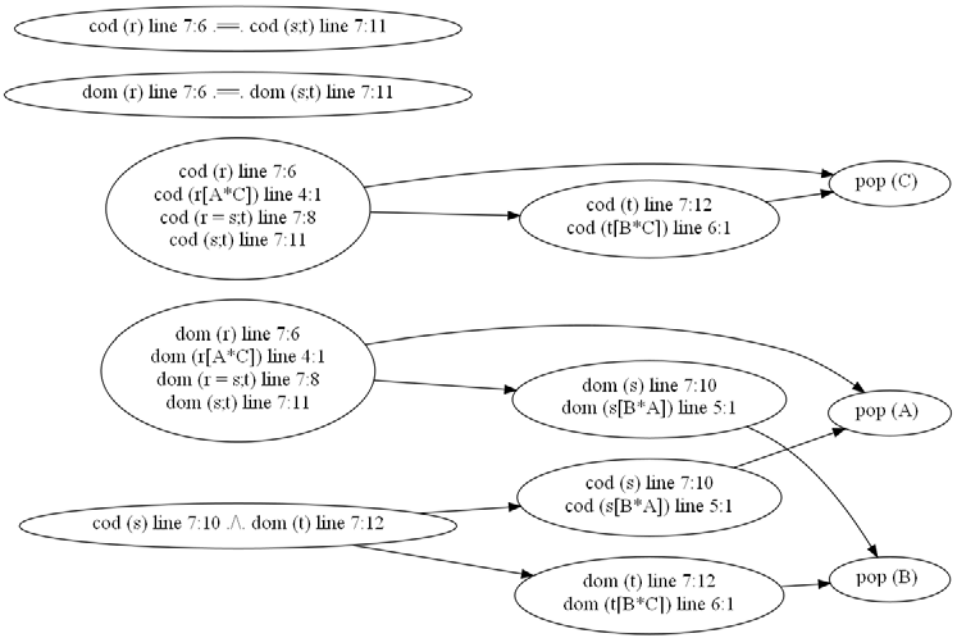


Fig. 2. Type graph for listing 1.2

Summarizing, the domain analysis yields a type graph, two samples of which are shown in listing 1.1 and listing 1.2[3]. For each term r, the type graph proves which concepts the domain and codomain of r are subsets of. The type checker uses that information to verify that each term has precisely one type.

7 Conclusion

This contribution demonstrates that domain analysis is useful as a mechanism for type checking. In my judgement, the main contribution is the type graph, which represents the required proofs. It simplifies the correctness argument of the type system, making it possible to get a correct type checker in practice. The type graph is a comprehensive representation of all proofs that are needed to show that every term has precisely one type. The algorithm for making the type graph will be documented in a paper yet to be published by Sebastiaan Joosten and myself.

The results obtained with this approach in the Ampersand compiler are encouraging. We have managed to incorporate overloading and generalization in the relation algebra, which was needed to make Ampersand into a practical tool. Besides, we have been able to incorporate union types and intersection types, which was not allowed in the former type checker. The performance of the algorithm seems a little better than before, but we are still gathering statistics to prove it.

[3] The Ampersand compiler can generate these type graphs for documentation purposes.

8 Appendix

This appendix gives the Haskell source of a simplified version of this type system, which is meant to communicate the overall idea and play with it in GHCi. The program can be executed as is.

```
{-# OPTIONS_GHC -Wall -XTypeSynonymInstances #-}
{- The SimpleTypeChecker.hs is a prototype ghci-script for the new type checker.
It is meant only for explaining how the type checker works.
Work needs to be done on commenting and cleaning up the code for this purpose.
This file is not meant to be included in the compile sequence for Ampersand.
-}
module SimpleTypeChecker where
  import qualified Data.Set as Set
  import qualified Data.Graph as Graph
  import qualified Data.Tree as Tree
  import Data.List

  data Concept    = Concept String deriving (Eq, Ord)
  data Relation   = Rel String deriving (Eq, Ord)
  data Expression = Pequ  (Expression, Expression) -- equivalence
                  | Pimp  (Expression, Expression) -- implication
                  | ExpRel Relation                -- we expect expressions in flip-normal form
                  | ExpFlp Relation                -- flip / relational inverse
                  | ExpId  Concept                 -- identity element restricted to a type
                  | ExpIsc Expression Expression   -- intersection
                  | ExpUni Expression Expression   -- union
                  | ExpApp Expression Expression   -- apply / compose / append
                  deriving (Eq, Ord)
  data Type       = TypExpr Expression deriving (Eq, Ord, Show)

  class Flippable a where -- notational convenience
   flp :: a -> Expression
  instance Flippable Expression where
   flp (Pequ (a,b)) = flp a .= flp b
   flp (Pimp (a,b)) = flp a .< flp b
   flp (ExpRel a)   = ExpFlp a
   flp (ExpFlp a)   = ExpRel a
   flp (ExpIsc a b) = flp a /\ flp b
   flp (ExpUni a b) = flp a \/ flp b
   flp (ExpApp a b) = flp b .* flp a
   flp (ExpId a)    = ExpId a

  instance Show Relation where
   showsPrec _ (Rel str) = showString (str)

  instance Show Concept where
   showsPrec _ (Concept str) = showString (str)

  instance Show Expression where
   showsPrec _ (Pequ (a,b)) = showString (show a++"␣.=␣"++ show b)
   showsPrec _ (Pimp (a,b)) = showString (show a++"␣.<␣"++ show b)
   showsPrec _ (ExpRel a)   = showString (show a)
   showsPrec _ (ExpFlp a)   = showString (show a++"~")
   showsPrec _ (ExpIsc a b) = showString ("("++show a++"/\\"++ show b++")")
   showsPrec _ (ExpUni a b) = showString ("("++show a++"\\/"++ show b++")")
   showsPrec _ (ExpApp a b) = showString ("("++show a++".*"++ show b++")")
   showsPrec _ (ExpId a)    = showString ("I["++show a++"]")

  -- constructor functions..
  infixl 3 .=
  infixl 3 .<
  infixl 4 \/
  infixl 5 /\
  infixl 7 .*
  (.=) :: Expression -> Expression -> Expression
  (.=) a b = Pequ (a,b)
  (.<) :: Expression -> Expression -> Expression
  (.<) a b = Pimp (a,b)
  (/\) :: Expression -> Expression -> Expression
  (/\) a b = ExpIsc a b
  (\/) :: Expression -> Expression -> Expression
  (\/) a b = ExpUni a b
  (.*) :: Expression -> Expression -> Expression
  (.*) a b = ExpApp a b

  sampleInput :: [Expression]
  sampleInput
   = [ i "Judge" .< i "Person"                                  -- an ISA rule:
     , i "Person" .< i "Judge"                                  -- Person is a judge
     , i "Judge" .* r "presides" .* i "Session"  .= r "presides" -- declaring a relation (simulated)
     , i "Session" .* r "chamber" .* i "Chamber" .= r "chamber"  -- declaring another relation
     , r "presides" .* r "chamber" .= r "resides␣in␣chamber"     -- rule about "resides in chamber"
     -- now some "maybe" type errors:
     , r "sleeps␣on" .< (i "Guy␣Fawkes␣night" /\ i "Sleeper" .* i "Home␣owner") .* flp (r "sleeps␣on")
     -- relation "given" cannot be typed:
     , r "chamber" .< r "given"

     ] where
        r  a = ExpRel (Rel a)
        i  a = ExpId (Concept a)

  subExpressions :: Expression -> [Expression]
  subExpressions (Pequ (a,b)) = [Pequ (a,b)] ++ subExpressions a ++ subExpressions b
  subExpressions (Pimp (a,b)) = [Pimp (a,b)] ++ subExpressions a ++ subExpressions b
```

```
subExpressions (ExpRel a)   = [ExpRel a]
subExpressions (ExpFlp a)   = [ExpFlp a]
subExpressions (ExpIsc a b) = [ExpIsc a b] ++ subExpressions a ++ subExpressions b
subExpressions (ExpUni a b) = [ExpUni a b] ++ subExpressions a ++ subExpressions b
subExpressions (ExpApp a b) = [ExpApp a b] ++ subExpressions a ++ subExpressions b
subExpressions (ExpId a)    = [ExpId a]

-- create unordered / non-increasing pair
unord :: Ord t => (t,t) -> (t,t)
unord (a,b) | a > b     = (a,b)
            | otherwise = (b,a)

typing :: [Expression] -> Set.Set (Type, Type) -- subtypes (.. is subset of ..)
typing exprs = Set.fromList [ t | expr<-exprs, x<-subExpressions expr, t<-uType x]
 where
   uType   (Pequ (a,b)) = [(dom a,dom b), (dom b,dom a), (cod a,cod b), (cod b,cod a)]
   uType   (Pimp (a,b)) = [(dom a,dom b), (cod a,cod b)]
   uType   (ExpRel _)   = []
   uType   (ExpFlp _)   = []
   uType o@(ExpIsc a b) = [(dom o,dom a), (dom o,dom b), (cod o,cod a), (cod o,cod b)]
   uType o@(ExpUni a b) = [(dom a,dom o), (dom b,dom o), (cod a,cod o), (cod b,cod o)]
   uType o@(ExpApp a@(ExpId _) b) = [(dom o,dom b), (dom o,dom a), (cod o,cod b)]
   uType o@(ExpApp a b) = [(dom o,dom a), (cod o,cod b)]
   uType o@(ExpId _)    = [(dom o,dom o)]

   dom = TypExpr
   cod = TypExpr . flp

printTypes :: IO ()
printTypes = foldr1 (>>) (map putStrLn lnes)
  where
    (typeErrors,conceptTree,typedRels,accounting) = calcTypes sampleInput
    lnes = ["Type␣errors:"]++typeErrors
           ++["","Concept␣tree:"]++conceptTree
           ++["","Relations:"]++typedRels
           ++[accounting] -- in order to see intermediate results...

calcTypes :: [Expression] -> ([String],[String],[String],String)
calcTypes sentences = (typeErrors,conceptTree,typedRels,accounting)
 where
  accounting = -- accounting for the results: (for debugging purposes)
    "\nsubexpressions:\n␣␣"++
    intercalate "\n␣␣" ((map show.nub) [x | expr<-sentences, x<-subExpressions expr])++
    "\neq:\n␣␣"++intercalate "\n␣␣" (map show (Set.toList eq))++
    "\neqClasses:\n␣␣"++intercalate "\n␣␣" (map show eqClasses)++
    "\nst:\n␣␣"++intercalate "\n␣␣" (map show (Set.toList st)) ++
    "\nstClasses:\n␣␣"++intercalate "\n␣␣" (map show stClasses)++
    "\neqVerts:\n␣␣"++intercalate "\n␣␣" (map show eqVerts) ++
    "\nstVerts:\n␣␣"++intercalate "\n␣␣" (map show stVerts)
  st :: Set.Set (Type, Type)
  st = typing sentences
  eq :: Set.Set (Type, Type)
  eq = Set.map unord (Set.intersection (Set.map swap st) st) where swap (a,b) = (b,a)
  eqGraph
   = makegraph eqVerts eq
  eqVerts
   = zip [0..] $ Set.toAscList $ Set.union (Set.map fst eq) (Set.map snd eq)
  eqClasses
   = forestToClasses eqVerts (Graph.components eqGraph)
  stVerts
   = zip [0..] $ Set.toAscList $ Set.map getEqClass $
       (Set.union (Set.union (Set.map fst st) (Set.map snd st))
                  (Set.fromList (map (\x -> snd (fst x)) eqClasses)))
  getEqClass vert = head ([a | ((_,a),as)<-eqClasses, Set.member vert as]++[vert])
  stGraph = makegraph stVerts (Set.map (\(x,y) -> (getEqClass x,getEqClass y)) st)
  stClasses
   = forestToClasses stVerts (Graph.scc stGraph)
  makegraph verts tuples
   = Graph.buildG (0,length verts - 1)
       [(i,j) | (k,l)<-Set.toAscList tuples,(i,n)<-verts,k==n,(j,m)<-verts,l==m]
  forestToClasses verts forest
   = map (\x->(verts !! (foldr1 min x)
               , Set.fromList $ map (\y->(snd (verts !! y))) x))
      $ map Tree.flatten forest
  -- type errors come in three kinds:
  -- 1. Two named types are equal.
  --    This is usually unintended: user should give equal types equal names.
  -- 2. The type of a relation cannot be determined.
  --    This means that there is no named type in which it is contained.
  -- 3. The type of a term has no name??
  --    I do not know if these can be considered as type-errors
  --    perhaps if this type is too high up, or too low under
  typeErrors
   = [ "1.␣These␣concepts␣are␣trivially␣equal:␣" ++ intercalate ",␣" (map show concs)
     | (_,as)<-eqClasses, let concs = [c|TypExpr (ExpId c) <- Set.toList as], length concs>1]
     ++
     [ "1.␣These␣concepts␣are␣derived␣to␣be␣equal:␣" ++ intercalate ",␣" (map show concs)
     | (_,as)<-stClasses, let concs = [c|TypExpr (ExpId c) <- Set.toList as], length concs>1]
  conceptTree
   = [show src ++ "␣ISA␣" ++ (
      foldr1 (+++)  trgs)
       | ((i,TypExpr (ExpId src)),_) <- stClasses
       , let trgs= [show tn | (_,TypExpr (ExpId tn))<-map ((!!) stVerts) (Graph.reachable stGraph i)]
       ]
  typedRels
   = [relname++"[␣"++srcnames++"␣*␣"++trgnames++"␣]"
     | (Rel relname) <- Set.toList $ rels sentences
     , let srcnames = nametree (ExpRel (Rel relname))
```

```
188         , let trgnames = nametree (ExpFlp (Rel relname))]
189       nametree e
190        = wrisects [s | Just s<-map getName $ Graph.dfs stGraph [i|(i,tp)<-stVerts,tp==getEqClass (TypExpr e)]]
191       getName :: Tree.Tree Int -> Maybe String
192       getName a
193        = prefer (snd (stVerts !! Tree.rootLabel a)) ([n | Just n <- map getName (Tree.subForest a)])
194       prefer :: Type -> [String] -> Maybe String
195       prefer (TypExpr (ExpId c)) _ = Just (show c)
196       prefer _ [] = Nothing
197       prefer _ as = Just (wrisects as)
198       wrisects [] = "??"
199       wrisects as = foldr1 (\x y-> x ++ "␣/\\␣" ++ y) as
200       (+++) a b = a ++",␣"++ b
201
202     (!!!) :: [a] -> Int -> a
203     (!!!) a b | b >= length a = error "index␣too␣large!"
204               | otherwise = (!!) a b
205
206     rels :: [Expression] -> Set.Set Relation
207     rels x = (Set.fromList . concat . map erels) x
208      where
209       erels (Pequ (a,b)) = erels a ++ erels b
210       erels (Pimp (a,b)) = erels a ++ erels b
211       erels (ExpRel a)   = [a]
212       erels (ExpFlp a)   = [a]
213       erels (ExpId _)    = []
214       erels (ExpIsc a b) = erels a ++ erels b
215       erels (ExpUni a b) = erels a ++ erels b
216       erels (ExpApp a b) = erels a ++ erels b
```

References

1. C. Aarts, R.C. Backhouse, P. Hoogendijk, T.S. Voermans, and J. van der Woude. A relational theory of datatypes. STOP summer school notes, `http://www.cs.nott.ac.uk/~rcb/papers/abstract.html#book`, 1992.
2. C. Brink, W. Kahl, and G. Schmidt, editors. *Relational methods in computer science*, Advances in computing. Springer, 1997.
3. P.J. Freyd and A. Scedrov. *Categories, allegories*. North-Holland Publishing Co., 1990.
4. Business Rules Group. The business rules manifesto. http://www.businessrulesgroup.org/brmanifesto.htm.
5. D. Jackson. A comparison of object modelling notations: Alloy, uml and z. Technical report, http://sdg.lcs.mit.edu/publications.html, 1999.
6. R.D. Maddux. *Relation Algebras*, volume 150 of *Studies in logic*. Elsevier, Iowa, USA, 2006.
7. G. Michels, S. Joosten, J. van der Woude, and S. Joosten. Ampersand: Applying relation algebra in practice. In *Proceedings of the 12th conference on Relational and Algebraic Methods in Computer Science*, Lecture Notes in Computer Science 6663, pages 280–293, Berlin, 2011. Springer-Verlag.
8. J. Rumbaugh, I. Jakobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Mass., 1999.
9. J.G. Sanderson. *A Relational Theory of Computing*, volume 82 of *Lecture Notes in Computer Science*. Springer, 1980.
10. G. Schmidt and T. Ströhlein. *Relationen und Grafen*. Springer-Verlag, 1988.
11. B. Selic. Uml 2: a model-driven development tool. *IBM Systems Journal*, 45(3):607–620, 2006.
12. J.M. Spivey. *The Z Notation: A reference manual*. International Series in Computer Science. Prentice Hall, New York, 2nd edition, 1992.
13. S.D. Swierstra. *Lawine: an Experiment in Language and Machine Design*. PhD thesis, Twente University of Technology, January 1981.
14. A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.
15. J. Woodcock and J. Davies. *Using Z - Specification, Refinement, and Proof*. Prentice Hall, London, 1996.

∀UNITY for Everyone

Ignatius Sri Wishnu Brata Prasetya[1] and *Tanja Ernestina Jozefina Vos*[2]

[1] s.w.b.prasetya@uu.nl
Dept. of Information and Computing Sciences, Utrecht University
Buys Ballot Laboratorium (BBL), Princetonplein 5, 3584 CC Utrecht, Netherlands
[2] tvos@dsic.upv.es
Dept. of Systems, Information and Computation, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain

Dear *Doaitse*,

For many years we have worked together in the area of the formal verification of distributed algorithms. In 1995, Wishnu finished his PhD thesis in this area, paving the way for Tanja who defended hers in 2000. Not only did we successfully verify some exciting algorithms [27, 14], but in the tradition of Software Technology we also strived to improve the underlying infrastructure to make it more reusable, productive, and reliable. Indeed, we were not always successful, but nevertheless we got quite far to be at least modestly proud. We invested effort to verify the inference rules of the UNITY formalism [2]. And to make sure that we did not make any mistake in our proofs, we used the theorem prover HOL [6]. This was a good decision since we did discover (and fix) inconsistencies in hand written proofs. For example with regard to the UNITY's Substitution Axiom [16] and proofs including the Transparency law [28]. But we did more. We extended UNITY with stronger refinement [30] and composition operators [15, 26] and verified these extensions, so that people can safely and more cleverly split their proofs. We have used all these verified rules and operators, to prove the correctness of two classes of distributed algorithms. First, self-stabilizing algorithms that can compute any defined round solvable problem [14]. Second, diffusing computations (or distributed hylomorphisms as we called them due to the structure of their correctness proofs) that perform tasks like termination detection, leader election and propagation of information with feedback [25]. All this has led to cleaner proofs and better representations of the distributed algorithms we have studied, and led to substantial savings on effort during verification work. However, the most important outcome of all this work is the lasting relationship that has resulted from doing two consecutive PhD theses on this topic with you, Doaitse, as a supervisor. Living in different countries and working at different universities, does not prevent us to continue collaborating and almost naturally result in looking for each other when we want to do some fun research (or need any other favour ☺).

The formalism UNITY has been used as a vehicle for facilitating our exploration. To support stronger refinement and composition it was necessary to extend it, by parametrizing the UNITY operators with an invariant and information about which variables the specified properties depend on. For example, as in:

$$\underbrace{Prog_1,\ x > 0,\ \{x, y\}}_{Context\ parameters} \quad \vdash \quad p \ \mathsf{unless}\ q$$

We can see these parameters as context information. In itself, context does not add expressiveness, but it facilitates compositional reasoning. For our own applications these

parameters contained exactly the contextual information that we needed[3], but it is possible that for other applications people may need a different kind of information as context. For example, in some problems certain things can only be done once. If an event a can only be executed once, the behaviour of the rest of the system before and after a executes is typically quite different. Shifting this to the context can facilitate compositional reasoning. Another example is security. When two processes interact, the interaction may critically depend on the security level of each. Again, shifting this aspect to the context may facilitate compositional reasoning.

As exciting as it might sound, we did not actually go into exploring the above examples. We did however work out a generalization of UNITY, and verified it mechanically with HOL. This means that other people can just instantiate it, using any custom context they might need, and get the corresponding UNITY inference rules proven sound for free! This was as far as we went. As the examples above suggest, there are more hills to climb. But who knows; our paths may come together again, in a different time, under different circumstances. It would be fun to try another climb, and see what's beyond those hills.

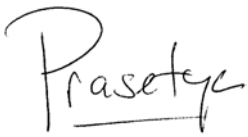
The article below presents ∀UNITY. We will just give you a summary, containing the needed definitions and the verified theorems as results. We deliberately typeset them in the typewrite font, as some kind of nostalgia to the PhD theses [27, 14] we wrote (and the two times wonderful four years dedicated to them).

Although the contribution is modest, we hope you like it. And as usual with our work on UNITY, you do not have to check the proofs, since these have been mechanically verified with HOL ...

Valencia, 3 April 2010

—Tanja—

Houten, 3 April 2010



—Wishnu—

1 Introduction

UNITY is a formalism introduced by Chandy and Misra back in 1988 [2] to model and prove the correctness of distributed systems. The logic is simple, consisting of a small set of temporal operators (unless, ensures, and $\mapsto$ (leads-to)) and inference rules. In terms of LTL, UNITY's temporal operators have an implicit outer $\Box$; other than that, arbitrarily nested temporal properties cannot be expressed. For many applications these are not needed, and otherwise in UNITY one would introduce auxiliary variables to express them. Unlike LTL, the meaning of the operators are defined in terms of the program itself, rather than in terms of its executions. This means that safety properties (unless) and primitive progress

[3] If you recall, they also conveniently solve the aforementioned problem with the Substitution Axiom. The "invariant" parameter is actually required to be "strong". That is, it should hold initially, and is individually preserved by each action in the program (which is a different concept than to be preserved by the program collectively). We constrained the Substitution Axiom to be applied only within the context of such an invariant. This is safe (and we proved it).

(ensures) can always be verified, even if the program has infinitely many states. The original presentation does not include an automatic verification technique for general progress ($\mapsto$), but for finite state programs it is possible to use fix-point iterations to calculate the weakest predicate to guarantee progress to a given goal [8].

UNITY's style of inference rules makes it a nice formalism to prove the correctness of various distributed algorithms [11, 22, 1, 21, 13, 10, 3, 24, 31, 17, 27, 29]. But we also notice that on some occasions people modify/extend the logic to allow them to deal with the problems more effectively [21, 4, 23, 14, 20, 5, 7, 19, 12]. Modifying a logic is risky: we may introduce inconsistency. There have been many examples of such mistakes, for example Sanders showed in 1991 [21] that the Substitution Law in UNITY [2] is actually unsound. It took several more years for people to realize that Sanders's own correction is also flawed [16].

This article presents ∀UNITY; it is a generalization of UNITY. Although it provides the same set of inference rules, these are now derived from a set of more primitive (weaker) rules. From it, a custom UNITY variant can be easily instantiated by only showing that the variant upholds the primitive rules. This is significantly less work than having to redo the entire UNITY soundness proof. ∀UNITY is provided as a library within the HOL theorem prover [6] and all its derived laws have been mechanically verified. HOL ensures that any concrete UNITY variant which is derived from ∀UNITY is sound and complete, in the sense that it will satisfy all standard UNITY inference rules. Effort has been made to make each derived rule minimal, in the sense that it explicitly mentions which primitive rules it requires. Consequently, it is possible to make an instantiation that, for example, only upholds the Completion Rule under certain circumstances.

The ∀UNITY HOL library can be downloaded here: `www.cs.uu.nl/~wishnu`; or else requested from the authors if the URL ceases to exist.

To read the rest of this article, the reader is presumed to be familiar with UNITY; if not, the reader is recommended to read the long version of this article [18] instead.

2 ∀UNITY, Basic Idea

In UNITY, temporal properties are written like this: $P \vdash p \text{ unless } q$, to mean that the behavior p unless q is valid on all executions of the program P. The right hand side of $\vdash$ specifies the intended behavior, and the left side can be seen as the context of behavior. In other UNITY extensions, more information are added to the context, e.g. a global invariant [21], or assumptions on the set of variables on which the behavior depends [15].

To make it general, in ∀UNITY we abstract away from the exact structure of the context; thus, we focus only on the behavior part. Traditionally, UNITY properties are defined in terms of the program. Since we now want to treat programs as a part of the context, and thus abstracted away, we should find another way to define them. In a variant that carries a context C, the relation $(\lambda p, q.\ C \vdash p \text{ unless } q)$ is usually intended to form a subset of the traditional unless . C determines which subset it is, but it cannot be an arbitrary subset either. It needs to keep sufficient properties to, in the end, give us all or some UNITY inference rules. To illustrate this, consider this so-called Post-weakening Rule of the traditional UNITY; it says that the 'escape' part of unless can be weakened:

$$\frac{P \vdash p \text{ unless } q \quad , \quad \models q \Rightarrow r}{P \vdash p \text{ unless } r}$$

In ∀UNITY the rule is represented as a theorem, and generalized to:

```
|- includesIMP implies ∧
   (∀p. inDomain unless p ⇒ inDomain implies p) ∧
   hasProperDomain unless ∧
   satisfiesUNLESS_Subst implies unless ∧
   unless p q ∧ implies q r
   ⇒
   unless p r
```

The last three lines correspond to the original rule as displayed before. But in the above theorem, `implies` and `unless` are actually free variables. So, the theorem quantifies over them, and thus accomodates any custom variants of predicate implication and UNITY's traditional unless, with the first four premises above specifying the properties that the variants must meet in order to give us the original inference rule.

The context C is not explicit in the theorem above; but we do have the expresiveness to express it. Both `implies` and `unless` above are just binary relations over state predicates. That is, they are of type $Pred \rightarrow Pred \rightarrow$`bool`. An `unless` with a context C, which we would usually denote as $C \vdash p$ unless q is then treated as a relation $(\lambda p, q.\ C \vdash p$ unless $q)$; and thus can be instantiated from the theorem.

For each traditional UNITY inference rule, we will give the correspoding generalized rule. As in the example above, each general rule quantifies over `implies`, `unless`, etc, and specifies a set of more primitive properties these variants must satisfy to get us the rule. As above, the rule is represented as a theorem, which means that it has indeed been proven.

2.1 How to read the rest of the article

Subsection 2.2 briefly explains our notation. Section 4 lists the generalized inference rules. The needed definitions are given in Section 3. The reader can optionally just jump to Section 4, and return to Section 3 as needed.

2.2 Notation

We will deviate from the usual UNITY style of notation [2]. ∀UNITY is implemented in the theorem prover HOL [6] as a HOL library. We do not expect people to instantiate it or to use it manually in a pen-and-paper proof, but to use it within HOL, since this is the usage that will maximize its benefit. We will thus use HOL ASCII-style notation to describe ∀UNITY. Admittedly this is less stylish, but it will make it easier for the reader to associate the presentation here to the corresponding HOL theorems in our library.

Formulas are written in the type writer font and UNITY operators are written in the prefix-style, e.g. `unless p q`. A UNITY inference rule will be written like this:

```
|- A1 ∧ ... ∧ An ⇒ C
```

which means that `C` is derivable from `A1` ... `An`. The notation makes an inference rule looks like a theorem. Indeed: in ∀UNITY an inference rule is represented as a HOL theorem, which also means that it is only a rule if its validity has be proven!

There are two levels of logical operators in ∀UNITY. We have the usual:

$$\mathtt{T}, \mathtt{F}, \neg, \vee, \wedge, \forall, \exists$$

These operate on boolean values.

The temporal operators of UNITY operate however on a program's state predicates. In HOL we represent such a predicate as a function `'state → bool`; the exact structure of `'state` can be left unspecified (that is, `'state` can be left polymorphic). So, we have another set of operators:

$$\texttt{TT}, \texttt{FF}, \texttt{NOT}, \texttt{OR}, \texttt{AND}, !!, ??$$

which are just the previously listed boolean operators lifted to the state predicate level. For example, `AND` is defined as:

```
p AND q  =  (λs. p s ∧ q s)
```

Abstractly though, the reader can pretend that both sets of operators are equivalent. If `p` is a state predicate, `valid p` is defined as follows:

```
valid p = ∀s. p s
```

meaning that `p` is predicate that holds for all states in the assumed universe of states.

3 Definitions

Definition 1. We define when a domain `dom` of predicates is closed under the standard predicate operators:

```
|- ∀dom.  isClosed_under_PredOPS dom
           =
           dom TT ∧ (∀p. dom p ⇒ dom (NOT p))
           ∧ (∀p q. dom p ∧ dom q ⇒ dom (p AND q))
           ∧ (∀W. (∀p. W p ⇒ dom p) ⇒ dom (!!p::W. p))
```

Definition 2. We define when a relation `U` subsumes $\models p \Rightarrow q$:

```
|- ∀U.  includesIMP U
          =
          ∀p q. inDomain U p ∧ inDomain U q ∧ valid (p IMP q) ⇒ U p q
```

Definition 3. Below we want to define when `p` is a member of the domain of a relation `U`, where `U` is a UNITY operator. Since reflexivity is a desired property of all UNITY operators, we define this domain membership as follows:

```
|- ∀U p. inDomain U p = U p p
```

And consequently, we define when `U` is proper with respect to this reflexivity:

```
|- ∀U.  hasProperDomain U
          =
          ∀p q. U p q ⇒ inDomain U p ∧ inDomain U q
```

Definition 4.

```
|- ∀U V. isSubRelationOf U V = ∀x y. U x y ⇒ V x y
```

Definition 5. We define when a relation is anti-reflexive (we do not mean to say that it is therefore not reflexive):

```
|- ∀unless.  satisfiesUNLESS_AntiRefl unless
              =
              ∀p. inDomain unless p ⇒ unless p (NOT p)
```

Notice that the definition quantifies over the specific `unless` used. UNITY's traditional unless is anti-reflexive. □

Definition 6. We define when a relation `unless` is conjunctive; again notice that the definition quantifies over the specific `unless` used:

```
|- ∀unless.  satisfiesUNLESS_Conj unless
             =
             ∀p1 q1 p2 q2.
                 unless p1 q1 ∧ unless p2 q2
                 ⇒
                 unless (p1 AND p2) (q1 AND p2 OR q2 AND p1 OR q1 AND q2)
```

Similarly, we define when it is disjunctive:

```
|- ∀unless.  satisfiesUNLESS_Disj unless
      =
      ∀p1 q1 p2 q2.
          unless p1 q1 ∧ unless p2 q2
          ⇒
          unless (p1 OR p2) (q1 AND NOT p2 OR q2 AND NOT p1 OR q1 AND q2)
```

Definition 7. We define when `unless` satisfies UNITY's Substitution Axiom:

```
|- ∀implies unless.  satisfiesUNLESS_Subst implies unless
       =
       ∀p q a b.
          unless p q ∧ implies p a ∧ implies a p ∧ implies q b
          ⇒
          unless a b
```

Definition 8. We define when a given relation is left-disjunctive:

```
|- ∀leadsto.  isLeftDisj leadsto
       =
       ∀W q. (∃p. W p) ∧  (∀p. W p ⇒ leadsto p q)
              ⇒
              leadsto (??p::W. p) q
```

Definition 9. We define the smallest transitive and left-disjunctive closure of `ensures`; notice that we quantify over the specific `ensures` used:

```
|- ∀ensures p q.  GLEADSTO ensures p q
       =
       ∀U. isSubRelationOf ensures U ∧ isTransitive U ∧ isLeftDisj U
           ⇒
           U p q
```

Definition 10. We define when progress and `unless` satisfy UNITY's Progress Safety Progress (PSP) rule:

```
|- ∀progress unless.  satisfiesPSP progress unless
       =
       ∀p q a b. progress p q ∧ unless a b
                 ⇒
                 progress (p AND a) (q AND a OR b)
```

4 ∀UNITY Inference Rules

In the theorems below, identifiers like `p`, `q`, `implies`, `unless`, and `ensures` are free; so, they are implicitly universally quantified.

Theorem 11 : Unless Reflexivity

```
|- inDomain unless p ⇒ unless p p
```

Theorem 12 : Unless, Implies Lifting

```
|- includesIMP implies ∧ isSubRelationOf implies unless ∧
   inDomain implies p  ∧ inDomain implies q ∧
   valid (p IMP q)
   ⇒
   unless p q
```

Theorem 13 : Unless, Post-condition Weakening

```
|- includesIMP implies ∧
   (∀p. inDomain unless p ⇒ inDomain implies p) ∧
   hasProperDomain unless ∧ satisfiesUNLESS_Subst implies unless ∧
   unless p q ∧ implies q r
   ⇒
   unless p r
```

Theorem 14 : Unless, Simple Conjunctivity

```
|- includesIMP implies ∧ isClosed_under_PredOPS (inDomain implies) ∧
   (∀p. inDomain unless p ⇒ inDomain implies p) ∧
   hasProperDomain unless ∧ satisfiesUNLESS_Subst implies unless ∧
   satisfiesUNLESS_Conj unless ∧
   unless p1 q1 ∧ unless p2 q2
   ⇒
   unless (p1 AND p2) (q1 OR q2)
```

Theorem 15 : Unless, Simple Disjunctivity

```
|- includesIMP implies ∧ isClosed_under_PredOPS (inDomain implies) ∧
   (∀p. inDomain unless p ⇒ inDomain implies p) ∧
   hasProperDomain unless ∧ satisfiesUNLESS_Subst implies unless ∧
   satisfiesUNLESS_Disj unless ∧
   unless p1 q1 ∧ unless p2 q2
   ⇒
   unless (p1 OR p2) (q1 OR q2)
```

Theorem 16 : Leadsto, Implies Lifting

```
|- includesIMP implies ∧ isSubRelationOf implies ensures ∧
   inDomain implies p ∧ inDomain implies q ∧
   valid (p IMP q)
   ⇒
   GLEADSTO ensures p q
```

Theorem 17 : Leadsto, Ensures Lifting

```
|- ensures p q ⇒ GLEADSTO ensures p q
```

Theorem 18 : Leadsto Transitivity

```
|- GLEADSTO ensures p q ∧ GLEADSTO ensures q r ⇒ GLEADSTO ensures p r
```

Theorem 19 : Leadsto Left-Disjunctivity

```
|- W i ∧ (∀i. W i ⇒ GLEADSTO ensures (f i) q)
   ⇒
   GLEADSTO ensures (??i::W. f i) q
```

The `W i` condition is equivalent with (∃`i`. `W i`); hence, requiring `W` to be non-empty. If `W` is empty, `(??i::W. f i)` is then equivalent to `FF`. In classical UNITY, `FF` ↦ `q` is technically a valid property of any program. ∀UNITY only allows `GLEADSTO ensures FF q` to be inferred if `q` is actually in the domain of `ensures`. See also the `implies` lifting rule. Keeping the expressions confined inside their domain is important to keep the behavior part consistent with the context. □

Theorem 20 : Leadsto Simple Disjunction

```
|- includesIMP implies ∧
   isClosed_under_PredOPS (inDomain implies) ∧
   isSubRelationOf implies ensures ∧
   hasProperDomain ensures ∧
   (∀p. inDomain ensures p ⇒ inDomain implies p) ∧
   GLEADSTO ensures p q ∧ GLEADSTO ensures r s
   ⇒
   GLEADSTO ensures (p OR r) (q OR s)
```

Theorem 21 : Leadsto, Substitution Rule

```
|- isSubRelationOf implies ensures ∧
   implies a p ∧ implies q b ∧
   GLEADSTO ensures p q
   ⇒
   GLEADSTO ensures a b
```

Theorem 22 : Cancellation Rule

```
|- includesIMP implies ∧
   isClosed_under_PredOPS (inDomain implies) ∧
   isSubRelationOf implies ensures ∧ hasProperDomain ensures ∧
   (∀p. inDomain ensures p ⇒ inDomain implies p) ∧
   inDomain (GLEADSTO ensures) q ∧
   GLEADSTO ensures p (q OR r) ∧ GLEADSTO ensures r s
   ⇒
   GLEADSTO ensures p (q OR s)
```

Theorem 23 : Bounded Progress Rule

```
|- includesIMP implies ∧
   isClosed_under_PredOPS (inDomain implies) ∧
   isSubRelationOf implies ensures ∧ hasProperDomain ensures ∧
   (∀p. inDomain ensures p ⇒ inDomain implies p) ∧
   ADMIT_WF_INDUCTION LESS ∧
   inDomain (GLEADSTO ensures) q ∧
   (∀m. GLEADSTO ensures
                (p AND (λs. M s = m))
                (p AND (λs. LESS (M s) m) OR q))
   ⇒
   GLEADSTO ensures p q
```

`ADMIT_WF_INDUCTION LESS` above means that `LESS` is a relation that admits a well founded induction. □

Theorem 24 : Progress Safety Progres (PSP) Rule

```
|- includesIMP implies                                ∧
   isClosed_under_PredOPS (inDomain implies)          ∧
   hasProperDomain unless                             ∧
```

```
(∀p. inDomain unless p ⇒ inDomain implies p)        ∧
hasProperDomain ensures                             ∧
isSubRelationOf implies ensures                     ∧
(∀p. inDomain ensures p ⇒ inDomain implies p)       ∧
satisfiesPSP ensures unless                         ∧
GLEADSTO ensures p q                                ∧
unless a b
⇒
GLEADSTO ensures (p AND a) (q AND a OR b)
```

Progress is usually disjunctive but in general it is not conjunctive. For example, if a program P can progress from p to q and from p to r we do not know if it can progress to a state where both q and r hold. However, if we know that, for example, both q and r are stable predicates, then we know that the conjunction of them will hold eventually. This kind of property of progress is often very useful, e.g. as in [9, 17]. In UNITY, this is captured by the so-called Completion Rule. Before we present ∀UNITY's version of the rule, let us define a derived UNITY operator that specifies progress from p to some q, and where q will then either remains stable or at some point the program escapes to a:

Definition 25.

```
|- ∀progress unless p q b.
    COMPLETES leadsto unless p q b = leadsto p q ∧ unless q b
```

Such a progress can be composed conjunctively, using UNITY's Completion Rule, which now looks like this in ∀UNITY:

Theorem 26 : Completion Rule

```
|- includesIMP implies                                ∧
   (∀p. inDomain unless p ⇒ inDomain implies p)       ∧
   isClosed_under_PredOPS (inDomain implies)          ∧
   hasProperDomain unless                             ∧
   isSubRelationOf implies unless                     ∧
   isSubRelationOf implies ensures                    ∧
   satisfiesUNLESS_Subst implies unless               ∧
   satisfiesUNLESS_Conj unless                        ∧
   satisfiesUNLESS_Disj unless                        ∧
   satisfiesPSP ensures unless                        ∧
   hasProperDomain ensures                            ∧
   (∀p. inDomain ensures p ⇒ inDomain implies p)      ∧
   isSubRelationOf ensures unless                     ∧
   satisfiesPSP ensures unless                        ∧
   COMPLETES (GLEADSTO ensures) unless p q b          ∧
   COMPLETES (GLEADSTO ensures) unless r s b
   ⇒
   COMPLETES (GLEADSTO ensures) unless (p AND r) (q AND s OR b) b
```

All the above inference rules allow us to infer new temporal property from others, but because the corresponding variables representing the used temporal operators are bound to the same name, effectively this means that they are all parameterized by the same context.

So far, we have no rule to infer a property about the used context, nor a rule to allow us to change to a different context.

We introduce first the following two operators. The first simply formalizes concept of inferring properties of the context; the second expresses the concept of extending the concept conservatively. In those definitions, `unityOp C p q` represents $C \vdash p$ unityOp q, where unityOp is a variant of some UNITY operator; e.g. the variant of unless , and C is some context. Note that `unityOp C` is therefore a relation over state predicates.

Definition 27.

```
|- ∀unityOp property.  implicitlyImplies unityOp property
                       =
                       ∀C p q. unityOp C p q ⇒ property C
```

Definition 28.

```
|- ∀unityOp extend.  isConservative unityOp extend
                     =
                     ∀C p q. unityOp C p q ⇒ unityOp (extend C) p q
```

It is then quite trivial to obtain the following theorems:

Theorem 29 : Implicitly Implies Rule

```
|- implicitlyImplies unityOp property ∧ unityOp C p q  ⇒  property C
```

Theorem 30 : Conservative Context Change

```
|- isConservative unityOp extend  ∧ unityOp C p q
   ⇒
   unityOp (extend C) p q
```

When `unityOp` is leads-to then we also have the following stronger theorems:

Theorem 31 :

```
|- implicitlyImplies ensures_ property ∧ GLEADSTO (ensures_ C) p q
   ⇒
   property C
```

Theorem 32 :

```
|- isConservative ensures_ extend  ∧ GLEADSTO (ensures_ C) p q
   ⇒
   GLEADSTO (ensures_ (extend C)) p q
```

5 Conclusion

∀UNITYis a generalization of UNITY. It provides the same set of inference rules as the standard UNITY but it does not impose any concrete interpretation of what `unless` and `ensures` are. Instead, it gives a set of quite weak primitive properties that abstractly model a minimalistic requirement to obtain a UNITY-like logic. A user can create a custom variant of UNITY by providing a concrete definition of `unless` and `ensures` and then showing that they satisfy the primitive properties.

∀UNITYis provided as a HOL (a theorem prover) library. Creating a UNITY variant using ∀UNITYhas the following advantages:

1. A rich set of standard inference rules have already proven; this saves a lot of work.
2. The rules have been proven *mechanically* in HOL, so they are very safe to use.
3. The user automatically gets all theorem proving support of HOL.

References

1. I. Chakravarty, M. Kleyn, T.Y.C. Woo, R. Bagrodia, and V. Austel. UNITY to UC: A case study in the derivation of parallel programs. *Lecture Notes of Computer Science*, 574:7–20, 1991.
2. K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley Publishing Company, Inc., 1988.
3. C. Christian and R. Gruia-Catalin. Formal specification and design of a message router. *ACM Transactions on Software Engineering and Methodology*, 3(4):271–307, October 1994.
4. P. Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23:107–125, December 1994.
5. P. Collette and E. Knapp. Logical foundations for compositional verification and development of concurrent programs in UNITY. *Lecture Notes in Computer Science*, 936:353–??, 1995.
6. Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.
7. R. Gruia-Catalin and P. J. McCann. An introduction to mobile UNITY. *Lecture Notes in Computer Science*, 1388:871–880, 1998.
8. Knapp. A predicate transformer for progress. *IPL: Information Processing Letters*, 33, 1990.
9. P.J.A. Lentfert. *Distributed Hierarchical Algorithms.* PhD thesis, Utrecht University, April 1993.
10. P.J.A. Lentfert and S.D. Swierstra. Distributed maximum maintenance on hierarchically divided graphs. *Formal Aspects of Computing*, 5(1):21–60, 1993.
11. Zhiming Liu. Modelling checkpointing and recovery within UNITY. Research Report CS-RR-145, Department of Computer Science, University of Warwick, Coventry, UK, August 1989.
12. J. Misra. *A Discipline of Multiprogramming.* Springer-Verlag, 2001.
13. A. Pizzarello. An indrustial experience in the use of UNITY. *Lecture Notes of Computer Science*, 574:38–49, 1991.
14. I. S. W. B. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms.* PhD thesis, Inst. of Information and Comp. Science, Utrecht Univ., 1995. Download: `www.cs.uu.nl/library/docs/theses.html`.
15. I.S.W.B. Prasetya. Variable access constraints and compositionality of liveness properties. In H.A. Wijshoff, editor, *Proceeding of Computing Science in the Netherlands 94*, pages 12–23. SION, Stichting Matematisch Centrum, 1993.
16. I.S.W.B. Prasetya. Error in the UNITY substitution rule for subscripted operators. *Formal Aspects of Computing*, 6:466–470, 1994.
17. I.S.W.B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. *Lecture Notes in Computer Science*, 1217:399 – 415, 1997.

18. I.S.W.B. Prasetya, T.E.J. Vos, A. Azurat, and S.D. Swierstra. !UNITY: A theory of general UNITY. Technical Report UU-CS-2002-025, Inst. of Information and Computer Sci., Utrecht University, 2002. Downloadable from `www.cs.uu.nl`.
19. I.S.W.B. Prasetya, T.E.J. Vos, S.D. Swierstra, and B. Widjaja. A theory for composing distributed components based on temporary interface. In *Proceedings of the Workshop on Formal Aspects of Component Software (FACS) 2003*, 2003.
20. J.R. Rao. *Extensions of the UNITY methodology: compositionality, fairness, and probability in parallelism*, volume 908 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1995.
21. B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.
22. M.G. Staskaukas. The formal specification and design of a distributed electronic funds-transfer system. *IEEE Trans. on Computers*, 37(12):1515–1528, December 1988.
23. R.T. Udink. *Program Refinement in UNITY-like Environments*. PhD thesis, Inst. of Information and Computer Sci., Utrecht University, 1995. Downloadable from `www.cs.uu.nl`.
24. R.T. Udink and J. N. Kok. The RPC-Memory specification problem: UNITY + refinement calculus. *Lecture Notes in Computer Science*, 1169:521–??, 1996.
25. T.E. J. Vos and S.D. Swierstra. Facilitating the verification of diffusing computations and their applications. *CLEI Electron. J.*, 8(1), 2005.
26. T.E.J. Vos. *Sequential program composition in UNITY*. Technical report. University of Cambridge, Computer Laboratory, 2000.
27. T.E.J. Vos. *UNITY in Diversity: A Stratified Approach to the Verification of Distributed Algorithms*. PhD thesis, Inst. of Information and Computer Sci., Utrecht University, 2000. Download: `www.cs.uu.nl`.
28. T.E.J. Vos and S.D. Swierstra. Make your enenemies transparent. In Rolf H. Mohring, editor, *Graph-Theoretic Concepts in Computer Science*, volume LNCS 1335, pages 342–353, 1997.
29. T.E.J. Vos and S.D. Swierstra. Proving distributed hylomorphisms. Technical Report UU-CS-2001-40, Inst. of Information and Comp. Science, Utrecht University, 2001. Download: `www.cs.uu.nl`.
30. T.E.J. Vos and S.D. Swierstra. Yet another program refinement relation. In *International Workshop on Refinement of Critical Systems: Methods, Tools and Experience, (RCS'02)*, pages 12–23, 2002.
31. T.E.J. Vos, S.D. Swierstra, and I.S.W.B. Prasetya. Formal methods and mechanical verification applied to the development of a convergent distributed sorting program. Technical Report UU-CS-1996-37, Inst. of Information and Computer Sci., Utrecht University, 1996. downloadable from `www.cs.uu.nl`.

Squiggoling with Bialgebras

Recursion Schemes from Comonads Revisited

Ralf Hinze and Nicolas Wu*

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
{ralf.hinze,nicolas.wu}@cs.ox.ac.uk

1 Introduction

A broad class of algorithms fall within the framework of dynamic programming, where the results of smaller parts of a problem are used to build efficient solutions of the whole. Such algorithms are an essential tool in the repertoire of every skilled programmer. A classic example is the *unbounded knapsack* problem, which so neatly demonstrates how a greedy algorithm can be implemented efficiently using dynamic programming.

We start with a specification of the problem. There was once a professor who had finally reached his time to retire. His final task, before going on indefinite gardening leave, was to bring home the books and papers from his office which he treasured the most. Unfortunately, he was unable to bring them all, and could only carry a fixed weight of capacity c in his knapsack. There was no end to the number of documents he could have chosen from in his office, and so the best he could manage was to categorise them into groups, where each group i contained items whose weight and value were w_i and v_i. He was interested, of course, in computing the maximum value that would fit into his knapsack. Thus, given capacity 15 and a list of groups *wvs* with

$$\begin{aligned} &\mathit{wvs} :: [(\mathbb{N}, \mathit{Value})] \\ &\mathit{wvs} = [(12,4),(1,2),(2,2),(1,1),(4,10)] \ , \end{aligned}$$

the maximum value possible is 36, using three items each from the second and fifth groups. How would he proceed to efficiently choose what to take with him?

The naive recursive solution of the problem takes exponential time, since each intermediate level of the recursion spawns further recursive calls, and no common results are shared:

$$\begin{aligned} &\mathit{knapsack}_1 :: \mathbb{N} \to \mathit{Value} \\ &\mathit{knapsack}_1\, 0 \qquad\quad = 0 \\ &\mathit{knapsack}_1\, (c+1) = \\ &\quad \mathit{maximum}'\, [v + \mathit{knapsack}_1\, (c-w) \mid (w+1, v) \leftarrow \mathit{wvs}, w \leqslant c] \ . \end{aligned}$$

We suppose here that the maximum value of the empty list of candidate solutions is zero, and so let $\mathit{maximum}'\, [\,] = 0$ and be the maximum of the list otherwise.

This example lends itself perfectly to a solution that uses dynamic programming: each recursive step can naturally be thought of as a subproblem whose result can be reused time and again. The translation into an efficient version that uses an immutable Array datatype,

* This work has been funded by EPSRC grant number EP/J010995/1.

that is dynamically constructed to store the results, is a fairly routine exercise:

$$
\begin{array}{l}
\mathit{knapsack}_2 :: \mathbb{N} \to \mathit{Value} \\
\mathit{knapsack}_2\ c = \mathit{table}\ !\ c \\
\quad \textbf{where} \\
\qquad \mathit{table}\ ::\ \mathsf{Array}\,\mathbb{N}\ \mathit{Value} \\
\qquad \mathit{table} = \mathit{array}\,(0,c)\,[(i, \mathit{ks}\ i) \mid i \leftarrow [0\,..\,c]] \\
\qquad \mathit{ks} :: \mathbb{N} \to \mathit{Value} \\
\qquad \mathit{ks}\ i = \ \mathit{maximum}'\,[v + \mathit{table}\ !\ (i-w) \mid (w,v) \leftarrow \mathit{wvs}, w \leqslant i]\ .
\end{array}
$$

The improvement in performance is dramatic, resulting in a pseudo-polynomial time algorithm. The key to this efficiency lies in the fact that the array *table* allows constant time indexing of results that are reused in different recursive calls of the function.

However, despite the performance gains, the solution we have arrived at remains unsatisfactory. A fundamental problem with both of the implementations we have seen is that they rely on general recursion, and it has long been understood that general recursion is the 'goto' of functional programming. Of course, what we desire is a version that might be expressed as an instance of a recursion scheme that holds the promise that it terminates, and has the efficiency we expect from this algorithm.

Where do we turn to for a squiggoly answer to this problem? One approach is to use recursion schemes from comonads [1], in particular, *histomorphisms*, which are the squiggol rendering of course-of-value recursion:

Recursion schemes from comonads form a general recursion principle that makes use of a comonad $(\mathsf{N}, \epsilon, \delta)$, to provide 'contextual information' to the body of the recursion. The scheme is 'doubly generic': it is parametric in a datatype $\mu\mathsf{F}$, and in the comonad N. Histomorphisms are a particularly nice instance of the recursion scheme, where a *cofree comonad* is used to make the results of recursive calls on *all* subterms available at each iteration.

More formally, recursion schemes from comonads make use of a coalgebra $\mathit{fan} : \mu\mathsf{F} \to \mathsf{N}\,(\mu\mathsf{F})$ that embeds a subterm in a context. The coalgebra can be defined generically in terms of a distributive law $\lambda : \mathsf{F} \circ \mathsf{N} \stackrel{\cdot}{\to} \mathsf{N} \circ \mathsf{F}$, which is subject to certain conditions (3), detailed below. Here is the scheme in its full glory:

Let $\lambda : \mathsf{F} \circ \mathsf{N} \stackrel{\cdot}{\to} \mathsf{N} \circ \mathsf{F}$ be a distributive law, and $\mathit{fan} = (\!|\mathsf{N}\,\mathit{in} \cdot \lambda\,(\mu\mathsf{F})|\!)$. For any $(\mathsf{F} \circ \mathsf{N})$-algebra (B, b) there is a unique arrow $f : \mu\mathsf{F} \to B$ such that

$$
\begin{array}{ccc}
\mathsf{F}\,(\mu\mathsf{F}) & \xrightarrow{\mathsf{F}\,(\mathsf{N}\,f \cdot \mathit{fan})} & \mathsf{F}\,(\mathsf{N}\,B) \\
{\scriptstyle \mathit{in}}\downarrow & & \downarrow{\scriptstyle b} \\
\mu\mathsf{F} & \xrightarrow[f]{} & B
\end{array}
\quad . \tag{1}
$$

The composition $\mathsf{N}\,f \cdot \mathit{fan}$ creates a context that makes the results of 'recursive calls' available to the algebra b. Note that b is a context-sensitive algebra—an $(\mathsf{F} \circ \mathsf{N})$-algebra, rather than merely an F-algebra.

The recursion scheme is quite amazing in its generality: it works for an arbitrary functor F and an arbitrary comonad N, as long as they can be related by a distributive law. Now, the goal of this paper is to establish the correctness of the scheme, deriving the unique solution of (1) in the process. To this end we shall need quite a bit of machinery, which we shall introduce in the subsequent sections. But first let us revisit our introductory example.

To phrase the knapsack problem as an instance of the scheme we need to identify the initial algebra $(\mu\mathsf{F}, \mathit{in})$ and the comonad N. The first is easy: the type of natural numbers $\mathbb{N}$

is the initial algebra of the functor $\mathsf{Nat}\,A = 1 + A$. The second choice is specific to the class of algorithms: histomorphisms rely on the cofree comonad, in our example, on the cofree comonad for the base functor Nat, which we denote Nat_∞. We will go into more detail in the next section, but for now, Nat_∞ can be understood as the type of nonempty lists, which for our purposes, are treated as simple look-up tables. In particular, it supports a function $\mathit{lookup} :: \mathsf{Nat}_\infty\ v \to \mathbb{N} \to \mathit{Maybe}\ v$ that acts as a means of indexing values that have already been calculated.

$$
\begin{aligned}
&\mathit{knapsack}_3 :: \mathbb{N} \to \mathit{Value}\\
&\mathit{knapsack}_3 = \mathit{knap} \cdot \mathit{fmap}\,(\mathit{fmap}\,\mathit{knapsack}_3 \cdot \mathit{fan}) \cdot \mathit{in}^\circ\\
&\mathit{knap} :: \mathsf{Nat}\,(\mathsf{Nat}_\infty\ \mathit{Value}) \to \mathit{Value}\\
&\mathit{knap}\,(\mathit{Zero}) \qquad = 0\\
&\mathit{knap}\,(\mathit{Succ}\ \mathit{table}) =\\
&\quad \mathit{maximum}'\,[v_1 + v_2 \mid (w+1, v_1) \leftarrow \mathit{wvs}, \mathit{Just}\ v_2 \leftarrow [\mathit{lookup}\ \mathit{table}\ w]]
\end{aligned}
$$

The helper function *knap* plays the part of the context-sensitive algebra. Note that $\mathit{lookup}\ \mathit{table}\ w$ corresponds to the operation $\mathit{table}!(i-w)$ of the array-based implementation. In other words, the look-up table of type $\mathsf{Nat}_\infty\ \mathit{Value}$ stores the values in reverse order.

2 Background: Comonad and Distributive Law

Comonad. Functional programmers have embraced monads, and to a lesser extent, comonads, to capture effectful and context-sensitive computations. We use comonads to model 'recursive calls in context'. A comonad is a functor $\mathsf{N} : \mathscr{C} \to \mathscr{C}$ equipped with a natural transformation $\epsilon : \mathsf{N} \dot{\to} \mathsf{Id}$ (counit), that extracts a value from a context, and a second natural transformation $\delta : \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{N}$ (comultiplication), that duplicates a context, such that the following laws hold:

$$\epsilon \circ \mathsf{N} \cdot \delta = \mathsf{N}\ , \tag{2a}$$

$$\mathsf{N} \circ \epsilon \cdot \delta = \mathsf{N}\ , \tag{2b}$$

$$\delta \circ \mathsf{N} \cdot \delta = \mathsf{N} \circ \delta \cdot \delta\ . \tag{2c}$$

Here we use categorical notation, where natural transformations can be composed horizontally ($\circ$), and vertically ($\cdot$), and the identity natural transformation for a functor is denoted by the functor itself. The first two properties, the counit laws, state that duplicating a context and then discarding a duplicate is the same as doing nothing. The third property, the coassociative law, equates the two ways of duplicating a context twice.

In Haskell, we can capture the interface to comonads by using the following type class, where *extract* corresponds to ϵ, and *duplicate* to δ.

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Comonad}\ n\ \mathbf{where}\\
&\quad \mathit{extract}\ \ :: n\ a \to a\\
&\quad \mathit{duplicate} :: n\ a \to n\,(n\ a)\ .
\end{aligned}
$$

We have already noted that histomorphisms employ the so-called *cofree comonad* of a functor F. As Haskell supports higher-kinded datatypes, the functor part of the comonad can be readily implemented as follows.

$$
\begin{aligned}
&\mathbf{data}\ f_\infty\ a = \mathit{Cons}\,\{\mathit{head} :: a, \mathit{tail} :: f\,(f_\infty\ a)\}\\
&\mathbf{instance}\,(\mathit{Functor}\ f) \Rightarrow \mathit{Functor}\,(f_\infty)\ \mathbf{where}\\
&\quad \mathit{fmap}\ f\,(\mathit{Cons}\ a\ \mathit{ts}) = \mathit{Cons}\,(f\ a)\,(\mathit{fmap}\,(\mathit{fmap}\ f)\ \mathit{ts})
\end{aligned}
$$

The type f_∞ can be seen as the type of generalised streams of observations—'generalised' because the 'tail' is a F-structure of 'streams' rather than just a single one. A generalised stream is, in fact, very similar to a generalised rose tree, except that the latter is usually seen as an element of an inductive type, whereas this construction is patently coinductive.

A cofree value can be built by coiteration from some seed value, where a given function *hd* is used to produce a value from a seed, and *tl* produces the seeds in the next level of coiteration:

$$\begin{aligned}&\mathit{coiterate} :: (\mathit{Functor}\, f) \Rightarrow (a \to b) \to (a \to f\ a) \to (a \to f_\infty\ b)\\&\mathit{coiterate}\ \mathit{hd}\ \mathit{tl}\ x = \mathit{Cons}\ (\mathit{hd}\ x)\ (\mathit{fmap}\ (\mathit{coiterate}\ \mathit{hd}\ \mathit{tl})\ (\mathit{tl}\ x))\ \ .\end{aligned}$$

The function $h = \mathit{coiterate}\, f\ c$ enjoys a universal property: it is the unique F-coalgebra homomorphism $h : (A, c) \to (\mathsf{F}_\infty\ B, \mathit{tail}\ B)$ with $f = \mathit{head}\ B \cdot h$. Together with the destructors of the datatype, *coiterate* can be used to produce an instance of the *Comonad* class:

$$\begin{aligned}&\mathbf{instance}\ (\mathit{Functor}\, f) \Rightarrow \mathit{Comonad}\ (f_\infty)\ \mathbf{where}\\&\quad \mathit{extract}\ \ \ = \mathit{head}\\&\quad \mathit{duplicate} = \mathit{coiterate}\ \mathit{id}\ \mathit{tail}\ \ .\end{aligned}$$

If we instantiate the base functor of the cofree comonad to Id, we obtain the type of streams. A more interesting base functor is

$$\begin{aligned}&\mathbf{data}\ \mathsf{Nat}\ a = \mathit{Zero} \mid \mathit{Succ}\ a\\&\mathbf{instance}\ \mathit{Functor}\ \mathsf{Nat}\ \mathbf{where}\\&\quad \mathit{fmap}\ f\ \mathit{Zero}\ \ \ \ \ \ = \mathit{Zero}\\&\quad \mathit{fmap}\ f\ (\mathit{Succ}\ n) = \mathit{Succ}\ (f\ n)\ \ ,\end{aligned}$$

which gives rise to the type Nat_∞ of non-empty colists. Colists support the indexing operation that was already used in the definition of *knap*.

$$\begin{aligned}&\mathit{lookup} :: \mathsf{Nat}_\infty\ v \to \mathbb{N} \to \mathit{Maybe}\ v\\&\mathit{lookup}\ (\mathit{Cons}\ a\ \_)\ \ \ \ \ \ \ 0\ \ \ \ \ \ \ = \mathit{Just}\ a\\&\mathit{lookup}\ (\mathit{Cons}\ a\ (\mathit{Zero}))\ \ (n+1) = \mathit{Nothing}\\&\mathit{lookup}\ (\mathit{Cons}\ a\ (\mathit{Succ}\ t))\ (n+1) = \mathit{lookup}\ t\ n\end{aligned}$$

Distributive law. A distributive law $\lambda : \mathsf{F} \circ \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{F}$ of an endofunctor F over a comonad N is a natural transformation satisfying the two coherence conditions:

$$\epsilon \circ \mathsf{F} \cdot \lambda = \mathsf{F} \circ \epsilon\ \ , \tag{3a}$$

$$\delta \circ \mathsf{F} \cdot \lambda = \mathsf{N} \circ \lambda \cdot \lambda \circ \mathsf{N} \cdot \mathsf{F} \circ \delta\ \ . \tag{3b}$$

The function *coiterate* that we saw earlier can be used to create a generic distributive law for the cofree comonad. (The proof that this is, in fact, a distributive law of an endofunctor over a comonad is beyond the scope of this paper). We have $\lambda = \mathit{coiterate}\ (\mathsf{F}\ \mathit{head})\ (\mathsf{F}\ \mathit{tail})$, which is implemented as:

$$\begin{aligned}&\mathit{dist} :: (\mathit{Functor}\, f) \Rightarrow f\ (f_\infty\ a) \to f_\infty\ (f\ a)\\&\mathit{dist} = \mathit{coiterate}\ (\mathit{fmap}\ \mathit{head})\ (\mathit{fmap}\ \mathit{tail})\ \ .\end{aligned}$$

The coalgebra *fan*, which generates the stream of all subterms, enjoys a generic definition in terms of *dist*. Below we have specialised its type to the initial algebra $\mathbb{N}$.

$$\begin{aligned}&\mathit{fan} :: \mathbb{N} \to \mathsf{Nat}_\infty\ \mathbb{N}\\&\mathit{fan} = (\!|\mathit{fmap}\ \mathit{in} \cdot \mathit{dist}|\!)\end{aligned}$$

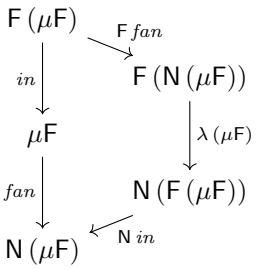
As an example, the call $fan\,3$ generates the colist

$$Cons\,3\,(Succ\,(Cons\,2\,(Succ\,(Cons\,1\,(Succ\,(Cons\,0\,Zero))))))\ .$$

This corresponds to the list of all predecessors.

3 Bialgebra

The recursion scheme involves both algebras and coalgebras, and combines them in an interesting way. We have noted above that *fan* is a coalgebra, but it is actually a bit more: it is a coalgebra *for the comonad* N. Furthermore, the algebra *in* and the coalgebra *fan* go hand-in-hand. They are related by the distributive law λ and form what is known as a λ-bialgebra, a combination of an algebra and a coalgebra with a common carrier. In particular, *in* and *fan* satisfy the so-called pentagonal law.



(4)

The diagram commutes simply because the coalgebra $fan = (\!|\mathsf{N}\,in \cdot \lambda\,(\mu\mathsf{F})|\!)$ is an F-homomorphism of type $(\mu\mathsf{F}, in) \to (\mathsf{N}\,(\mu\mathsf{F}), \mathsf{N}\,in \cdot \lambda\,(\mu\mathsf{F}))$, more about this shortly.

Bialgebras come in many flavours; we need the variant that combines F-algebras and coalgebras for a comonad N. The two functors have to interact coherently, described by the distributive law $\lambda : \mathsf{F} \circ \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{F}$.

Background: Coalgebra for a comonad. A coalgebra for a comonad N is an N-coalgebra (C, c) that respects ϵ and δ:

$$\epsilon\,C \cdot c = id_C\ , \tag{5a}$$

$$\delta\,C \cdot c = \mathsf{N}\,c \cdot c\ . \tag{5b}$$

If we first create a context and then focus, we obtain the original value. Creating a nested context is the same as first creating a context and then duplicating it. For example, the coalgebra $(\mathsf{N}\,A, \delta\,A)$ is respectful—this is the so-called *cofree coalgebra* for the comonad N. The coherence conditions, (5a) and (5b), are just two of the comonad laws, (2a) and (2c). Coalgebras that respect ϵ and δ and N-coalgebra homomorphisms form a category, known as the *(co)-Eilenberg-Moore category* and denoted $\mathscr{C}_{\mathsf{N}}$.

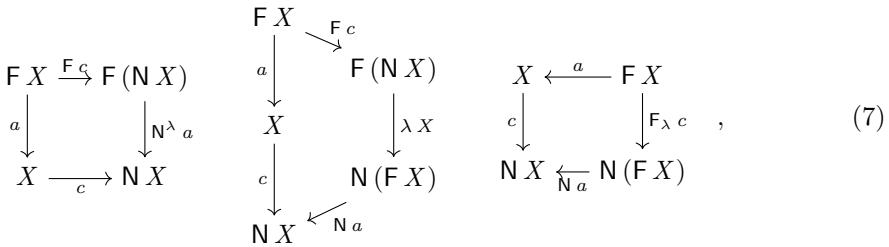
The second law (5b) also enjoys an alternative reading: c is an N-coalgebra homomorphism of type $(C, c) \to (\mathsf{N}\,C, \delta\,C)$. This observation is at the heart of the Eilenberg-Moore construction, see Section 4.

Background: Bialgebra. Let $\lambda : \mathsf{F} \circ \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{F}$ be a distributive law for the endofunctor F over the comonad N. A *λ-bialgebra* (X, a, c) consists of an F-algebra a and a coalgebra c for the comonad N such that the *pentagonal law* holds:

$$c \cdot a = \mathsf{N}\,a \cdot \lambda\,X \cdot \mathsf{F}\,c\ . \tag{6}$$

Loosely speaking, this law allows us to swap the algebra a and the coalgebra c. A λ-bialgebra homomorphism is both an F-algebra and an N-coalgebra homomorphism. λ-bialgebras and their homomorphisms form a category.

The pentagonal law (6) also has two asymmetric renderings



(7)

which relate it to so-called liftings and coliftings, which we introduce next.

Background: Lifting and colifting. A functor $\bar{\mathsf{H}} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathsf{G}\text{-}\mathbf{Alg}(\mathscr{D})$ is called a *lifting* of $\mathsf{H} : \mathscr{C} \to \mathscr{D}$ iff $\mathsf{H} \circ \mathsf{U}^{\mathsf{F}} = \mathsf{U}^{\mathsf{G}} \circ \bar{\mathsf{H}}$, where $\mathsf{U}^{\mathsf{F}} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathscr{C}$ and $\mathsf{U}^{\mathsf{G}} : \mathsf{G}\text{-}\mathbf{Alg}(\mathscr{D}) \to \mathscr{D}$ are forgetful functors.

Given a distributive law $\lambda : \mathsf{H} \circ \mathsf{F} \leftarrow \mathsf{G} \circ \mathsf{H}$, we can define a lifting as follows:

$$\mathsf{H}^{\lambda}\,(A, a) = (\mathsf{H}\,A, \mathsf{H}\,a \cdot \lambda\,A) \ ,$$
$$\mathsf{H}^{\lambda}\,h = \mathsf{H}\,h \ .$$

For liftings, the action on the carrier and on homomorphisms is fixed; the action on the algebra is determined by the distributive law.

It is customary to use the action of an algebra to refer to the algebra itself. In this vein, we simplify our notation and use $\mathsf{H}^{\lambda}\,a$ to mean $\mathsf{H}^{\lambda}\,(A, a)$. We abuse this in certain contexts by using $\mathsf{H}^{\lambda}\,a$ for the arrow of the resultant algebra, $\mathsf{H}\,a \cdot \lambda\,A$.

Dually, a functor $\underline{\mathsf{H}} : \mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C}) \to \mathsf{G}\text{-}\mathbf{Coalg}(\mathscr{D})$ is called a *colifting* of $\mathsf{H} : \mathscr{C} \to \mathscr{D}$ iff $\mathsf{U}_{\mathsf{G}} \circ \underline{\mathsf{H}} = \mathsf{H} \circ \mathsf{U}_{\mathsf{F}}$. Given a distributive law $\lambda : \mathsf{H} \circ \mathsf{F} \dot{\to} \mathsf{G} \circ \mathsf{H}$ we can define a colifting as follows:

$$\mathsf{H}_{\lambda}\,(C, c) = (\mathsf{H}\,C, \lambda\,C \cdot \mathsf{H}\,c) \ ,$$
$$\mathsf{H}_{\lambda}\,h = \mathsf{H}\,h \ .$$

The distributive law $\lambda : \mathsf{F} \circ \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{F}$ underlying λ-bialgebras induces the lifting $\mathsf{N}^{\lambda} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C})$. The coherence conditions (3) ensure that N^{λ} is a comonad. In particular, the natural transformations ϵ and δ are F-algebra homomorphisms of type $\epsilon\,A : \mathsf{N}^{\lambda}\,(A, a) \to (A, a)$ and $\delta\,A : \mathsf{N}^{\lambda}\,(A, a) \to \mathsf{N}^{\lambda}\,(\mathsf{N}^{\lambda}\,(A, a))$. Dually, we can use λ to colift F to the category $\mathscr{C}_{\mathsf{N}}$. Now, the coherence conditions (3) guarantee that $\mathsf{F}_{\lambda} : \mathscr{C}_{\mathsf{N}} \to \mathscr{C}_{\mathsf{N}}$ preserves respect for ϵ and δ, that is, it maps coalgebras for N to coalgebras for N.

Returning to (7), the diagram on the left shows that $c : (X, a) \to \mathsf{N}^{\lambda}\,(X, a)$ is an F-algebra homomorphism. Dually, the diagram on the right identifies $a : \mathsf{F}_{\lambda}\,(X, c) \to (X, c)$ as an N-coalgebra homomorphism. Thus, we can interpret the bialgebra (X, a, c) both as an algebra over a coalgebra $((X, c), a)$, or as a coalgebra over an algebra $((X, a), c)$.

4 Eilenberg-Moore construction

We are nearly ready to tackle the proof of uniqueness. Before we head out to the garden, we should fetch one more tool from the shed. First observe that the recursion scheme (1)

involves actually two arrows: f and $\mathsf{N}\,f \cdot \mathit{fan}$. Perhaps surprisingly, the latter is an N-coalgebra homomorphism of type $(\mu\mathsf{F}, \mathit{fan}) \to (\mathsf{N}\,B, \delta\,B)$. To understand why, we delve a bit deeper into the theory.

Background: Eilenberg-Moore construction. The so-called Eilenberg-Moore construction [2] applied to comonads shows that arrows $f : A \to B$ of $\mathscr{C}$ and homomorphisms $h : (A, c) \to (\mathsf{N}\,B, \delta\,B)$ of $\mathscr{C}_{\mathsf{N}}$ are in one-to-one correspondence:

$$f = \epsilon\,B \cdot h \quad \Longleftrightarrow \quad \mathsf{N}\,f \cdot c = h \ . \tag{8}$$

The homomorphism h is also called the transpose of f. To get into a squiggoly mood, let us establish the one-to-one correspondence.

"$\Longrightarrow$": We have to show that $\mathsf{N}\,f \cdot c$ is an N-coalgebra homomorphism of type $(A, c) \to (\mathsf{N}\,B, \delta\,B)$

$$\begin{array}{ll} & \mathsf{N}\,(\mathsf{N}\,f \cdot c) \cdot c \\ = & \quad \{\ \mathsf{N} \text{ functor and } c \text{ coalgebra for } \mathsf{N}\ (5b)\ \} \\ & \mathsf{N}\,(\mathsf{N}\,f) \cdot \delta\,A \cdot c \\ = & \quad \{\ \delta \text{ natural and } f : A \to B\ \} \\ & \delta\,B \cdot \mathsf{N}\,f \cdot c\ , \end{array}$$

and that h is uniquely determined by $f = \epsilon\,B \cdot h$:

$$\begin{array}{ll} & h \\ = & \quad \{\ \text{comonad counit (2b)}\ \} \\ & \mathsf{N}\,(\epsilon\,B) \cdot \delta\,B \cdot h \\ = & \quad \{\ h : (A, c) \to (\mathsf{N}\,B, \delta\,B)\ \} \\ & \mathsf{N}\,(\epsilon\,B) \cdot \mathsf{N}\,h \cdot c \\ = & \quad \{\ \mathsf{N} \text{ functor and assumption: } f = \epsilon\,B \cdot h\ \} \\ & \mathsf{N}\,f \cdot c\ . \end{array}$$

"$\Longleftarrow$": For the other direction we reason

$$\begin{array}{ll} & f \\ = & \quad \{\ c \text{ coalgebra for } \mathsf{N}\ (5a)\ \} \\ & f \cdot \epsilon\,A \cdot c \\ = & \quad \{\ \epsilon \text{ natural and } f : A \to B\ \} \\ & \epsilon\,B \cdot \mathsf{N}\,f \cdot c \\ = & \quad \{\ \text{assumption: } \mathsf{N}\,f \cdot c = h\ \} \\ & \epsilon\,B \cdot h\ . \end{array}$$

5 Proof

Equipped with our shiny new tools, we can prepare the ground to solve the central problem, the proof that Equation (1) has a unique solution. Our strategy for conquering this proof is in two parts: first, we establish a bijection between certain arrows and λ-bialgebra homomorphisms; second, we instantiate the bijection to the initial λ-bialgebra. Without going into details, we assume that the ambient categories support the initial and final constructions that we will make use of.

5.1 Proof: First Half

We abstract away from the initial object $(\mu\mathsf{F}, in, fan)$, generalising to an arbitrary λ-bialgebra (A, a, c). The first goal is to establish a bijection between arrows $f : A \to B$ satisfying $f \cdot a = b \cdot \mathsf{F}\,(\mathsf{N} f \cdot c)$ and λ-bialgebra homomorphisms $h : (A, a, c) \to (\mathsf{N}\,B, b_\sharp, \delta\,B)$, where $b_\sharp$ is a to-be-determined F-algebra.

The Eilenberg-Moore construction (8) shows that arrows $f : A \to B$ and N-coalgebra homomorphisms $h : (A, c) \to (\mathsf{N}\,B, \delta\,B)$ are in one-to-one correspondence. So we identify $\mathsf{N} f \cdot c$ as the transpose of f and simplify f's equation to $f \cdot a = b \cdot \mathsf{F}\,h$.

$$\begin{array}{ccc} \mathsf{F}\,A & \xrightarrow{\mathsf{F}\,h} & \mathsf{F}\,(\mathsf{N}\,B) \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} \\ A & \xrightarrow[f]{} & B \end{array} \quad \Longleftrightarrow \quad \begin{array}{ccc} \mathsf{F}\,A & \xrightarrow{\mathsf{F}\,h} & \mathsf{F}\,(\mathsf{N}\,B) \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b_\sharp} \\ A & \xrightarrow[h]{} & \mathsf{N}\,B \\ {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle \delta\,B} \\ \mathsf{N}\,A & \xrightarrow[\mathsf{N}\,h]{} & \mathsf{N}\,(\mathsf{N}\,B) \end{array} \tag{9}$$

"$\Longrightarrow$": We already know that $h : (A, c) \to (\mathsf{N}\,B, \delta\,B)$ is an N-coalgebra homomorphism. It remains to show that h is an F-algebra homomorphism of type $(A, a) \to (\mathsf{N}\,B, b_\sharp)$, deriving $b_\sharp$ in the calculation. The strategy for the proof is clear: we have to transmogrify f into $\mathsf{N} f \cdot c$. Thus, we apply N to both sides of $f \cdot a = b \cdot \mathsf{F}\,h$ and then 'swap' a and c using the pentagonal law (6).

$$\begin{array}{ll} & f \cdot a = b \cdot \mathsf{F}\,h \\ \Longrightarrow & \quad \{\ \mathsf{N}\ \text{functor}\ \} \\ & \mathsf{N} f \cdot \mathsf{N}\,a = \mathsf{N}\,b \cdot \mathsf{N}\,(\mathsf{F}\,h) \\ \Longrightarrow & \quad \{\ \text{Leibniz}\ \} \\ & \mathsf{N} f \cdot \mathsf{N}\,a \cdot \mathsf{F}_\lambda\,c = \mathsf{N}\,b \cdot \mathsf{N}\,(\mathsf{F}\,h) \cdot \mathsf{F}_\lambda\,c \\ \Longleftrightarrow & \quad \{\ a : \mathsf{F}_\lambda\,(A, c) \to (A, c)\ (6)\ \} \\ & \mathsf{N} f \cdot c \cdot a = \mathsf{N}\,b \cdot \mathsf{N}\,(\mathsf{F}\,h) \cdot \mathsf{F}_\lambda\,c \\ \Longleftrightarrow & \quad \{\ \mathsf{F}_\lambda\,h : \mathsf{F}_\lambda\,(A, c) \to \mathsf{F}_\lambda\,(\mathsf{N}\,B, \delta\,B)\ \text{and}\ \mathsf{F}_\lambda\,h = \mathsf{F}\,h\ \} \\ & \mathsf{N} f \cdot c \cdot a = \mathsf{N}\,b \cdot \mathsf{F}_\lambda\,(\delta\,B) \cdot \mathsf{F}\,h \end{array}$$

$$\begin{array}{ll} & \mathsf{N} f \cdot c \cdot a = \mathsf{N}\,b \cdot \mathsf{F}_\lambda\,(\delta\,B) \cdot \mathsf{F}\,h \\ \Longleftrightarrow & \quad \{\ \mathsf{N} f \cdot c = h\ \} \\ & h \cdot a = \mathsf{N}\,b \cdot \mathsf{F}_\lambda\,(\delta\,B) \cdot \mathsf{F}\,h \end{array}$$

The proof makes essential use of the fact that a and h are N-coalgebra homomorphisms, and that F_λ preserves coalgebra homomorphisms. Along the way, we have derived a formula for $b_\sharp$:

$$b_\sharp = \mathsf{N}\,b \cdot \mathsf{F}_\lambda\,(\delta\,B) = \mathsf{N}\,b \cdot \lambda\,(\mathsf{N}\,B) \cdot \mathsf{F}\,(\delta\,B)\ . \tag{10}$$

We have to make sure that $(\mathsf{N}\,B, b_\sharp, \delta\,B)$ is a λ-bialgebra. Since $\mathsf{F}_\lambda\,(\mathsf{N}\,B, \delta\,B)$ is a coalgebra for the comonad N, we can conclude using (8) that $b_\sharp$ is a coalgebra homomorphism of type $\mathsf{F}_\lambda\,(\mathsf{N}\,B, \delta\,B) \to (\mathsf{N}\,B, \delta\,B)$, which establishes the desired result. Furthermore, we have $b = \epsilon\,B \cdot b_\sharp$, which is essential for the reverse direction:

"$\Longleftarrow$": Again, the strategy is clear: we have to transmogrify h into $\epsilon\, B \cdot h$. Thus, we precompose both sides of the homomorphism condition with $\epsilon\, B$.

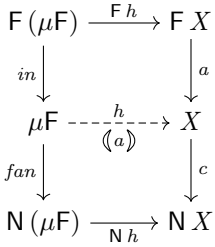
$$
\begin{array}{ll}
 & h \cdot a = b_\sharp \cdot \mathsf{F}\, h \\
\Longrightarrow & \quad \{ \text{ Leibniz } \} \\
 & \epsilon\, B \cdot h \cdot a = \epsilon\, B \cdot b_\sharp \cdot \mathsf{F}\, h \\
\Longleftrightarrow & \quad \{\ f = \epsilon\, B \cdot h \text{ and } b = \epsilon\, B \cdot b_\sharp\ \} \\
 & f \cdot a = b \cdot \mathsf{F}\, h
\end{array}
$$

To summarise, f and h are related by the Eilenberg-Moore construction, as are b and $b_\sharp$.

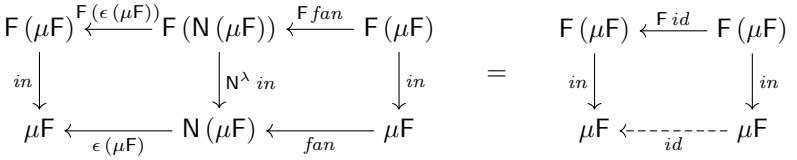
5.2 Proof: Second Half

Now, we can reap the harvest: the initial object in the category of λ-bialgebras is $(\mu\mathsf{F}, in, fan)$ where $fan = (\!|\mathsf{N}^\lambda\, in|\!) = (\!|\mathsf{N}\, in \cdot \lambda\,(\mu\mathsf{F})|\!)$. Several proof obligations arise. We have already noted that the pentagonal law (6) holds, see Diagram (4).

Since $(\mu\mathsf{F}, in)$ is the initial F-algebra there is a unique F-algebra homomorphism h to any target algebra. Because of uniqueness, h is also an N-coalgebra homomorphism—recall that the coalgebra of a λ-bialgebra is simultaneously an F-algebra homomorphism.



It remains to show that $(\mu\mathsf{F}, fan)$ is a coalgebra for the comonad N. The proofs make essential use of the fact that ϵ and δ are F-algebra homomorphisms. The coalgebra fan respects ϵ (5a):



It also respects δ (5b):

$$\begin{array}{ccccc}
\mathsf{F}\,(\mathsf{N}\,(\mathsf{N}\,(\mu\mathsf{F}))) & \xleftarrow{\mathsf{F}\,(\delta\,(\mu\mathsf{F}))} & \mathsf{F}\,(\mathsf{N}\,(\mu\mathsf{F})) & \xleftarrow{\mathsf{F}\,fan} & \mathsf{F}\,(\mu\mathsf{F}) \\
{\scriptstyle \mathsf{N}^\lambda\,(\mathsf{N}^\lambda\,in)}\downarrow & & \downarrow{\scriptstyle \mathsf{N}^\lambda\,in} & & \downarrow{\scriptstyle in} \\
\mathsf{N}\,(\mathsf{N}\,(\mu\mathsf{F})) & \xleftarrow[\delta\,(\mu\mathsf{F})]{} & \mathsf{N}\,(\mu\mathsf{F}) & \xleftarrow[fan]{} & \mu\mathsf{F}
\end{array}$$

$$= \quad \begin{array}{ccc}
\mathsf{F}\,(\mathsf{N}\,(\mathsf{N}\,(\mu\mathsf{F}))) & \longleftarrow & \mathsf{F}\,(\mu\mathsf{F}) \\
{\scriptstyle \mathsf{N}^\lambda\,(\mathsf{N}^\lambda\,in)}\downarrow & & \downarrow{\scriptstyle in} \\
\mathsf{N}\,(\mathsf{N}\,(\mu\mathsf{F})) & \longleftarrow & \mu\mathsf{F}
\end{array}$$

$$= \quad \begin{array}{ccccc}
\mathsf{F}\,(\mathsf{N}\,(\mathsf{N}\,(\mu\mathsf{F}))) & \xleftarrow{\mathsf{F}\,(\mathsf{N}\,fan)} & \mathsf{F}\,(\mathsf{N}\,(\mu\mathsf{F})) & \xleftarrow{\mathsf{F}\,fan} & \mathsf{F}\,(\mu\mathsf{F}) \\
{\scriptstyle \mathsf{N}^\lambda\,(\mathsf{N}^\lambda\,in)}\downarrow & & \downarrow{\scriptstyle \mathsf{N}^\lambda\,in} & & \downarrow{\scriptstyle in} \\
\mathsf{N}\,(\mathsf{N}\,(\mu\mathsf{F})) & \xleftarrow[\mathsf{N}\,fan]{} & \mathsf{N}\,(\mu\mathsf{F}) & \xleftarrow[fan]{} & \mu\mathsf{F}
\end{array} \quad .$$

Note that $\mathsf{N}\,fan$ is the lifting of fan and hence an F-homomorphism. Since there is only one homomorphism from $(\mu\mathsf{F}, in)$ to $\mathsf{N}^\lambda\,(\mathsf{N}^\lambda\,(\mu\mathsf{F}, in))$, both compositions are equal.

Consequently, the unique homomorphism from the initial λ-bialgebra to the bialgebra $(\mathsf{N}\,B, b_\sharp, \delta\,B)$ is $h = (\!|b_\sharp|\!)$.

$$\begin{array}{ccc}
\mathsf{F}\,(\mu\mathsf{F}) & \xrightarrow{\mathsf{F}\,h} & \mathsf{F}\,(\mathsf{N}\,B) \\
{\scriptstyle in}\downarrow & & \downarrow{\scriptstyle b_\sharp} \\
\mu\mathsf{F} & \xrightarrow[(\!|b_\sharp|\!)]{h} & \mathsf{N}\,B \\
{\scriptstyle fan}\downarrow & & \downarrow{\scriptstyle \delta\,B} \\
\mathsf{N}\,(\mu\mathsf{F}) & \xrightarrow[\mathsf{N}\,h]{} & \mathsf{N}\,(\mathsf{N}\,B)
\end{array}$$

Furthermore, $f = \epsilon\,B \cdot h = \epsilon\,B \cdot (\!|b_\sharp|\!)$ is the unique solution of

$$\begin{array}{ccc}
\mathsf{F}\,(\mu\mathsf{F}) & \xrightarrow{\mathsf{F}\,(\mathsf{N}\,f \cdot fan)} & \mathsf{F}\,(\mathsf{N}\,B) \\
{\scriptstyle in}\downarrow & & \downarrow{\scriptstyle b} \\
\mu\mathsf{F} & \xrightarrow[f]{} & B
\end{array} \quad .$$

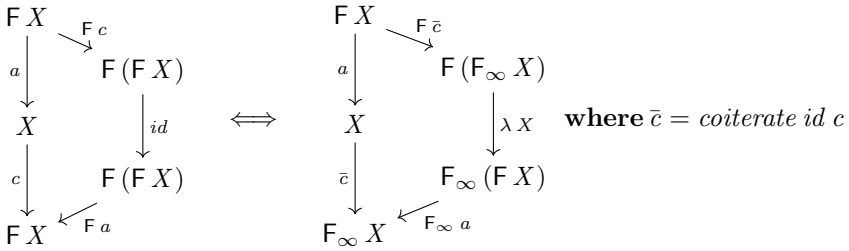
6 Knapsack revisited

Obtaining an efficient implementation of *knapsack* is now simply a matter of instantiating the framework above to the cofree comonad of $\mathbb{N}$.

$$\begin{array}{l}
knapsack_4 :: \mathbb{N} \to Value \\
knapsack_4 = head \cdot (\!|knap_\sharp|\!) \\
(-)_\sharp :: (Functor\,f) \Rightarrow (f\,(f_\infty\,a) \to a) \to (f\,(f_\infty\,a) \to f_\infty\,a) \\
b_\sharp = fmap\,b \cdot dist \cdot fmap\,duplicate
\end{array}$$

Recall that *knap* is a context-sensitive algebra of type $\mathsf{Nat}\,(\mathsf{Nat}_\infty\ \mathit{Value}) \to \mathit{Value}$: as such it has access to the recursive images of all natural numbers smaller than the current one. The implementation of $(-)_\sharp$ builds on the generic definition that works for an arbitrary comonad. As a final tweak let us simplify its implementation for the comonad at hand:

We have emphasised before that $b_\sharp$ is a coalgebra for N and consequently an N-coalgebra homomorphism. If N is the cofree comonad F_∞, then $b_\sharp$ is also an F-coalgebra homomorphism, which is the key to improving its definition. A central observation is that λ-bialgebras with $\lambda : \mathsf{F}\circ\mathsf{F}_\infty \dot{\to} \mathsf{F}_\infty\circ\mathsf{F}$ are in one-to-one correspondence to id-bialgebras with $id : \mathsf{F}\circ\mathsf{F} \dot{\to} \mathsf{F}\circ\mathsf{F}$. (The correspondence builds on the fact that the category of F-coalgebras is isomorphic to the (co)-Eilenberg-Moore category $\mathscr{C}_{\mathsf{F}_\infty}$.)

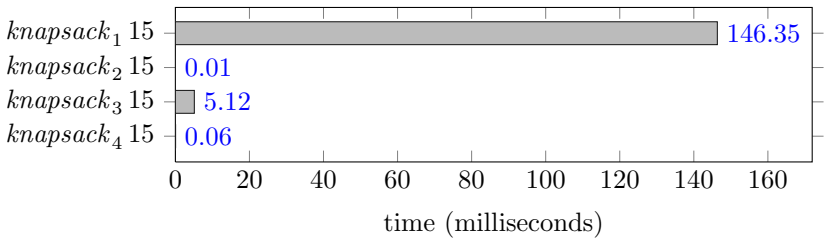


Since the comultiplication is defined $\delta\ A = \mathit{coiterate}\ \mathit{id}\ (\mathit{tail}\ A)$, the id-bialgebra corresponding to the λ-bialgebra $(\mathsf{N}\ B, b_\sharp, \delta\ B)$ is $(\mathsf{N}\ B, b_\sharp, \mathit{tail}\ B)$. Consequently $b_\sharp$ is an F-coalgebra homomorphism: $\mathit{tail}\ B \cdot b_\sharp = \mathsf{F}\ b_\sharp \cdot \mathsf{F}\ (\mathit{tail}\ B)$. Since furthermore $\mathit{head}\ B \cdot b_\sharp = b$, we have $b_\sharp = \mathit{coiterate}\ b\ (\mathsf{F}\ (\mathit{tail}\ B))$.

$$
\begin{aligned}
&(-)_\sharp :: (\mathit{Functor}\ f) \Rightarrow (f\ (f_\infty\ a) \to a) \to (f\ (f_\infty\ a) \to f_\infty\ a)\\
&b_\sharp = \mathit{coiterate}\ b\ (\mathit{fmap}\ \mathit{tail})
\end{aligned}
$$

Quite interestingly, the final solution of the unbounded knapsack problem, that is, $\mathit{knapsack} = \mathit{head} \cdot (\!|\mathit{coiterate}\ \mathit{knap}\ (\mathit{fmap}\ \mathit{tail})|\!)$ is roughly a *fold of a coiteration*, a generalisation of a fold of an unfold—recall that unfolds are related to coiterations in the same way final coalgebras are related to cofree comonads.

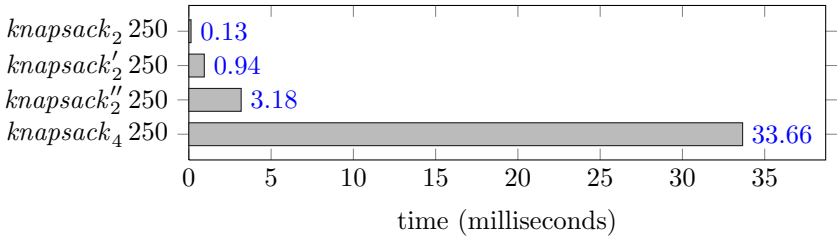
Benchmarks. We now have four different versions of knapsack, and it is a worthwhile exercise to compare their performance with some benchmarks. The charts below show the results of the mean time from a sample of 1000 measurements. The first benchmark presents the results of solving the problem with a capacity of 15.



The results of $\mathit{knapsack}_1$ are underwhelming, even for very small knapsack capacities. This is entirely to be expected, given that it is an exponential algorithm, and served only as a specification of the problem.

We have claimed that our final derived version of $\mathit{knapsack}_4$ is efficient, so how does it compare with the array-based version discussed in Section 1? Looking more closely at

$knapsack_2$ and $knapsack_4$, over a much larger capacity of 250, shows that despite our efforts, the version based on arrays is still significantly faster.



We might expect a result along these lines, given that $knapsack_2$ uses constant time look-ups in the array that is built, whereas $knapsack_4$ must still perform a linear traversal to get to its data. However, the results of $knapsack'_2$ show what happens when we replace the underlying array structure of $knapsack_2$ with a list that is treated as an indexed structure using the (!!) operator. Similarly $knapsack''_2$ is a version where the list is treated as an association list and indexed using *lookup* from the prelude. This difference in performance is rather disappointing, but note, however, that $knapsack_1$ and $knapsack_3$ were unable to complete within a reasonable time. Why is $knapsack_4$ so much slower?

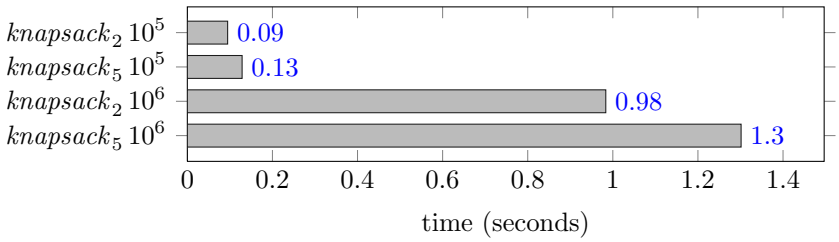
The main problem occurs not in the look-up of values, but rather, in the construction of the look-up table. For $knapsack_2$, a single iteration is required to build the table. Looking more carefully at the code that was derived for $knapsack_4$, it should be clear that the function responsible for creating the tables, *knap*, is used within a *coiterate* that is nested in a fold. Thus there is a linear factor difference between the two algorithms.

However, all is not lost, since we can use this observation to adjust our definition to become the following:

$$
\begin{aligned}
&knapsack_5 :: \mathbb{N} \to Value \\
&knapsack_5 = head \cdot (\!|knap_\flat|\!) \\
&(-)_\flat :: (f\ (f_\infty\ a) \to a) \to (f\ (f_\infty\ a) \to f_\infty\ a) \\
&b_\flat\ ts = Cons\ (b\ ts)\ ts\ \ .
\end{aligned}
$$

The algebra $b_\flat$ constructs the look-up table in time proportional to the running-time of b, cleverly re-using its argument for the tail of the table.

How does this compare to $knapsack_2$? The benchmarks show that the two are now in the same ball park, and their performance scales linearly. Not bad!



But what about the proof that $knapsack_5$ is correct? Does it follow the specification? What is its relationship to bialgebras? We leave the proof that $knapsack_5$ satisfies our requirements to the avid reader: without a doubt, this should be a manageable task for a distinguished professor with plenty of time on his hands.

7 Conclusion

In this paper we have given a proof of correctness of recursion schemes from comonads. Along the way, we have shown derivations of the unique arrow that solves these schemes, and presented ways of optimising this computation. Our analysis shows that the optimisations we introduced improve upon the efficiency of the standard definition of a histomorphism. Furthermore, the final version we presented, whose derivation is left as a challenge, is comparable to an array-based version. While the efficiency of our final algorithm falls slightly short of an array-based one, it gains in an important way: by construction, it is guaranteed to terminate, and we squiggolers favour correctness over speed. And so, we keenly await the derivation of our final implementation.

Acknowledgements

The authors would like to thank Jeremy Gibbons for pointing them to the knapsack problem as an interesting example of a histomorphism, and for his useful suggestions for improving this paper.

On a personal note, I would like to thank you, Doaitse, for your support and encouragement over the past fifteen years. I do hope that you enjoy your newly gained freedom. Ralf

References

1. Uustalu, T., Vene, V., Pardo, A.: Recursion schemes from comonads. Nordic J. of Computing **8** (September 2001) 366–390
2. Eilenberg, S., Moore, J.C.: Adjoint functors and triples. Illinois J. Math **9**(3) (1965) 381–398

Een Veelkleurige Universiteit

Marinus Veldhorst

Departement Informatica, Universiteit Utrecht

Toen ik na een verblijf van twee jaar in de USA in 1984 als Universitair Docent terugkeerde naar de Vakgroep Informatica[1] van de Rijksuniversiteit Utrecht[2], trof ik daar een nieuwe collega aan, namelijk prof.dr. S. Doaitse Swierstra. Ik maakte geen deel uit van zijn leerstoelgroep, maar kon hem ook niet ontlopen, omdat hij toen voorzitter van het vakgroepsbestuur was. Nu kun je überhaupt een hoofd van een vakgroep moeilijk ontlopen, zeker niet als de vakgroep een kleine omvang heeft, maar daar gaat het nu niet om. Ik werd zelfs geacht met Doaitse samen te werken, want er werd mij al snel gevraagd om zitting te nemen in het Dagelijks Bestuur van de Vakgroep.

Sinds 1984 heb ik bestuurlijk gezien vaak dicht bij Doaitse en samen met Doaitse gefunctioneerd:

- Samen in het Dagelijks Bestuur van de Vakgroep Informatica (1984 – 1987)
- Samen in de Faculteitsraad van de faculteit Wiskunde en informatica (1987 – 1988)
- Samen in de Examencommissie van de Opleiding Informatica
- Samen in het Dagelijks Bestuur van het Informatica Instituut (ong 1999 – 2003).

En dan waren daar ook nog de klussen als visitaties onderwijs en visitaties onderzoek die ik meerdere keren vanuit het Departement Informatica gecoördineerd heb, en waarin o.a. met Doaitse afgestemd moest worden. Of Doaitse nu wel of niet op dat moment een bestuurstaak had, deed niet ter zake. Hij drukte op een open manier toch zijn stempel op de visitaties, om de eenvoudige reden dat hij hart voor de zaak had. Kortom, jarenlang heb ik op een prettige manier in allerlei zaken met Doaitse samengewerkt. En Doaitse bleek een kleurrijk iemand te zijn. Was de omgeving waar hij werkte, te weten de Universiteit Utrecht, ook kleurrijk? Vanuit een bestuurlijke gezichtspunt zou ik daar graag uit eigen ervaring wat over willen schrijven.

1 Verandering in organisatie

Al die jaren was ik Universitair Docent, dus besturen/managen was een bijtaak, maar voor een Departement wel een essentiële bijzaak. Je leert mensen kennen, en je leert de cultuur van Vakgroep/Instituut/Departement kennen. In 2010 werd ik bestuurssecretaris voor het Departement, en daarmee werd besturen/managen/ondersteunen een hoofdtaak en verschoof de aandacht meer en meer naar de relatie tussen Departement en Faculteit, tussen Departement en Universiteit, etc. Die buitenwereld (zo noem ik nu maar even het conglomeraat van Faculteit, Universiteit, NWO, Nederland, Europa) verwacht bewust en onbewust, gestuurd of ongestuurd, van alles en nog wat van het Departement; b.v. dat het zich mee verandert samen met deze buitenwereld. Hoe doe je dat als Departement? Als bestuurssecretaris mag je van de facultaire afdeling Personeel & Organisatie met allerlei cursussen meedoen die relevant (zouden) zijn voor zogenoemd veranderingsmanagement.

Op zich is er niets mis met die aandacht voor verandering. Opvallend is eerder dat er zo weinig bestuurlijke aandacht lijkt te zijn voor behoud en onderhoud. Voor zangers in een

[1] In de loop der jaren sindsdien uitgegroeid tot het huidige Departement Informatica

[2] Sindsdien wegens naamswijziging geworden tot Universiteit Utrecht

koor is behoud van hetzelfde, d.w.z. het lang aanhouden van een en dezelfde toon, verre van simpel. Om zo'n toon mooi te houden moet de individuele zanger volle mentale aandacht aan die ene toon geven; en tegelijkertijd moet met medezangers afgestemd worden om te voorkomen dat allen tegelijkertijd op hetzelfde moment adem halen. Het koormanagement, zijnde de dirigent, moet regelmatig aandacht besteden aan deze problematiek.

Ja, die buitenwereld verwacht verandering. Verwacht die buitenwereld dan geen behoud? Natuurlijk verwacht de buitenwereld behoud van goede zaken; alleen zegt ze het niet. Pas als iets goeds verdwijnt (een verandering dus), wordt dat opgemerkt, en wordt een terugverandering verwacht. Zo gaat dat nu eenmaal met mensen: wij zien alleen maar verschillen en het constante, het ongewijzigde valt niet op, krijgt geen aandacht en komt niet in de pers. Het behoud van het goede wordt vaak voor vanzelfsprekend aangenomen. Als individu kun je je wel bewust maken van het gewone: onthoud geschiedenis en besef dat het huidige gewone een verschil betreft met het gewone van vroeger.

Er valt dus best wat voor te zeggen om naast de aandacht voor veranderingsmanagement ook behoudmanagement in een contekst van veranderende omgeving op het aandachtslijstje te zetten, compleet met theorievorming, cursussen, toetsing, en de hele rataplan meer. Het zou naar mijn mening geen kwaad kunnen als naast iedere cursus over veranderingsmanagement door dezelfde cursusleiding ook een cursus voor dezelfde doelgroep gegeven zou worden over kwaliteit in continuïteit. Eigenlijk is mij onbekend of in de organisatieleer er enige theorievorming is over kwaliteit in continuïteit.

Nu maar weer eens terug naar veranderingsmanagement. In het praten over veranderingsmanagement (in cursussen, in boeken) gaat het altijd over mensen (of groepen mensen) die zich dienen te verhouden tot mogelijke en/of gewenste veranderingen in en/of voor een organisatie. Een aardige theorie vond ik de kleurentheorie van De Caluwé en Vermaak. In deze theorie worden kleuren toegekend aan ..... Ja, aan wat/wie eigenlijk? Het hebben van een kleur is natuurlijk een eigenschap. Maar wat of wie is nu de drager van zo'n eigenschap? Zijn individuen dragers, zijn groepen individuen dragers, is gedrag een drager, is een maatregel een drager van een kleur?
Om de kleurentheorie uit te leggen, zal ik mensen indelen volgens bepaald gedrag. En dan worden types personen drager van een kleur. Concrete personen kunnen dan afhankelijk van de contekst waarin ze functioneren een bepaalde kleur kiezen[3].

Verder zal ik de kleuren meer tot leven brengen door ze te koppelen aan wat ik de afgelopen jaren heb gezien en geobserveerd aan deze universiteit. En daarbij wil ik dan ook graag kijken naar de hiërarchie, en niet alleen de hiërarchie binnen de universiteit, maar ook ruimer:

Europees beleid en regelgeving
|
De Nederlandse minister en/of de Nederlandse wetten
|
Het Ministerie en zijn regelgeving
|
De Universiteit Utrecht (CvB)
|
De Faculteit en het Faculteitsbestuur
|
Het Departement, de onderwijsdirecteur, de onderzoekdirecteur

[3] Dat ik hier het werkwoord 'kiezen' gebruik, is al een tamelijk 'gele' opmerking; voor de uitleg van de kleur geel, zie verderop.

2 De kleur Blauw

Dit type mens kijkt tegen organisatieverandering aan vanuit bedrijfsprocessen en optimalisering. De persoon zal niet alleen een blauwdruk maken van de veranderde toekomstige organisatie, maar ook van de veranderende organisatie, dus ook van de weg naar het doel. Technische risico's worden van te voren ingeschat. Zo'n aanpak vooronderstelt een goed zicht op het doel en de doelstelling van de verandering.

Een voorbeeld hiervan uit het verleden van het Departement is het zogenoemde 'gele boek' uit omstreeks 1992[4]. Het behelsde een volledige herziening van het onderwijsprogramma informatica en de vakken in de eerste twee jaar werden volledig beschreven qua verplichte inhoud en qua mogelijke extra inhoud. De besluitvorming hierover vond centraal in het Departement plaats, en werd niet per vak afzonderlijk door een docent of leerstoelgroep genomen.
Volgens mij was dit een van de weinige keren vanaf 1984 dat er zo centraal en zo gedetailleerd over het onderwijsprogramma van de opleiding informatica is besloten.

Een andere manier om de kleur blauw te overzien, zit misschien wel in het aspect van kwaliteit in continuïteit. Waar kwaliteit in (danwel ondanks) verandering behouden moet blijven, kan een blauwe aanpak misschien garanties bieden. Denk aan behoud van een kwalitatief hoogstaande financiële administratie, aan behoud van kwaliteit in promoties, aan behoud van kwaliteit in het onderwijsprogramma, etc. Regelgeving is bovendien een middel van hogere niveaus in de hiërarchie om te zorgen dat lagere niveaus aan bepaalde randvoorwaarden blijven voldoen.

De huidige eisen en regelgeving van Ministerie en onderwijsinspectie betreffende onderwijsdoelen, eindtermen, opleidingsvakken alsmede de toetsing van dit alles heeft een sterk blauw karakter met als doel (van het Ministerie) behoud en stimulans van kwaliteit in een veranderende omgeving. Dit gaat er daarmee vanuit dat de invulling van het begrip kwaliteit nauwelijks zal veranderen de komende jaren. Zijn daar vraagtekens bij te zetten?

3 De kleur Groen

Het type mens van de kleur groen gaat er van uit dat veranderingen het beste door mensen van binnen uit gerealiseerd kunnen worden, en hen als het ware zelf lerenderwijs het doel van de verandering maar ook de weg erheen te laten ontdekken danwel bedenken. Iemand van binnen de veranderende organisatie is als het ware de leerling, en iemand van buiten de organisatie gedraagt zich als coach, als didacticus. Echter, een groene buitenstaander met een vleugje rood en een vleugje geel kan gemakkelijk een manipuleerder worden.

Groen is in het Departement een veel voorkomende kleur. Een verandering in een vak wordt veelal aan een docent overgelaten met enige coaching door de hoogleraar. Het concrete materiaal wordt in vrijheid ontworpen.

[4] In 1992 hadden De Caluwé en Vermaak hun kleurentheorie nog niet geformuleerd; dus het woordje 'geel' in het gele boek heeft niets met hun theorie te maken. Het woordje 'geel' sloeg louter op de kleur van de omslag van het document.

Ook op een hoger niveau komt de kleur groen vrij veel voor. De Faculteit wenste dat het Departement zich zou profileren naar de Faculteit Bètawetenschappen toe. Hoe zij dat moest doen, werd in principe aan het Departement zelf overgelaten door een zogenoemde Task Force in te richten die goed voeling moest hebben met het Departement.

Doaitse heeft in het kader van functioneel programmeren in het onderwijsprogramma nog iets op een groene manier voor elkaar gekregen. Doaitse was toentertijd nauw betrokken bij het vak Functioneel Programmeren. Ik was docent van het vak Datastructuren en nauwelijks bekend met functioneel programmeren. Doaitse was van mening dat de inhouden van deze twee vakken dicht bij elkaar stonden (of zouden kunnen staan). Hij deed de suggestie dat ik werkcollegeleider FP kon worden en op die manier ingevoerd zou worden in het functioneel programmeren. Toen de gelegenheid daar was, heb ik zelf geregeld dat ik werkcollegeleider bij FP werd. Op basis van wat ik daar leerde, heb ik de presentatie van algoritmen in het vak Datastructuren gewijzigd. En het werd opgepikt door studenten: *"Uw vak Datastructuren lijkt toch wel erg veel op FP"* mocht ik op een gegeven moment uit de mond van een student horen.
Terugkijkend: Doaitse had iets voor ogen, maar niet concreet. Hij gaf mij de gelegenheid dat doel zelf in te vullen en uit te werken. En dat is gebeurd, en gerealiseerd, al weet ik dat Doaitse graag gezien had dat de integratie van beide vakken verder zou gaan.

4 De kleur Rood

Het type mens van de kleur rood hecht aan de inbreng van mensen in de verandering. Het gedrag van mensen moet veranderen, en daartoe moeten mensen worden aangezet. Deze aanpak veronderstelt dat het doel van een verandering redelijk duidelijk is en dat men zicht heeft op de houding van individuen in het veranderingsproces. En vanuit het zicht op mensen kunnen zij gestimuleerd en aangespoord worden, b.v. met beloning en stimulansen in wat voor vorm dan ook (gestimuleerd, beloond, gestraft, etc). Het verschil met de aanpak van een groen persoon zit vooral daarin hoe de rode persoon denkt over de intrinsieke motivatie en intrinsieke beweegredenen van de ander. Ik vraag mij wel af of de rode persoon van buiten de te veranderen organisatie voor de verleiding kan komen te staan om de speler van het poppenkasttoneel te worden. De speler is onzichtbaar voor het publiek, maar is wel de grote roerganger in het spel.

Rode voorbeelden zijn er te over, zowel in de maatschappij als aan de universiteit. Bijvoorbeeld, beleidslijnen die uitgezet worden met beloning. Het zijn lang niet altijd de leidinggevenden die rood gekleurd zijn. Ook studenten die louter denken in studiepunten, nut voor het tentamencijfer, maar zonder een intrinsieke motivatie te hebben voor het onderwerp zelf, tonen een tamelijk rode kleur. En geldverdelers die een bonus zetten op b.v. interdisciplinaire onderzoeksprojecten om op die manier het interdisciplainaire onderzoek te stimuleren, tonen daarin enige rode denkwijzen. Onderzoekers die hierin meegaan en een beloning verwachten, laten daarin ook wat zien.
Samenwerking wordt gestimuleerd b.v. door het beschikbaar stellen van reisgeld. Conferentiebezoek heeft niet alleen tot doel om je wetenschappelijke resultaat te presenteren of voordrachten aan te horen, maar heeft ook tot doel om met andere onderzoekers kennis te maken en samen te werken.

Ook Doaitse heeft wel wat roods laten zien in het verleden. Zo heeft hij eens een stylefile voor Latex gemaakt (waarschijnlijk voor zichzelf, maar later aangeboden aan anderen) waarmee je gemakkelijk notulen van vergadering kon maken, als ook de voortgang en controle op uitvoering wat kon bijhouden. De beloning voor de notulist zit dan in de snelheid en gemak waarin dingen geregeld worden, zodat meer tijd overblijft voor andere (leukere?) zaken.

5 De kleur Geel

Een mens van het type geel ziet een organisatie en daarmee ook de verandering in een organisatie als een samenspel van en tussen belangen. Hij accepteert daarmee verschillen in beweegredenen van de verschillende personen/organisaties in een veranderingsproces. Hij beseft heel goed dat men vanuit verschillende redenen best dezelfde verandering kan nastreven. Hij zal proberen om coalities te smeden en belangen bijeen te brengen, maar zal ook pogen belangen te wijzigen, en is bereid gebruik te maken van hiërarchie. Tevens is de persoon heel gevoelig voor het juiste tijdstip om bepaalde acties te plegen of voorstellen te doen.

Geel gedrag doet zich makkelijk voor in situaties waarbij er iets van beperkte omvang verdeeld moet worden over meerdere min of meer onafhankelijke partijen. Ergens omstreeks 1990 heeft het College van Bestuur een naar mijn mening typisch gele maatregel genomen. Zij nam het besluit dat de UU niet meer van de wieg tot het graf voor de werknemer zou zorgen, maar zou er voortaan van uit gaan dat de werknemer voor zijn eigen belangen zou staan. Het is een gele maatregel omdat het uitgaat van een soort belangentegenstelling. Zo'n maatregel kan oranje worden door er rood bij te voegen (beloning). Misschien is dat ook wel gebeurd; denk b.v. aan het instellen van Basis en Senior onderwijskwalificaties. Maar het is zeker geen groene maatregel waarin werkgever en werknemers als het ware samen een leertraject ingaan.

Uberhaupt zit het ingewikkeld aan een universiteit waar het belangen betreft. Er is een wetenschappelijk belang, te onderscheiden in belang voor onderzoek, belang voor onderwijs en belang voor maatschappelijke dienstverlening. Dan is er het belang van de organisatie, met daarin dan weer de belangen van de delen van de organisatie te zien; en er zijn de belangen van de individuele medewerkers. Al deze belangen overlappen en beïnvloeden elkaar maar zijn niet identiek. Met zo'n ingewikkelde belangenstructuur is er natuurlijk ook veel geel gedrag.

De Universiteit Utrecht kent de kleur geel al heel lang, en hecht er veel waarde aan. Zoveel waarde zelfs dat het als een belangrijke kleur in het sol-embleem is opgenomen. Maar gelukkig werd het sol-embleem van de universiteit al ontworpen voordat de kleurentheorie van De Caluwé en Vermaak het licht zag.

6 De kleur Wit

Mensen van deze kleur zijn eigenlijk de ideale mensen om veranderingen in organisaties te weeg te brengen. Zij zijn een mix van alle andere kleuren, en weten de juiste aanpak op een juiste manier op het juiste moment in te zetten.

Conclusie

Natuurlijk is de universiteit een kleurrijke organisatie, en het is goed als de organisatie in zijn geheel een witte kleur heeft. Maar als het om personen gaat, wil ik wel betwijfelen of witte personen de meest interessante mensen zijn. Witte personen kunnen je met een intense uitstraling gemakkelijk verblinden. Geef mij dan maar een kleurrijk iemand die met een ongebalanceerde kleurenmix een mooie kleur vertoont.

Doing a Doaitse: Simple Recursive Aggregates in Datalog

Oege de Moor

oege@semmle.com
Semmle Limited, Blue Boar Court, 9 Alfred Street, Oxford OX1 4EH, UK

Abstract. Recursive aggregates in Datalog are considered problematic because aggregates are not monotonic with respect to subset inclusion. By presenting an embedding of Datalog in Haskell, we show that there is a natural way of redefining aggregates so that they are monotonic.

1 Introduction

The year is 1987; a group of undergraduates taking Doaite's compiler class have been battling the GAG attribute grammar system to create a compiler for OCCAM, working overnight on the VAX. There is an insidious bug that remains. Morning has come, and Doaitse cheerfully loiters behind the frustrated students, and peers over our shoulders. After a minute or two he exclaims "Aha — there is the problem!" A stunningly simple solution side-steps the difficulties. We gratefully accept the idea and go home to sleep, feeling a little sheepish. How did he see that so quickly?

Over the years we have seen Doaitse do this time and again, and we started to give the phenomenon a name: *Doing a Doaitse*. *Doing a Doaitse* means to take a fresh look at a problem, to ignore that someone else said it's difficult, and to make the problem vanish, preferably by ingenious reduction to an easy idea in functional programming. Oh, and you have to make everyone else feel bad by saying "It's all really simple". Very hard problems may require the additional remark "I thought about it on the bike ride from Houten".

This paper attempts to *Do a Doaitse* on a vexing problem in the design of a query language. We shall pose the problem and its solution in Haskell because that way the solution stares you in the face. The problem has been open for over 20 years, however, and the solution has big monetary value to our company.

2 The problem

Datalog is a small logic programming language that enables the definition of recursive relations [1]. The semantics are disarmingly simple: just least fixpoints in the obvious subset order on relations. The existence of such fixpoints is guaranteed by the Knaster-Tarski fixpoint theorem, and they can be computed by iteration from the empty relation. Unlike more well-known logic programming languages such as Prolog, the clean semantics of Datalog enables aggressive optimisations. Indeed, in its purest form, without arithmetic, all Datalog queries are guaranteed to terminate.

Datalog has been very well-studied in the theoretical database community [2] and recently it has seen an explosion of applications [3]. The core technology of our company named Semmle is an optimising compiler for a modern object-oriented variant of Datalog [4–8].

The problem that we seek to address here is to allow the use of recursive aggregate operations in Datalog — taking the maximum, minimum, average, sum and so forth. This

is a problem because the obvious definition of aggregates is not monotonic with respect to subset inclusion, and so the existence of least fixpoints cannot be guaranteed. For over twenty years this issue has been the subject of deep academic studies, motivating the invention of complex new semantics of Datalog, and new implementation techniques. A sampling of these efforts spanning the years 1991–2011 is [9–14].

We are going to take a fresh look at the problem by creating a small set of combinators for embedding Datalog into Haskell.

3 Relations as set-valued functions

A common view of binary relations is as total set valued functions:

$$\textbf{type}\ a \leftrightarrow b = a \to \mathit{Set}\ b$$

The obvious point-wise order on such set-valued functions corresponds to inclusion of relations. It cannot be implemented directly in Haskell because relations can be infinite, even if we stipulate that the result sets are finite.

For example, here is a relation that relates a list to its maximum element. If the list is empty, the result is empty, and otherwise it is the singleton containing just the maximum element:

$$\begin{aligned}&\mathit{listMax} :: \mathit{Ord}\ a \Rightarrow [a] \leftrightarrow a\\ &\mathit{listMax}\ \mathit{as} \mid \mathit{null}\ \mathit{as} \quad = \mathit{empty}\\ &\qquad\qquad \mid \mathit{otherwise} = \mathit{singleton}\ (\mathit{maximum}\ \mathit{as})\end{aligned}$$

As functional programmers, whenever we introduce a new type of arrow it makes sense to think about higher-order operations like *map*, which applies an operation to all elements of a data structure. When mapping a relation over a list, one possible definition is to use relational image:

$$\begin{aligned}&\mathit{imageRel} :: (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow (a \leftrightarrow b) \to ([a] \leftrightarrow [b])\\ &\mathit{imageRel}\ r = \mathit{singleton} \circ \mathit{concatMap}\ (\mathit{toList} \circ r)\end{aligned}$$

This is, in fact, the operator you find in most text books on discrete mathematics, for applying a relation to a set of values, and in more familiar notation we could write the right-hand side as

$$\mathit{imageRel}\ r\ x = \{\{\, b \mid a \leftarrow x, b \leftarrow r\ a \,\}\}$$

Note that in the Haskell definition of *imageRel* we make use of *toList* to guarantee a particular sequence of results, relying on the ordering relation on the underlying type b. Later we shall return to this point, as it is important in the semantics of aggregates.

Note that *imageRel* satisfies the usual properties of a functor, namely that it preserves the identity relation

$$\mathit{imageRel}\ \mathit{singleton} = \mathit{singleton}$$

and it preserves composition of relations (modulo the list representation of sets, writing $\circ$ for relational composition):

$$\begin{aligned}&\mathit{fromList} \circ \mathit{imageRel}\ (r \circ s)\\ &=\\ &\mathit{fromList} \circ \mathit{imageRel}\ r \circ \mathit{imageRel}\ s\end{aligned}$$

Although the above definition of *imageRel* is familiar from discrete mathematics, it is not monotonic. By contrast, this definition, which is more natural to experienced functional programmers, is monotonic:

$$mapRel :: (Ord\ a, Ord\ b) \Rightarrow (a \leftrightarrow b) \rightarrow ([a] \leftrightarrow [b])$$
$$mapRel = mapM$$

Here we are relying on *mapM*, which is a monad primitive. Spelling out its definition, we have

$$mapRel\ f = cp \circ map\ f$$

where *cp* is the Cartesian product function [15]:

$$cp :: Ord\ a \Rightarrow [Set\ a] \rightarrow Set\ [a]$$
$$cp = List.foldr\ cross\ (singleton\ [])$$
$$\textbf{where}\ cross\ s\ ts = [(a : t) \mid a \leftarrow s, t \leftarrow ts]$$

This definition of *mapRel* via Cartesian product may appear somewhat contrived, but there's good reason why it is the chosen definition of *mapM* in the Haskell libraries. It is, in fact, the only monotonic definition of map on relations possible, as proven by [16], page 100.

Like *imageRel*, the *mapRel* operator preserves identities and composition.

Note furthermore that

$$cp \circ map\ singleton = singleton$$

and

$$concat \circ map\ (toList \circ singleton) = concat \circ map\ wrap = id$$

so in fact we have

$$imageRel\ (singleton \circ f) = mapRel\ (singleton \circ f)$$

In words, for relations that are functions the two definitions of *imageRel* and *mapRel* coincide.

4 Relations as finite sets of pairs

Of course there is a yet more common view of finite relations, namely as sets of pairs:

$$\textbf{type}\ a \nleftrightarrow b = Set\ (a, b)$$

Here is a definition of the empty relation:

$$emptyRel :: (Ord\ a, Ord\ b) \Rightarrow a \nleftrightarrow b$$
$$emptyRel = empty$$

The *domain* of a relation is the set of elements occurring as the left-hand component of a pair:

$$domain :: (Ord\ a, Ord\ b) \Rightarrow (a \nleftrightarrow b) \rightarrow Set\ a$$
$$domain = Set.map\ fst$$

Similarly the *range* is the set of values on the right-hand side:

```
range :: (Ord a, Ord b) ⇒ (a ↔ b) → Set b
range = Set.map snd
```

The *carrier* of a relation is simply the union of its domain and range:

```
carrier :: Ord a ⇒ (a ↔ a) → Set a
carrier r = domain r `union` range r
```

There are obvious conversions from finite relations to relations as set-valued functions, and vice versa:

```
apply :: (Ord a, Ord b) ⇒ (a ↔ b) → (a ↔ b)
apply abs a = [b | (a', b) ← abs, a' ≡ a]


finite :: (Ord a, Ord b) ⇒ Set a → (a ↔ b) → (a ↔ b)
finite as r = [(a, b) | a ← as, b ← r a]
```

5 Least fixpoints

Many useful relations can be defined as a least fixpoint of a monotonic function between relations. To compute such a least fixpoint, we iterate from the empty relation:

```
lfp :: (Ord a, Ord b) ⇒ ((a ↔ b) → (a ↔ b)) → (a ↔ b)
lfp rec = fix emptyRel
  where fix x = let y = rec x
                in  if x ≡ y
                    then y
                    else fix y
```

Let's now consider an example recursion programmed in terms of a least fixpoint, namely transitive closure of a graph. A graph is just a relation over vertices:

```
type Vertex = Int
type Graph = Vertex ↔ Vertex
```

The transitive closure is computed by finding any vertex w reachable from v, and then recursively finding all vertices reachable from w:

```
closure :: Graph → Graph
closure
  = lfp ∘ c
    where c g rec = finite (carrier g) f
      where f v = do w ← apply g v
                     insert w (apply rec w)
```

Note that this works fine even for cyclic graphs — unlike the built-in recursion of languages like Haskell, here we truly compute the least fixpoint in the subset order.

When the universe of values is finite, recursive computations on relations always terminate. As an extreme example, consider the least fixpoint of the identity function:

$$
\begin{aligned}
&\mathit{trivial} :: \mathit{Graph} \\
&\mathit{trivial} = \mathit{lfp}\ \mathit{id}
\end{aligned}
$$

This gives the least solution of the equation

$$
\begin{aligned}
&\mathit{trivial} :: \mathit{Graph} \\
&\mathit{trival} = \mathit{trivial}
\end{aligned}
$$

and so it evaluates to the empty relation.

6 Recursive aggregates

Now let us apply this to the computation of the depth of each vertex in a graph. A *leaf* in a graph is a node with no outgoing edges. The *depth* of a vertex is the length of the longest path to a leaf. If the graph is a tree, this corresponds to the well-known notion of tree depth. If there are cycles in the graph, there are vertices that do not have a depth.

To illustrate, here is an example graph, which happens to be a tree:

$$
\begin{aligned}
&\mathit{example1} :: \mathit{Graph} \\
&\mathit{example1} = \mathit{fromList}\ [(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)]
\end{aligned}
$$

We intend to define the depth operation so that

$$
\mathit{depth}\ \mathit{example1} = \mathit{fromList}\ [(0,2),(1,1),(2,1),(3,0),(4,0),(5,0),(6,0)]
$$

This corresponds to the usual definition of depth on trees. Let us now consider a graph with a cycle:

$$
\begin{aligned}
&\mathit{example2} :: \mathit{Graph} \\
&\mathit{example2} = \mathit{fromList}\ [(0,1),(0,2),(1,3),(1,4),(2,2),(2,4)]
\end{aligned}
$$

In this case it is clear that vertices 0 and 2 do not have a depth, because there is no maximum distance from a leaf. Consequently we have

$$
\mathit{depth}\ \mathit{example2} = \mathit{fromList}\ [(1,1),(3,0),(4,0)]
$$

The correct way to define the notion of depth uses the monotonic mapping of relations over the set of children:

$$
\begin{aligned}
&\mathit{depth} :: \mathit{Graph} \to (\mathit{Vertex} \nleftrightarrow \mathit{Int}) \\
&\mathit{depth} \\
&\quad = \mathit{lfp} \circ d \\
&\quad\ \ \mathbf{where}\ d\ g\ \mathit{rec} = \mathit{finite}\ (\mathit{carrier}\ g)\ f \\
&\qquad\qquad \mathbf{where}\ f\ v \mid \mathit{null}\ \mathit{cs} \quad\ = \mathit{singleton}\ 0 \\
&\qquad\qquad\qquad\qquad \mid \mathit{otherwise} = [m+1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{depths} \leftarrow \mathit{mapRel}\ (\mathit{apply}\ \mathit{rec})\ \mathit{cs} \\
&\qquad\qquad\qquad\qquad\qquad\qquad , m \leftarrow \mathit{listMax}\ \mathit{depths} \\
&\qquad\qquad\qquad\qquad\qquad\qquad] \\
&\qquad\qquad\qquad\qquad \mathbf{where}\ \mathit{cs} = \mathit{toList}\ (\mathit{apply}\ g\ v)
\end{aligned}
$$

It would be incorrect to define the same using relational image instead of monotonic relational map:

```
depth_wrong :: Graph → (Vertex ↔ Int)
depth_wrong
  = lfp ∘ d
    where d g rec = finite (carrier g) f
      where f v | null cs     = singleton 0
                | otherwise = [m + 1
                               | depths ← imageRel (apply rec) cs
                               , m ← listMax depths]
                  where cs = toList (apply g v)
```

By accident, this happens to return the right result on *example1*, but it diverges on *example2*, while the correct result is returned by *depth*. It is instructive to see how this computation goes wrong, by tracing the steps of the fixpoint iteration of *depth_wrong*:

```
[[],
  [(3,0),(4,0)],
  [(1,1),(2,1),(3,0),(4,0)],
  [(0,2),(1,1),(2,2),(3,0),(4,0)],
  [(0,3),(1,1),(2,3),(3,0),(4,0)],
  [(0,4),(1,1),(2,4),(3,0),(4,0)],
  [(0,5),(1,1),(2,5),(3,0),(4,0)],
  [(0,6),(1,1),(2,6),(3,0),(4,0)],
  [(0,7),(1,1),(2,7),(3,0),(4,0)],
  ...]
```

One might argue that the mere failure to find a least fixpoint in the presence of infinite relations is not that catastrophic, so let us now examine an example where use of a non-monotonic recursion actually terminates but leads to an unwanted result.

It is easily checked that the least fixpoint of a monotonic function f is the same as that of *identify f*, where

```
identify f r = r `union` f r
```

So let us modify the non-monotonic recursion to throw in identity elements, modifying the definition of d to:

```
where d g rec = finite (carrier g) f `union` rec
```

When applied to a simple tree-shaped graph

```
example3 :: Graph
example3 = fromList [(0,1),(0,2),(2,3),(2,4)]
```

the result is

```
fromList [(0,1),(0,2),(1,0),(2,1),(3,0),(4,0)]
```

Which is a fixpoint, but not the functional relation we expected for *depth*.

7 Aggregates in Datalog

Datalog is a programming language for defining relations; it has a clean least-fixpoint semantics, and a simple computational model based on iteration from the empty relation.

Because of these straight forward foundations, it lends itself to extremely aggressive automatic optimisation.

There exists a rich literature on the topic of adding aggregates to Datalog, so that recursion through aggregates is permitted [9–14] . The difficulty observed by all these authors is that the obvious semantics of aggregates is not monotonic in the subset order. In terms of our earlier discussion, they all define aggregates incorrectly in terms of *imageRel* operation we encountered above. Their proposed solutions are to significantly complicate the semantics, and to introduce complex evaluation mechanisms.

We can only speculate why this incorrect definition became so prevalent, but we believe it is because traditionally, there is no clear separation between the *range* of an aggregate and the *terms* being aggregated.

That separation is made explicit in the Eindhoven Quantifier Notation. A generic aggregate operation in the Eindhoven Quantifier Notation [17, 18] takes the form

$$agg\ (Dummy\ d \mid range\ d \mid term\ d)$$

In words, we collect all values of d that satisfy the *range*, and then we apply *term* to each of those values. Problems arise when the term can be a partial or even a non-deterministic operation, rather than a total function. As we have seen,

$$imageRel\ term = mapRel\ term \quad \text{provided } term \text{ is a total function}$$

but that equation does *not* hold in general.

Summarising the above discussion, we are proposing to give the quantifier notation the following semantics (the function is named after Doaitse's famous uncle Edsger W. Dijkstra, who pioneered the Eindhoven Quantifier Notation):

$$\begin{aligned}
&ewd :: (Ord\ a, Ord\ b, Ord\ c, Ord\ d) \Rightarrow \\
&\quad ([a] \leftrightarrow b) \to (c \mathrel{\dot{\leftrightarrow}} d) \to (d \leftrightarrow a) \to (c \leftrightarrow b) \\
&ewd\ agg\ range\ term\ c = \mathbf{let}\ ds = toList\ (apply\ range\ c) \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \ mapRel\ term\ ds \gg\!= agg
\end{aligned}$$

Using *ewd*, our previous definition of *depth* is rewritten

$$\begin{aligned}
&depth2 :: Graph \to (Vertex \mathrel{\dot{\leftrightarrow}} Int) \\
&depth2 \\
&\quad = lfp \circ d \\
&\quad\ \ \mathbf{where}\ d\ g\ rec = finite\ (carrier\ g)\ f \\
&\quad\qquad \mathbf{where}\ f\ v \mid Set.null\ (apply\ g\ v) = singleton\ 0 \\
&\qquad\qquad\qquad\quad \mid otherwise\qquad\qquad\ = Set.map\ (+1) \\
&\qquad\qquad\qquad\qquad\qquad\quad\ (ewd\ listMax\ g\ (apply\ rec)\ v)
\end{aligned}$$

A subtle point to address now is why we convert the range to a list in the definition of *ewd*. This is important when using other aggregations than *max* or *min*, which are sensitive to the multiplicity of a value, such as taking the sum. This ensures that, for example, we can correctly count elements by summing:

$$\begin{aligned}
&ewd\ (singleton \circ sum)\ range\ (singleton \circ const\ 1) \\
&\equiv \\
&singleton \circ size \circ apply\ range
\end{aligned}$$

The traditional semantics of the quantifier notation in query languages is to use relational image instead of relational map:

$$
\begin{array}{l}
sql :: (Ord\ a, Ord\ b, Ord\ c, Ord\ d) \Rightarrow \\
\quad ([a] \leftrightarrow b) \to (c \mathrel{\dot{\leftrightarrow}} d) \to (d \leftrightarrow a) \to (c \leftrightarrow b) \\
sql\ agg\ range\ term\ c = \mathbf{let}\ ds = toList\ (apply\ range\ c) \\
\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ imageRel\ term\ ds \ggg agg
\end{array}
$$

As we have seen, this unfortunately precludes recursive aggregates with a simple least-subset semantics. Traditional non-monotonic aggregates are expressible in terms of the monotonic semantics proposed here, because according to Doaitse's uncle, we have the *trading rule*:

$$
\begin{array}{l}
agg\ (Dummy\ d \mid range\ d \mid term\ d) \\
= \\
agg\ (Dummy\ d, A\ a \mid range\ d\ and\ a = term\ d \mid a)
\end{array}
$$

In general the trading rule is only valid in the traditional semantics, however. Because in the right-hand side of the trading rule the term is a total function (the identity), it follows that

$$
sql\ agg\ range\ term = ewd\ agg\ (term \circ range)\ singleton
$$

In words, we have not lost any expressive power by deviating from the traditional semantics of aggregates: everything expressible by *sql* is expressible by *ewd* but not *vice versa*.

8 Shortest paths in a graph

By now some functional programmers may be wondering what all the fuss is about: why would anyone use relational image in defining the semantics of aggregates, and not relational map? To answer that question, let's have a look at an example familiar from any introduction to algorithm design, namely the length of a shortest path in a directed graph:

$$
\begin{array}{l}
shortest\ (v, w) = \\
\quad \mathbf{if}\ v \equiv w \\
\quad \mathbf{then}\ 0 \\
\quad \mathbf{else}\ 1 + min\ (Vertex\ u \mid (v, u) \in edges \mid shortest\ (u, w))
\end{array}
$$

Usually there is some remark about the minimum of the empty set being ∞, so that *shortest* is in fact a total function. In the above recursion we emphatically need to read the quantifier notation in the traditional manner. So we have a recursive aggregate, well-known to any computer scientist, and it is not monotonic in the subset order.

The above recursion can still be used for fixpoint iteration, provided we employ a different lattice of approximations, namely total functions with the reverse pointwise numerical order. This observation is the basis for earlier works on recursive aggregates in Datalog, for example [10]. They show how this view fits with the *well-founded semantics* for Datalog, which approximates every relation both from above and from below (as opposed to the simple least fixpoint semantics in the subset order, which only approximates from below). Unfortunately no efficient implementation method is known for the well-founded semantics (you need to compute greatest as well as least fixpoints), so this was not a route open to us and the above recursion cannot be directly executed, even with our monotonic proposed semantics for aggregates. It is possible to describe the usual algorithms in Datalog with these new aggregates.

9 Conclusion

It's all really simple, but no, I didn't think about it on the bike ride from Houten.

Acknowledgements

Arthur Baars assisted with tidying up the Haskell code. My colleagues at Semmle, in particular Pavel Avgustinov, Julian Tibble and Lex Spoon patiently listened to earlier half-baked versions of these ideas, and helped get them into shape.

References

1. Gallaire, H., Minker, J.: Logic and Databases. Plenum Press (1978)
2. Ceri, S., Gottlob, G., Letizia, T.: What you always wanted to know about Datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering **1**(1) (1989) 146–166
3. de Moor, O., Gottlob, G., Furche, T., Sellers, A.J.: Datalog Reloaded. Volume 6702 of Lecture Notes in Computer Science. Springer (2011)
4. de Moor, O., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Sereni, D.: Keynote address: .QL for source code analysis. In: Source Code Analysis and Manipulation, IEEE (2007) 3–16
5. de Moor, O., Sereni, D., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Tibble, J.: .QL: Object-oriented queries made easy. In: Generative and Transformational Techniques in Software Engineering II. Volume 5235 of Lecture Notes in Computer Science., Springer (2007) 78–133
6. Sereni, D., Avgustinov, P., de Moor, O.: Adding magic to an optimising Datalog compiler. In: SIGMOD International Conference on Management of Data, ACM (2008) 553–566
7. de Moor, O., Sereni, D., Avgustinov, P., Verbaere, M.: Type inference for Datalog and its application to query optimisation. In: SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), ACM (2008) 291–300
8. Schäfer, M., de Moor, O.: Type inference for Datalog with complex type hierarchies. In: SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM (2010) 145–156
9. Kemp, D.B., Stuckey, P.J.: Semantics of logic programs with aggregates. In: International Symposium on Logic Programming '91, MIT Press (1991) 387–401
10. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive databases. Journal of Computer and System Sciences **54**(1) (1997) 79–97
11. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and stable semantics of logic programs with aggregates. Theory and Practice of Logic Programming **7**(3) (2007) 301–353
12. Pelov, N., Truszczynski, M.: Semantics of disjunctive programs with monotone aggregates — an operator-based approach. In: 10th International workshop on Non-Monotonic Reasoning. (2004) 327–334
13. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artificial Intelligence **175**(1) (2011) 278–298
14. Alviano, M., Calimeri, F., Faber, W., Leone, N., Perri, S.: Unfounded sets and well-founded semantics of answer set programs with aggregates. Journal of Artificial Intelligence Research **42** (2011) 487–527
15. Danvy, O., Spivey, J.M.: On Barron and Strachey's cartesian product function. In: International Conference on Functional Programming. (2007) 41–46
16. Carboni, A., Kelly, G.M., Wood, R.J.: A 2-categorical approach to geometric morphisms, I. Technical Report 89-19, Department of Mathematics, the University of Sydney, NSW 2006 (1989)

17. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science. Springer (1990)
18. Kaldewaij, A.: The Derivation of Algorithms. Prentice Hall (1990)

Doaitse – Mijn Mentor in de Nieuwe Wereld

Toen ik op 1 oktober 1990 als OIO bij Doaitse begon, was ik eigenlijk natuurlijk nog een beetje groen. Ik vond mezelf heel wat, net ingenieur, net een nieuwe baan, een appartement gekocht, in een nieuwe stad. Beetje Randstad. Doaitse was mijn officiële mentor (promotor), maar stiekem ook een beetje mijn officieuze mentor (surrogaatvader). Ik herinner me de talloze keren dat ik hem te voet, met z'n grote passen, probeerde bij te benen. Een kleine jongen naast zijn vader?

Hoe kwam ik in Utrecht? Door mijn echte vader. Hij wist dat ik was afgestudeerd op "the derivation of an editor" en zag een advertentie voor een NWO onderzoek rond "language based editors". Editors dus. Zodoende. Ik werd aangenomen. Tegelijkertijd met Frank. Mijn toekomstige kamergenoot.

Eerste baan, werk op de universiteit. Niet meer als student. Maar aan de andere kant van de streep. Wellicht dat die streep in Utrecht minder aanwezig was dan op de TU Eindhoven, maar ik was verrast door de toegankelijkheid van "de professoren". Maar toch, als ik er een paar op een rijtje moet zetten, zeg Van Leeuwen, Meertens, Overmars, Swierstra, dan had ik toch wel de meest joviale.

Ik herinner me Doaitses toespelingen op de "hormonen die hem tegemoet gierden" als hij op de AIO/OIO verdieping de klapdeuren openmaakte. Of Annerie, de afdelingssecretaresse. Daar kon Doaitse wonderlijk goed mee omgaan.

Natuurlijk gingen we samen op (dienst)reis. Doaitse kon dan in geuren en kleuren (en uren, want zo'n treinreis naar Schloss Dagstuhl duurt best lang) uitweiden over het Eurovisie song festival (in Dublin) en met name de jurken. Zag ik mijn prof van een andere kant. Op de zomerschool in Praag had hij een achteraf terras ontdekt waar we elke avond ons tekort aan vitamines aanvulden door wafels met fruit (en een beetje slagroom) te eten.

Maar dit is natuurlijk de officiële mentorrol. Wie schetst mijn verbazing toen ik op een zondagmorgen uit mijn bed werd gebeld (ik had pas net een vriendinnetje, vandaar) door Doaitse. Voor zijn officieuze taak. Hij had kaartje voor mij en mijn vriendin. Vredenburg. Concert. Of ik daar zin in had. Zo ja, dan kwam hij ze zelfs wel even brengen (met de trein, dat spreekt). En zo ging ik naar mijn eerste Vredenburgconcert. Later heeft Doaitse het nog een keer goed gemaakt met een ander kaartje, want die eerste keer was (vond ik toen) verschrikkelijk. Een open vleugel waar de pianist met een theelepeltje snaren aan tokkelde. De mogelijk opmerkelijkste stap van Doaitse was toen hij zijn huis "uitleende". Hij ging op vakantie. Of ik dan op zijn huis wilde oppassen. Ik mocht er gewoon de hele tijd wonen; het was toch fietsafstand tot de (R)UU. Ik was verbaasd. Maar voor Doaitse de gewoonste vraag van de wereld. En ik begon te wennen aan deze gewone dingen van Doaitse.

Ik weet niet goed welke "officieuze mentor" actie ik als hoogtepunt moet noemen. Feit is dat Doaitse mij en mijn toekomstge vrouw (Debbie) voor een etentje heeft uitgenodigd. Ik was al gewend aan Doaitse, maar Debbie vond dit etentje een hoog keuringsgehalte hebben. En ze weet nu overigens nog dat ze brie met pruimensaus kreeg. Dus voor mij staat een andere actie op de eerste plaats. De laatste keer dat Doaitse mij verbaasde was toen hij vroeg (ik meen dat ik al gepromoveerd, en dus weg van de (R)UU was) of ik niet op vakantie

wilde in zijn huis in Ameland. Daar zijn we al met al drie keer geweest.

Nou vermoed ik dat die Ameland uitnodiging ook op Doaitse's business aanleg gebaseerd was. Ik herinner me Doaitse's exposé over Omo Power, en zijn vergelijk tussen Omo's research budget en Omo's marketing budget. Ook de stoere actie van Doaitse om zijn business class ticket Cochabamba om te zetten (budgetair neutraal, dat spreekt) in een handvol touristclass tickets.

Maar dichter bij huis heeft hij (via zijn broer) een Dikke van Dale voor mij geregeld, tegen familieprijs. En via zijn vrouw heeft hij een bijbaan typesetting met LaTeX voor mij geregeld bij Kluwer rechtswetenschappen. En natuurlijk heeft hij bijgedragen aan mijn baan na de OIO plaats op de (R)UU: bij het Philips NatLab (mijn jongensdroom, als techneutje groot gebracht rond Eindhoven).

Wat mij ook altijd helder bij is gebleven is dat een groep Bolivianen naar Utecht kwam. Ze zouden mee fietsen op een of ander bedrijfsuitje. Eén van de Nederlands collega's zei dat hij dat niet leuk vond, al die Bolivianen erbij. Waarop Doaitse tegenwierp "dan vind ik jou ook niet meer leuk". Zo simpel kan het.

Ten slotte, dankzij Doaitse, heb ik een handgeschreven brief van de E.W. Dijkstra. Of ik daar trots op moet zijn weet ik niet, met fragmenten als "schamen voor de wartaal", "klein gillertje", "de tekst barst van dit soort mysteries". Maar ik ben inmiddels voldoende groot gegroeid.

Doaitse, bedankt!

Maarten Pennings

ps Weet je het antwoord nog van mijn moeder op jouw vraag over het verschil tussen Chateaubriand en Tournedos?

Circularity and Lambda Abstraction

Olivier Danvy[1], Peter Thiemann[2], and Ian Zerny[1⋆]

[1] {danvy,zerny}@cs.au.dk
Department of Computer Science, Aarhus University
[2] thiemann@acm.org
Institut für Informatik, Universität Freiburg

Abstract. In this tribute to Doaitse Swierstra, we present the first transformation between lazy circular programs à la Bird and strict circular programs à la Pettorossi. Circular programs à la Bird rely on lazy recursive binding: they involve circular unknowns and make sense equationally. Circular programs à la Pettorossi rely on the inductive construction of functions and their eventual application: they involve no circular unknowns and make sense operationally. Our derivation connects these equational and operational approaches: given a lazy circular program à la Bird, we decouple the circular unknowns from what is done to them, which we lambda-abstract with functions. The circular unknowns then become dead variables, which we eliminate. The result is a strict circular program à la Pettorossi. This transformation is reversible: given a strict circular program à la Pettorossi, we introduce circular unknowns as dead variables, and we apply the functions to them. The result is a lazy circular program à la Bird.
We illustrate the two transformations by mapping an algebraic construct to an isomorphic one with new leaves, reading a binary number as suggested by Knuth, and backpatching.

1 Introduction

You do not have to think operationally:
you can reason equationally
about your programs.
– S. Doaitse Swierstra

I prefer call by value
to call by name
because it is more predictable.
– Mitchell Wand

One of the wonderful things about functional programming is that we can both reason about programs equationally (regarding what they do) and think about them operationally (regarding how they do it). Take circular programs, for example. This technique was invented by Richard Bird in the early 1980's to eliminate multiple traversals of data [1]. It was then phrased operationally by Alberto Pettorossi in the late 1980's [2]. In this hommage to Doaitse Swierstra, we present what we believe to be the first transformation between circular programs à la Bird and circular programs à la Pettorossi. Each of the following sections illustrates this transformation.

Prerequisites and notation: It seems safe to assume that the reader knows what a circular program is, but we nevertheless explain the concept in Sec. 2. Likewise, in a structurally recursive function that visits an inductive data structure, we readily say that the arguments are "inherited" and the result is "synthesized," in reference to Knuth's attribute grammars [3]. Throughout, our programming language of discourse is Haskell.

⋆ Ian Zerny is a recipient of the Google Europe Fellowship in Programming Technology, and this research is supported in part by this Google Fellowship.

2 Minimum list

In his original article [1], Bird illustrated circular programming with a function mapping a binary tree of integers into an isomorphic binary tree where all the integers were replaced by the smallest integer in the given binary tree. Rather than composing two functions – one to compute the smallest integer in the given tree, and one to re-traverse the given tree to construct an isomorphic tree – Bird calculated a 'circular' function that ostensibly traverses the given tree once and yet gets the job done.

In this section, we treat in detail a simplified version of Bird's original function that operates not on binary trees of integers, but on lists of integers. We first present the circular function in the style of Bird (Sec. 2.1), and illustrate its working equationally (Fig. 1). We then present the circular function in the style of Pettorossi (Sec. 2.2), and illustrate its working equationally (Fig. 2). We finally present our transformation to map either function to the other (Sec. 2.3).

2.1 A Bird-style circular program

The circular program à la Bird is a function that uses lazy local recursion to circularly refer to the minimal element of the input list. In the function below, `m` is the circular unknown: it is circularly defined using local recursion and it is unknown in the body of `visit`:

```
minlist_RB :: [Int] -> [Int]
minlist_RB xs = ys
  where
    (m, ys) = visit m xs
    visit :: Int -> [Int] -> (Int, [Int])
    visit m [] =
      (maxBound, [])
    visit m (x : xs) =
      let (m', ys) = visit m xs
      in  (min x m', m : ys)
```

Fig. 1 displays successive unfoldings of this function when it is applied to the list `[3,1,4]`. These unfoldings illustrate equationally the resolution of the circular unknown `m`. The modified part is boxed at each step.

2.2 A Pettorossi-style circular program

The circular program à la Pettorossi is a function that uses lambda abstraction to refer to the minimal element of the input list. In the function below, `m` is an abstracted unknown: it is lambda-abstracted in the body of `visit` and it is subsequently instantiated when the lambda-abstraction is applied:

```
minlist_AP :: [Int] -> [Int]
minlist_AP xs = ys m
  where
    (m, ys) = visit xs
    visit :: [Int] -> (Int, Int -> [Int])
    visit [] =
      (maxBound, \m -> [])
    visit (x : xs) =
      let (m', ys) = visit xs
      in  (min x m', \m -> m : ys m)
```

Fig. 2 displays successive unfoldings of this Pettorossi-style program to the input list `[3,1,4]`. These unfoldings illustrate equationally the resolution of the abstracted unknown `m`. The modified part is boxed at each step.

```
minlist_RB_0 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = visit m (3 : 1 : 4 : [])
-- unfold the underlined call to visit
minlist_RB_1 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = visit m (1 : 4 : [])
              in (min 3 n, m : ys)
-- unfold the underlined call to visit
minlist_RB_2 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = let (n, ys) = visit m (4 : [])
                            in (min 1 n, m : ys)
              in (min 3 n, m : ys)
-- unfold the underlined call to visit
minlist_RB_3 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = let (n, ys) = let (n, ys) = visit m []
                                          in (min 4 n, m : ys)
                            in (min 1 n, m : ys)
              in (min 3 n, m : ys)
-- unfold the underlined call to visit
minlist_RB_4 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = let (n, ys) = let (n, ys) = (maxBound, [])
                                          in (min 4 n, m : ys)
                            in (min 1 n, m : ys)
              in (min 3 n, m : ys)
-- unfold the underlined let expression
minlist_RB_5 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = let (n, ys) = (min 4 maxBound, m : [])
                            in (min 1 n, m : ys)
              in (min 3 n, m : ys)
-- unfold the underlined call to min
minlist_RB_6 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = let (n, ys) = (4, m : [])
                            in (min 1 n, m : ys)
              in (min 3 n, m : ys)
-- unfold the underlined let expression
minlist_RB_7 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = (min 1 4, m : m : [])
              in (min 3 n, m : ys)
-- unfold the underlined call to min
minlist_RB_8 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = (1, m : m : [])
              in (min 3 n, m : ys)
-- unfold the underlined let expression
minlist_RB_9 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = (min 3 1, m : m : m : [])
-- unfold the underlined call to min
minlist_RB_10 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = (1, m : m : m : [])
-- unfold the underlined where expression, which is recursive
minlist_RB_11 (3 : 1 : 4 : []) = 1 : 1 : 1 : []
```

Fig. 1. Successive equational unfoldings of `minlist_RB [3,1,4]`

```
minlist_AP_0 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = visit (3 : 1 : 4 : [])
-- unfold the underlined call to visit
minlist_AP_1 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = visit (1 : 4 : [])
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined call to visit
minlist_AP_2 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = let (n, ys) = visit (4 : [])
                            in  (min 1 n, \m -> m : ys m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined call to visit
minlist_AP_3 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = let (n, ys) = let (n, ys) = visit []
                                          in  (min 4 n, \m -> m : ys m)
                            in  (min 1 n, \m -> m : ys m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined call to visit
minlist_AP_4 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = let (n, ys) = let (n, ys) = (maxBound, \m -> [])
                                          in  (min 4 n, \m -> m : ys m)
                            in  (min 1 n, \m -> m : ys m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined let expression
minlist_AP_5 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = let (n, ys) = (min 4 maxBound, \m -> m : (\m -> []) m)
                            in  (min 1 n, \m -> m : ys m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined call to min and the underlined beta-redex
minlist_AP_6 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = let (n, ys) = (4, \m -> m : [])
                            in  (min 1 n, \m -> m : ys m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined let expression
minlist_AP_7 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = (min 1 4, \m -> m : (\m -> m : []) m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined call to min and the underlined beta-redex
minlist_AP_8 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = (1, \m -> m : m : [])
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined let expression
minlist_AP_9 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = (min 3 1, \m -> m : (\m -> m : m : []) m)
-- unfold the underlined call to min and the undernined beta-redex
minlist_AP_10 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = (1, \m -> m : m : m : [])
-- unfold the underlined where expression, which is not recursive
minlist_AP_11 (3 : 1 : 4 : []) = (\m -> m : m : m : []) 1
-- unfold the underlined beta-redex
minlist_AP_12 (3 : 1 : 4 : []) = 1 : 1 : 1 : []
```

Fig. 2. Successive equational unfoldings of `minlist_AP [3,1,4]`

2.3 From either style to the other

The last steps of Fig. 1 and Fig. 2 differ in two key aspects:

1. In the substitution step from `minlist_RB_10` to `minlist_RB_11`, the where expression is recursive for the Bird-style program, whereas for the Pettorossi-style program, the where expression is non-recursive in the substitution step from `minlist_AP_10` to `minlist_AP_11`.
2. The instantiation of `m` takes place during resolution for the Bird-style program, i.e., `minlist_RB_11` is the final result, whereas for the Pettorossi-style program, the instantiation of `m` takes place subsequently, i.e., `minlist_AP_11` is not the final result.

The key distinction is that `m` is a circular unknown (i.e., a variable that is declared recursively) in the Bird-style program whereas it is not in the Pettorossi-style program.

Using this observation, given a circular program à la Bird, we decouple the circular unknown from what is done to it, which we represent as a function (e.g., initially as the identity function). Consequently, the unknown becomes a dead variable in Bird's program and it is our observation that omitting this dead variable gives exactly a circular program à la Pettorossi – in the present case, the same program as in Sec. 2.2.

Here is the minlist program à la Bird where we have marked (with a trailing underscore) all of the variables that depend on the circular unknown:

```
minlist_mark :: [Int] -> [Int]
minlist_mark xs = ys_
  where
    (m_, ys_) = visit m_ xs
    visit :: Int -> [Int] -> (Int, [Int])
    visit m_ [] =
      (maxBound, [])
    visit m_ (x : xs) =
      let (m', ys_) = visit m_ xs
      in  (min x m', m_ : ys_)
```

We decouple the circular unknown in two steps. Here is the decoupling (as a pair) of the inherited variables that depend on the circular unknown: the first component of the pair is the circular unknown, and the second component is what is done to the circular unknown:[3]

```
minlist_inher :: [Int] -> [Int]
minlist_inher xs = ys_
  where
    (m, ys_) = visit (m, id) xs
    visit :: (Int, Int -> Int) -> [Int] -> (Int, [Int])
    visit m_ [] =
      (maxBound, [])
    visit (m_ @ (m, f)) (x : xs) =
      let (m', ys_) = visit m_ xs
      in  (min x m', f m : ys_)
```

Here is the abstraction of the synthesized variables that depend on the circular unknown:

```
minlist_synth :: [Int] -> [Int]
minlist_synth xs = ys m
  where
    (m, ys) = visit (m, id) xs
    visit :: (Int, Int -> Int) -> [Int] -> (Int, Int -> [Int])
    visit m_ [] =
      (maxBound, \m -> [])
    visit (m_ @ (m, f)) (x : xs) =
      let (m', ys) = visit m_ xs
      in  (min x m', \m -> f m : ys m)
```

[3] In the interest of generality, we fully decouple the inherited variables, here `m_` of `visit`, even though nothing is done to them. In general, the inherited variables could be changed. Such is the case in Knuth's program for reading binary numbers (Sec. 4).

In `visit`, the variable `m` is dead (i.e., it is unused) and the variable `f` solely denotes the identity function (i.e., nothing is done to `m`). Thus, we strike out the first and we symbolically apply the second. The result is precisely the circular program à la Pettorossi from Sec. 2.2. This program is inductively defined and can be transliterated to an eager programming language such as ML.

Overall, each of the steps in the transformation from Bird style to Pettorossi style can be reversed.

In the following sections, we successively consider Bird's original circular program mapping a tree of numbers to an isomorphic tree of the least of these numbers (Sec. 3); Knuth's original attribute grammar for reading a binary number (Sec. 4); and conversely, how to express backpatching as a circular program (Sec. 5).

3 Minimum tree

Let us turn to Bird's circular function that given a binary tree of integers, maps it to an isomorphic binary tree where all the integers were replaced by the smallest integer in the given binary tree. The tree data type is declared as follows:

```
data BTree a = Leaf a | Node (BTree a) (BTree a)
```

For example, a tree such as `tin` below should be mapped to the tree `tout`:

```
tin  = Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
tout = Node (Leaf 1) (Node (Leaf 1) (Leaf 1))
```

Bird's circular program is the starting point of the transformation:

```
mintree_RB :: BTree Int -> BTree Int
mintree_RB t = t'
  where
    (m, t') = visit m t
    visit :: Int -> BTree Int -> (Int, BTree Int)
    visit m (Leaf n)
      = (n, Leaf m)
    visit m (Node l r)
      = let (lm, lt) = visit m l
            (rm, rt) = visit m r
        in (min lm rm, Node lt rt)
```

As in Sec. 2, we decouple the circular unknown (here `m`) from what is done to it. Here is the mintree program where we have marked (with a trailing underscore) all of the variables that depend on the circular unknown:

```
mintree_mark :: BTree Int -> BTree Int
mintree_mark t = t'_
  where
    (m_, t'_) = visit m_ t
    visit :: Int -> BTree Int -> (Int, BTree Int)
    visit m_ (Leaf n)
      = (n, Leaf m_)
    visit m_ (Node l r)
      = let (lm, lt_) = visit m_ l
            (rm, rt_) = visit m_ r
        in (min lm rm, Node lt_ rt_)
```

Here is the decoupling (as a pair: the circular unknown and what is done to it, represented as a function) of the inherited variables that depend on the circular unknown:

```
mintree_inher :: BTree Int -> BTree Int
mintree_inher t = t'_
  where
    (m, t'_) = visit (m, id) t
    visit :: (Int, Int -> Int) -> BTree Int -> (Int, BTree Int)
    visit (m, f) (Leaf n)
```

```
      = (n, Leaf (f m))
    visit m_ (Node l r)
      = let (lm, lt_) = visit m_ l
            (rm, rt_) = visit m_ r
        in (min lm rm, Node lt_ rt_)
```

Here is the abstraction of the synthesized variables that depend on the circular unknown:

```
mintree_synth :: BTree Int -> BTree Int
mintree_synth t = t' m
  where
    (m, t') = visit (m, id) t
    visit :: (Int, Int -> Int) -> BTree Int -> (Int, Int -> BTree Int)
    visit (m, f) (Leaf n)
      = (n, \m -> Leaf (f m))
    visit m_ (Node l r)
      = let (lm, lt) = visit m_ l
            (rm, rt) = visit m_ r
        in (min lm rm, \m -> Node (lt m) (rt m))
```

In `visit`, the variable `m` is dead (i.e., it is unused) and the variable `f` solely denotes the identity function (i.e., nothing is done to `m`). Thus, we eliminate both. The result is Pettorossi's one-pass solution to the problem [2]:

```
mintree_AP :: BTree Int -> BTree Int
mintree_AP t = t' m
  where
    (m, t') = visit t
    visit :: BTree Int -> (Int, Int -> BTree Int)
    visit (Leaf n)
      = (n, \m -> Leaf m)
    visit (Node l r)
      = let (lm, lt) = visit l
            (rm, rt) = visit r
        in (min lm rm, \m -> Node (lt m) (rt m))
```

Again, the Pettorossi-style program can be transliterated to an eager programming language. Also, each of the transformation steps is reversible.

4 Reading numbers

Knuth's seminal article on attribute grammars [3] starts with an example of a grammar that gives a precise definition of binary notation for numbers. The grammar generates the language with words of the form *num.mantissa* where both *num* and *mantissa* are bit strings. The attribution of the grammar computes, in an attribute v of the start symbol, the numeric value of the binary notation.

The interest in Knuth's second attribution of the grammar arises from a non-trivial attribute dependency that requires a two-pass traversal for evaluating all attributes. Thus, the "obvious" translation of the attribute grammar into a functional program results in a circular program of a slightly more general form as in the preceding examples.

But to start from the beginning, a slightly rephrased and simplified version of this example is sufficient to demonstrate the transformation. The simplified grammar only generates bit strings and the attribution considers the generated bit string as the binary notation for a number and computes it. The underlying grammar has terminals `O` and `I` representing the low bit and the high bit and three non-terminals *Bit*, *Bits*, and S, the start symbol. All non-terminals have a synthesized attribute v; *Bit* and *Bits* have an inherited attribute p; and *Bits* has an additional synthesized attribute l. The intention is that the attribute v computes the value of the respective bit or bit string relative to its starting position p — a *Bit* at position p counts 2^p. The attribute l computes the length of a bit string.

Fig. 3 contains the productions of the grammar and their attribution. The attribution rules use the notation suggested by Johnsson for referring to the attribute occurrences, with

$$
\begin{array}{lll}
S & \to \mathit{Bits} & S \uparrow v = \mathit{Bits} \uparrow v \\
 & & \mathit{Bits} \downarrow p = \mathit{Bits} \uparrow l - 1 \\
\mathit{Bits} & \to \varepsilon & \mathit{Bits} \uparrow v = 0 \\
 & & \mathit{Bits} \uparrow l = 0 \\
\mathit{Bits} & \to \mathit{Bit}\ \mathit{Bits}_1 & \mathit{Bits} \uparrow v = \mathit{Bit} \uparrow v + \mathit{Bits}_1 \uparrow v \\
 & & \mathit{Bit} \downarrow p = \mathit{Bits} \downarrow p \\
 & & \mathit{Bits}_1 \downarrow p = \mathit{Bits} \downarrow p - 1 \\
 & & \mathit{Bits} \uparrow l = \mathit{Bits}_1 \uparrow l + 1 \\
\mathit{Bit} & \to \texttt{O} & \mathit{Bit} \uparrow v = 0 \\
\mathit{Bit} & \to \texttt{I} & \mathit{Bit} \uparrow v = 2\hat{}(\mathit{Bit} \downarrow p)
\end{array}
$$

Fig. 3. Attribute grammar for interpreting numbers in binary notation

```
data Bit = O | I

digitval :: Int -> Bit -> Int
digitval p O = 0
digitval p I = 2 ^ p

dec n = n - 1
inc n = n + 1
```

Fig. 4. Auxiliary definitions for interpreting numbers in binary notation

↑ indicating synthesized attributes and ↓ indicating inherited ones [4]. His translation of an attribute grammar into a lazy program interprets a non-terminal as a function from its inherited attributes to its synthesized attributes. Applying this technique to the attribute grammar in Fig. 3 leads to the circular program `lexnum_RB` shown in Fig. 5 which uses the definitions in Fig. 4.

In Fig. 4, the function `digitval` is the interpretation of the *Bit* non-terminal. It has one (inherited) argument and one (synthesized) result. Its `Bit`-typed argument serves to distinguish the two production rules for *Bit*. The functions `dec` and `inc` stand for the decrement and increment operations in the attribution.

In Fig. 5, the function `lexnum_RB` is the transliteration of the attributions of the non-terminals S and *Bits*. There is no choice in the first equation of the `where` block because S has one production.

The function `visit` is the interpretation of the *Bits* non-terminal. Its first argument holds the inherited attribute p and its second determines the production. It computes a pair comprising the synthesized attributes.

As in Sec. 2 and Sec. 3, we decouple the circular unknown (here `l`) from what is done to it. Here is the lexnum program where we have marked (with a trailing underscore) all of the variables that depend on the circular unknown:

```
lexnum_mark :: [Bit] -> Int
lexnum_mark bs = v_
  where
    (l_, v_) = visit (dec l_) bs
    visit :: Int -> [Bit] -> (Int, Int)
    visit p_ [] =
      (0, 0)
    visit p_ (b:bs) =
      let (l, v_) = visit (dec p_) bs
      in (inc l, v_ + digitval p_ b)
```

```
lexnum_RB :: [Bit] -> Int
lexnum_RB bs = v
  where
    (l, v) = visit (dec l) bs
    visit :: Int -> [Bit] -> (Int, Int)
    visit p [] =
      (0, 0)
    visit p (b:bs) =
      let (l, v) = visit (dec p) bs
      in  (inc l, v + digitval p b)
```

Fig. 5. Bird-style circular program for interpreting numbers in binary notation

Here is the decoupling (as a pair: the circular unknown and what is done to it, represented as a function) of the inherited variables that depend on the circular unknown:

```
lexnum_inher :: [Bit] -> Int
lexnum_inher bs = v_
  where
    (l, v_) = visit (l, dec) bs
    visit :: (Int, Int -> Int) -> [Bit] -> (Int, Int)
    visit (p, f) [] =
      (0, 0)
    visit (p, f) (b:bs) =
      let (l, v_) = visit (p, dec . f) bs
      in  (inc l, v_ + digitval (f p) b)
```

In contrast to Sec. 2 and Sec. 3, something is actually being done to the circular unknown at call time (i.e., it is decremented).

Here is the abstraction of the synthesized variables that depend on the circular unknown:

```
lexnum_synth :: [Bit] -> Int
lexnum_synth bs = v l
  where
    (l, v) = visit (l, dec) bs
    visit :: (Int, Int -> Int) -> [Bit] -> (Int, Int -> Int)
    visit (p, f) [] =
      (0, \p -> 0)
    visit (p, f) (b:bs) =
      let (l, v) = visit (p, dec . f) bs
      in  (inc l, \p -> v p + digitval (f p) b)
```

In `visit`, the variable `p` is dead (i.e., it is unused) so we eliminate it. The result is a Pettorossi-style program:

```
lexnum_AP :: [Bit] -> Int
lexnum_AP bs = v l
  where
    (l, v) = visit dec bs
    visit :: (Int -> Int) -> [Bit] -> (Int, Int -> Int)
    visit f [] =
      (0, \p -> 0)
    visit f (b:bs) =
      let (l, v) = visit (dec . f) bs
      in  (inc l, \p -> v p + digitval (f p) b)
```

Again, this Pettorossi-style program can be transliterated to an eager programming language. Also, each of the transformation steps is reversible.

5 Backpatching

Backpatching is a traditional compilation technique [5]. It applies in a compiler that generates code using symbolic labels for jump targets in the first place. The main argument for

```
type Lab = Int
type Addr = Int
type Env = Map Lab Addr
data Source =
  SSUB | SPSH Int | SJMP Lab | SCJP Lab | SLAB Lab
data Target =
  TSUB | TPSH Int | TJMP Addr | TCJP Addr
```

Fig. 6. Type definitions for backpatching

```
backpatch_AP :: [Source] -> [Target]
backpatch_AP ss = f env
  where
    (env, f) = collect 0 ss
    collect :: Addr -> [Source] -> (Env, Env -> [Target])
    collect a [] =
      (empty, \env -> [])
    collect a (si : sis) =
      let (env, f) = collect (a + tsize si) sis in
      case si of
        SSUB   -> (env, \env -> TSUB : f env)
        SPSH i -> (env, \env -> TPSH i : f env)
        SJMP l -> (env, \env -> TJMP (env ! l) : f env)
        SCJP l -> (env, \env -> TCJP (env ! l) : f env)
        SLAB l -> (insert l a env, f)
```

Fig. 7. Pettorossi-style program for backpatching

doing so is to simplify the code generator and in particular subsequent transformation steps that may insert or remove instructions, or even rearrange entire code blocks.

Once the transformations are finished with the code, the symbolic labels have to be transformed to addresses. A typical approach is to traverse the code and build an environment that maps symbolic labels to addresses. A second pass uses this environment to resolve all jump addresses.

Backpatching is a one-pass implementation technique for this two-pass transformation. In this pass, label definitions are entered in the environment as they occur. For a label use, there are two possibilities, either the label is already defined (a backward reference) in which case the target address can be inserted directly, or the label is not yet defined (a forward reference), in which case this use-before-definition is registered in the environment. In general, there may be multiple uses before a definition is found, so the environment entry for a label may contain a list of unresolved targets. When defining a label that already has some uses, the unresolved targets are visited and overwritten with the address, hence the name backpatching.

This traditional algorithm is imperative. However, a purely functional implementation of backpatching can be given by abstracting the generation of the program with absolute addresses from the environment as illustrated by the code in Fig. 7. It relies on the datatype definitions given in Fig. 6. They define a type `Source` of source instructions for a stack machine, which are subtraction, push a constant, jump, conditional jump, and label definition. Both jump instructions refer to symbolic labels of type `Label`. The type `Target` of target instructions comprises subtraction, push, jump, and conditional jump, where the latter two refer to addresses. As an auxiliary definition, an environment of type `Env` is a mapping from labels to addresses. Furthermore, there is a function `tsize :: Source -> Int` that computes the size of the generated target code for each source instruction.

The function `collect` in Fig. 7 traverses the list of source instructions and keeps track of the current target address in its first argument `a`. It returns a pair of an environment and

a function that expects an environment and returns the target code. The transformation removes the label instruction `SLAB l` so that its target size is `0`.

This function is written in Pettorossi style: it is defined inductively and can be expressed directly in an eager programming language. We use it as the starting point to demonstrate the reverse transformation from Pettorossi style to Bird style, taking the reverse sequence of steps.

The first step is to introduce the abstracted value as a circular unknown of visit, here with the formal parameter `env1` of visit:

```
backpatch_circ :: [Source] -> [Target]
backpatch_circ ss = f env
  where
    (env, f) = collect 0 ss env
    collect :: Addr -> [Source] -> Env -> (Env, Env -> [Target])
    collect a [] env1 =
      (empty, \env -> [])
    collect a (si : sis) env1 =
      let (env, f) = collect (a + tsize si) sis env1 in
      case si of
        SSUB   -> (env, \env -> TSUB : f env)
        SPSH i -> (env, \env -> TPSH i : f env)
        SJMP l -> (env, \env -> TJMP (env ! l) : f env)
        SCJP l -> (env, \env -> TCJP (env ! l) : f env)
        SLAB l -> (insert l a env, f)
```

Next, we specialize the synthesized abstractions with respect to `env1`. Each abstraction passes `env1` unchanged and so `f env1` becomes the lazily constructed result, `ts`:

```
backpatch_synth :: [Source] -> [Target]
backpatch_synth ss = ts
  where
    (env, ts) = collect 0 ss env
    collect :: Addr -> [Source] -> Env -> (Env, [Target])
    collect a [] env1 =
      (empty, [])
    collect a (si : sis) env1 =
      let (env, ts) = collect (a + tsize si) sis env1 in
      case si of
        SSUB   -> (env, TSUB : ts)
        SPSH i -> (env, TPSH i : ts)
        SJMP l -> (env, TJMP (env1 ! l) : ts)
        SCJP l -> (env, TCJP (env1 ! l) : ts)
        SLAB l -> (insert l a env, ts)
```

Since there are no inherited abstractions we are done and the result is a circular backpatching program à la Bird.

6 Related work

Kuiper and Swierstra [6] noted the connection between attribute grammars and functional programs around the same time as Johnsson [4]. They note that rewrite rules employing tuples and derivations of circular programs can be conveniently expressed using attribute grammars. They define two mappings from attribute grammars to functional programs. One of the mappings can give rise to multiple traversals of a data structure whereas the other yields circular programs that traverse the structure at most once, but require lazy evaluation.

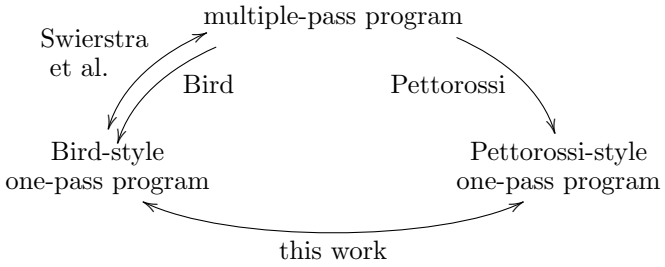
Fernandes and Saraiva [7] transform circular programs into efficient, strict and deforested, multiple-traversal programs by using attribute grammars-based techniques, in particular ordered attribute grammars [8]. This approach draws on ideas from earlier work by Saraiva, Swierstra, and Kuiper [9]. Both works rely on intricate analysis techniques for attribute grammars. Their transformations yield strict, but potentially multi-pass programs.

Fernandes and coworkers [10] suggest a strictification transformation for circular programs. Their transformation is based on a dependency analysis to discover the circularity.

They naively split the circular call, which returns a tuple, into several ones with each computing only one component. Specialization of these calls yields independent, non-circular definitions. The resulting programs are suitable for strict evaluation, but they are not in Pettorossi-style in that they might require multiple passes over the input data.

7 Conclusion

In the course of the 1980's, Bird and Pettorossi investigated how to calculate programs that traverse their input only once [1, 2]:



In his joint work on circular attribute grammars, Swiestra has shown how to transform Bird-style programs into multiple-pass programs and vice versa [6, 9].

In this tribute to Doaitse Swierstra, we have shown how to connect Bird-style and Pettorossi-style programs. A Bird-style program inductively extends a circular unknown until this extended unknown can be solved. We decouple the circular unknowns in a Bird-style program from what is inductively done to them, which we represent with functions. The circular unknowns then become dead variables. Symbolically applying the functions to the circular unknowns gives back a Bird-style program, and eliminating the dead variables gives a Pettorossi-style program. A Pettorossi-style program is therefore one that computes over differences, and so could be considered as the derivative of a Bird-style program.

Acknowledgments: We are grateful to Julia Lawall for comments on a preliminary version of this article, and to Doaitse Swierstra for many years of academic camaraderie. We wish him many happy returns!

References

1. Bird, R.S.: Using circular programs to eliminate multiple traversals of data. Acta Informatica **21** (1984) 239–250
2. Pettorossi, A.: Derivation of programs which traverse their input data only once. In Cioni, G., Salwicki, A., eds.: Advanced Programming Methodologies. Academic Press, Limited, San Diego, CA, USA (1989) 165–184
3. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **2**(2) (1968) 127–145
4. Johnsson, T.: Attribute grammars as a functional programming paradigm. In Kahn, G., ed.: Functional Programming Languages and Computer Architecture. Number 274 in Lecture Notes in Computer Science, Portland, Oregon, Springer-Verlag (September 1987) 154–173
5. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Second edn. Pearson Education, Inc. Addison-Wesley, London, United Kingdom (2006)
6. Kuiper, M.F., Swierstra, S.D.: Using attribute grammars to derive efficient functional programs. Technical Report RUU-CS-86-16, University of Utrecht (August 1986)

7. Fernandes, J.P., Saraiva, J.: Tools and libraries to model and manipulate circular programs. In Ramalingam, G., Visser, E., eds.: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2007), Nice, France, ACM Press (January 2007) 102–111
8. Kastens, U.: Ordered attributed grammars. Acta Informatica **13** (1980) 229–256
9. Saraiva, J., Swierstra, S.D., Kuiper, M.F.: Strictification of computations on trees. In: 3th Latin-American Conference on Functional Programming, CLaPF'99. (1999)
10. Fernandes, J.P., Saraiva, J., Seidel, D., Voigtländer, J.: Strictification of circular programs. In Khoo, S.C., Siek, J., eds.: Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2011), Austin, Texas, ACM Press (January 2011) 131–140

Doaitse Heeft Altijd Gelijk

Wilke Schram
Manager Beheer en Bestuurssecretaris Informatica 1999 - 2010

7 april 2013

Doaitse heeft altijd gelijk. En niet zozeer in de zin van "De baas heeft altijd gelijk, ook als hij geen gelijk heeft", maar gewoon omdat hij echt gelijk heeft. Dat komt door zijn exceptionele waarnemings- en analytisch vermogen. Ik sprak SDS (prof. dr. S.D. Swierstra) niet of nauwelijks over het vakgebied Informatica noch over Softwaretechnologie. Veelal ging het over zaken als bestuurlijke processen en de praktische dingen des levens en alles wat daar tussen zit, zoals strategische thema's van het departement, de faculteit en de UU: wat hebben we aan dit soort zwaartepuntzaken als de printers het niet doen?

Tijdens het rectoraat van prof. Willem Gispen werden enige tijd zogenaamde Kapittelavonden georganiseerd waarvoor (alle) UU hoogleraren werden uitgenodigd: bij een hapje en een drankje werden dan belangrijke relevante bestuurlijke thema's besproken. Deze bijeenkomsten vormden dé gelegenheid voor hoogleraren ("de werkvloer") hun mening te geven over actuele zaken. SDS heeft er van genoten. Echter, de serie gesprekken heeft niet al te lang geduurd: het CvB kreeg waarschijnlijk meer goede "suggesties" dan het kon verwerken. Terzijde: ik ben er altijd voorstander van geweest dat we dit model op facultair niveau blijvend overnemen: de kwaliteit van het bestuur en de legitimiteit van de bestuurderen kan er alleen maar bij winnen.

Zoals gezegd zit de kracht van SDS in de nauwkeurigheid van waarneming en de kracht van zijn analyses, niet in de bestuurlijke realiteitszin. Een echt autorisatie- en authenticatiesysteem voor alle automatiseringssystemen op de UU zal veel van de huidige ICT-problemen oplossen maar het betekent ook dat een fors aantal ego's een stevige deuk zullen oplopen en dat kan natuurlijk niet. En wat te denken van de vastgelopen UU-website "Webpresence" en het bijbehorende CMS: er was een alternatief, gratis nog wel, met meer functionaliteit, maar de UU importeerde een nest veel te dure KPMG-ers die met een bijna onwerkbaar product kwamen.

Toch wil er ook wel eens geluisterd worden naar SDS getuige dit citaat van ongeveer 10 jaar geleden: "Waarmee mijn stelling weer eens bewezen is dat, als ik moegestreden het hoofd in de schoot leg, juist dan de zaken in beweging komen".

Het mag waar zijn dat SDS altijd gelijk heeft, dat is gelijk een straf. Iemand die weet hoeveel leed en zinloze arbeid voorkomen kan worden, die bestuurders de weg wijst en die zijn ideen welwillend ziet aangehoord, maar tegelijk merkt dat er niks verandert, komt vroeg of laat voor de vraag te staan: "Wie is hier nu gek?"

Wat ik nog het meeste in SDS bewonder, is het feit dat als hij zijn doortimmerde visie één keer heeft uitgelegd aan de bestuurderen, hij daarmee kan volstaan. Ik zou deze lieden na een maand nog eens opbellen: "En, heb je wat kunnen doen met mijn suggesties?"

Gelukkig kunnen ze nog altijd terecht bij SDS. De bestuurderen moeten nu echter even doorreizen naar Tynaarlo, alwaar Doaitse en Agnes ze met open armen en een kopje thee zullen verwelkomen. Een goed gesprek is nooit weg.

Janboerenfluitjes | Zekerheidzoeken

Beste Doaitse,

We zagen elkaar voor het eerst op een IFIP-WG2.1 meeting in Warschau. Je was daar genodigd als observer. Toen ik tegen een van de leden opmerkte dat je mij in uiterlijk en optreden sterk deed denken aan ons toen nog medelid Edsger Dijkstra, werd me verteld dat je een neef was (omkesizzer in het Fries). Als persoonlijkheid was Edsger voor mij even fascinerend, als moeilijk in omgang: Fries in hart en nieren. Eerlijk. Halstarrig. Idealistisch bij het fanatieke af. Maar ook vrolijk en vol (soms venijnige) humor.

De werkgroep was toen, onder leiding van Aad van Wijngaarden in de laatste fase van de definitie van ALGOL68. Prominente leden waaronder Edsger, konden niet meegaan in het uiteindelijk resultaat. Ze vonden dat programmeurs behoefte hadden aan ander gereedschap dan grotere en zg "betere" programmeertalen.

De NATO Conference on Software Engineering in Garmisch (oktober 1968) bracht nieuw inzicht in de problematiek, constructie en implementatie van betrouwbare software. Edsger en zijn geestverwanten schreven een "Minority-Report", verlieten WG2.1 en vormden (Oslo 20-22 juli 1969) binnen IFIP een nieuwe werkgroep WG2.3: "Programming Methodology".

Ik herinner mij deze splitsing als tamelijk dramatisch. In hoge mate ook als een breuk tussen "vader" en "zoon". Edsger's carriere was begonnen als wetenschappelijk programmeur in het Mathematisch Centrum in Amsterdam bij Aad van Wijngaarden. Hij werkte er onder meer aan een eerste compiler voor Algol60. Ik ben getuige geweest van enkele woordenwisselingen: Edsger was volstrekt overtuigd van zijn gelijk. Aad was emotioneel aangeslagen, maar in zekere zin ook de promotor van Edsger's rebellie: hij zag in dat het voor computer-programmeurs vrijwel onleesbare "Report on the Algorithmic Language ALGOL 68" een "Informal Introduction" nodig had. Charles Lindsey en ik kregen deze opdracht van WG2.1.

Op de meeting in Warschau raakte ik onder de indruk van je enthousiasme en stellingname op het gebied van programmeertalen. Je was toen assistent professor aan de universiteit in Groningen. Ik attendeerde je op de open plaats voor een professoraat in Utrecht. Jouw reactie: "Dan werken we niet alleen in dezelfde vakgroep, maar ook in hetzelfde onderzoeksgebied. Als ik het goed zie, zal ik dan formeel als hoogleraar je 'baas' zijn. Jij bent 22 jaar ouder dan ik. Weet je zeker dat we dat aankunnen?" Ik heb toen zoiets geantwoord als: "We zijn beide Friezen. Die maken inderdaad makkelijk ruzie. Maar terwijl jij je leeropdracht vervult, schuif ik richting pensioen. Niet bang zijn Doaitse, je raakt me vanzelf kwijt."

Tot zover de voorgeschiedenis van onze, vooral in het begin, toch best wel moeilijke collegialiteit. Je start in Utrecht was niet makkelijk. Jij zocht in je werk vóór alles zekerheid, en was daarin, meer dan een omke sizzer vooral een omke folgjer.

Tot onze grote schrik gaf je een college over de beginselen van het programmeren na enkele weken in een vrijwel leeg lokaal. Je 2e jaars studenten knapten volledig af op die beginselen. Enkelen overwogen zelfs te switchen naar een andere studie. We hebben de zaak

kunnen redden, mede door jouw begrip van de penibele situatie. De wijze waarop je een nederlaag accepteerde heeft me toen gewoon ontroerd. Maar het was niet de laatste keer dat tussen ons een belangrijk verschil van inzicht aan het licht kwam. Daarmee hebben 22 jaar in redelijke harmonie samengewerkt.

Edsger kwalificeerde mijn stijl van programmeren ooit als "op z'n janboerenfluitjes". Denigrerend, maar niet helemaal onterecht. Jij vond, evenals je omke, dat een computerprogramma idealiter regel voor regel bewijsbaar-correct moet zijn. Mijn toentertijd achttien jaar ervaring als programmeur was voornamelijk intuïtief. Hier gingen onze wegen uit elkaar, en dat zijn ze gebleven. Van de "janboerenfluitjes" wil ik de fluitjes wel handhaven – daar kun je muziek mee maken.

In mijn Utrechtse tijd met jou speelde ik nog uren per dag piano. In de jaren daarvóór had ik als wiskunde-leraar bijverdiend met het schrijven van muziekrecensies. Muziek was en is voor mij een fenomeen dat uit intuïtie ontstaat. Je beredeneert geen muziek – je maakt het. En als je improviseert, de ene dag weer anders dan de andere. De titel van mijn bijdrage aan dit vriendschapsboek confronteert twee attitudes. Uit een zekere balorigheid blijf ik bij de door Edsger gesmade betekenis van de "janboeren". Dat is een persoonlijke kwalificatie waar ik destijds best wel om kon lachen. Edsger was in elk geval altijd verbaal duidelijk, maar wel geestig. Jouw attitude, "zekerheid zoeken" neem ik ook voor wat die is, al werd het me nooit duidelijk welke zekerheid je nastreefde. Een metafysische kan het niet geweest zijn. Metafysica lijkt mij een ver-van-jouw-bed show. Kort samengevat: we wilden hetzelfde, maar langs totaal verschillende wegen die we van elkaar in het "waarheen" niet altijd helder zagen.

De persoonlijke contacten tussen ons en onze gezinnen zijn steeds erg goed geweest. Ik herinner me wel de moeite die ik had met jullie beslissing zelf geen kinderen op deze morbide wereld te zetten: in de omgang met neefjes en nichtjes was volstrekt duidelijk dat jullie voortreffelijke ouders geweest zouden zijn voor die nooit geboren kinderen. Ook hierin had ik moeite met je rechtlijnigheid. Als vader van zes kinderen en grootvader van kleinkinderen weet ik dat in onze onvolmaakte wereld door opvoeding misschien toch iets te veranderen is. Maar wie ben ik jou in dezen nog eens te bekritiseren in een weloverwogen beslissing. Jij kunt mij met al die nakomelingen ongefundeerd optimisme verwijten.

Dit is een van die momenten in mijn verhaal waarin ik uitdrukking wil geven aan mijn grote erkentelijkheid dat je nooit in onze 22 jaren durende samenwerking op je strepen bent gaan staan. Ik ben er me van bewust dat ik voor jou geen gemakkelijke collega was. Mijn interesses en belangen doorkruisten nogal eens die van jou. Mijn nog jaren durende WG2.1 participatie was nooit een punt – wel soms mijn slordige financiele afwikkeling ervan, maar die werd altijd soepel rechtgetrokken.

Professioneel gingen onze wegen pas echt uit elkaar toen de cursor de macht op het beeldscherm overnam. Na een paar jaar konden alle janboeren met een computer spelen zonder programma's te moeten schrijven. Software verdween letterlijk achter de schermen. Hardware uit Silicon Valley maakte in enkele jaren de productie van computers ook nog eens zó goedkoop dat het apparaat voor iedereen betaalbaar werd. De wereld veranderde daarmee totaal. Korte tijd heb ik me gevoeld als iemand die zijn fraaie rijtuig – door het ALGOL paard over alle zandwegen getrokken – overbodig zag worden: in de kortst mogelijke keren kwamen er nieuwe verharde wegen. Daarover kon je nu snel en geriefelijk, na een paar rijlessen, je doel bereiken in een – het woord komt in deze context moeilijk uit de toetsen – automobiel.

Prachtig, maar dat vooral voor de latere generaties. De mijne moest emplooi zien te vinden in een nieuwe werk-omgeving waarin slechts enkele van de beste paarden en rijtuigen bleven bestaan omdat ze nog nodig waren: om het gebeuren achter de schermen in goede

banen te leiden.

Je hebt me alle ruimte gegeven. Totaan m'n pensioen kon ik me gaan orienteren in het door diezelfde computer-revolutie actueler geworden gebied van kunstmatige intelligentie. Het was een merkwaardige sensatie student te zijn in een door mezelf ingestelde afstudeerrichting. Voor de medewerkers die ik daarin vond, nam je – na hun academische credentials te hebben geverifieerd – zonder problemen de volle verantwoordelijkheid. En nu moet het toch maar even worden vastgelegd: mijn enige credentials waren een paar MO-akten Wiskunde.

Er zijn nogal wat coryfeeën onze universiteiten binnengekomen met vergelijkbare minimale achtergrond. Verreweg de meesten voltooiden voordien of achteraf nog een academische studie. En promoveerden. Niet ik – maar dat kon ook niet: Informatica was in Nederland in mijn tijd nog "in statu nascendi". Wat niet wegneemt dat het mij (en enkelen met mij) m'n hele (ook daardoor rare) loopbaan heeft dwars gezeten. Of het jou ook heeft dwarsgezeten, weet ik niet. Het is in elk geval nooit tussen ons ter sprake geweest.

Die laatste jaren waren heel prettig. Het was goed studeren bij de knappe jongelui die onze studierichting verrijkten.

Epiloog

Met mijn pensionering nam ik radicaal afstand van mijn vak. Iedereen die er door zijn of haar leeftijd wordt uitgegooid, raad ik dat aan. Begeef je niet in een zogenaamd "welverdiende rust". Als je niet levensmoe bent, grijp dan de kans er een nieuwe wending aan te geven. Voor mij lag die in Argentinie en Chili waar mijn halfbroer woont. Als het hier winter is, is het daar zomer. Het noorden is tropisch, het zuiden polair. Een aantal jaren geleden doorkruiste ik dat enorme stuk Zuidamerika van noord naar zuid in een meestal niet zo goede gehuurde auto.

Vanuit het mooie stadje Salta in de provincie JuJuy reed ik naar de grens met Bolivia. Ik wilde met de "Tren de las Nubes" naar boven – de meest spectaculaire treinrit ter wereld. Het is er niet van gekomen. Ik kon het niet inpassen in mijn reis-schema. Wel een fantastische rit door de droge bedding van een rivier tot vlakbij de grens met Bolivia. Met een huur-auto kon ik niet verder. Ik wandelde door de sneeuw tot de grenspost.

Wetend dat jij een stuk universiteit uit Nederland naar Bolivia had gebracht, heb ik daar – kleumend van de kou en een beetje achter adem van de hoogte – aan je gedacht. Het was een van de weinige keren dat ik echt last had van mijn leeftijd. Niet door de kou en de hoogte, maar door het besef dat ik, een jaar of 12 jonger, met jou een hoog-begaafd volk had kunnen begeleiden in de nieuwe wereld.
Doaitse, ik hoop dat je me – als er een "seminar" is in de Nieuwe Kerk (1656) in Den Haag waar je aan deelneemt – een e-mail stuurt. Ik woon er vlak naast.

Tot ziens,
Sietse

From:	Lidwien van de Wijngaert <lidwien@xs4all.nl>
Subject:	Liber Amicorum
Date:	May 30, 2013 1:00:04 PM GMT+01:00
To:	S. Doaitse Swierstra <doaitse@cs.uu.nl>

Beste Doaitse,

Ik wilde je om een gunst vragen want ik zit met een probleem en jij zegt meestal verstandige dingen.

Het volgende. Binnenkort gaat een vriend van mij, hoogleraar informatica, met emeritaat. Nu ben ik op de een of andere manier terecht gekomen op de mailing list voor zijn Liber Amicorum. Het probleem is dat ik niet weet wat op te sturen. Ik heb uiteraard geen kaas gegeten van zijn vakgebied. Iets inhoudelijks zit er dus niet in.

Nu ken ik toevallig ook het broertje van die hoogleraar (hij heeft ons ooit eens geïntroduceerd) en samen met die broer heb ik een paper geschreven over nano-technologie. Nu zouden we dat kunnen opsturen (is die broer ook gelijk klaar met zijn verhaal) maar een beetje vreemd is het natuurlijk wel. Nano. Ook lopen we het risico dat hij het stuk daadwerkelijk gaat lezen en dan aan alle kanten commentaar gaat leveren. Anderzijds: het is work-in-progress dus een beetje feedback kan geen kwaad.

Alternatief is dat ik een verhaaltje bij elkaar typ waarin staat hoe knap hij is maar daar wordt hij alleen maar verwaand van.

Vraag is dus: wat te doen?

Met groet,
Lidwien.

P.S. Zo volgt het stuk over nano-technologie, dan kun je even kijken of je denkt dat het zou passen in dat vriendenboekje.

Formation in the public debate about nanotechnology: how information does change knowledge but knowledge does not change attitude

Tsjalling Swierstra and Lidwien van de Wijngaert

May 7, 2013

Abstract

The diffusion of nanotechnology will strongly depend on the attitude of the public towards this new (and in some respects controversial) technology. As knowledge about nanotechnology is supposed to precede attitude formation, some governments organize public debates about nanotechnology to enhance knowledge about nanotechnology. This paper reports about the results of a Dutch project that was part of a 'National Dialogue' in 2010. The project applied a vignette approach to inform the Dutch public about possible ("hard" and "soft") impacts of nanotechnology. We present three findings. First: results of a survey among 1164 participants of the project show that attitude towards benefits of nanotechnology is independent from attitude towards its possible risks, although parties stressing benefits tend to downplay risks, and vice versa. Second, and more importantly: providing the public with information does change knowledge with regard to nanotechnology. People became more nuanced and developed a broader view of nanotechnology. However, increasing knowledge did not result in widespread optimism with regard to nanotechnology: a significant number of people still is ambivalent, skeptic or outright pessimistic vis à vis nanotechnology. Third: the attitude towards nanotechnology correlates with the way people think about the societal control over the development of nanotechnology. People primarily seeing benefits and people primarily seeing risks, have different ideas about how this control is and should be organized. The paper finishes by discussing some policy implications of these results.

1 Introduction

Nanotechnology is an emerging technology. It aims at manipulating matter on an atomic and molecular scale. Generally, nanotechnology deals with structures sized 1 to 100 nanometers and involves developing materials or devices possessing at least one dimension within that size. These materials and devices can be applied in a wide range of domains ranging from military applications to food preservation and from surveillance to health care. The successful application and diffusion of nanotechnology not only depends on the degree to which the development of these materials and devices is technically feasible. Success also depends on the degree to which the public is willing to accept this new technology (Rogers, 2003). Having learned from negative responses in society caused by other new technologies such as nuclear energy and GMO-technology, Dutch government realized the importance of involving the broader public in an early phase of technology development. With regard to nanotechnology Scheufele & Lewenstein (2005) show the importance of such an effort as people form

an opinion and attitude towards new technology even in the absence of relevant scientific or policy related information. From a more theoretical standpoint Ajzen (1991, 2001) argues that knowledge about the subject influences attitude towards the subject. Consequently the Dutch government created a program that aimed at developing awareness among the Dutch public about nanotechnology and its possible consequences (Nanopodium, 2009). The aim of the program was to initiate a public dialogue focusing on the ethical and societal aspects of nanotechnology applications. The Dutch initiative to involve the larger audience in identifying and discussing the impacts of an emerging technology like nanotechnology, is part of a larger trend in Western societies alternately referred to as Public Engagement, Awareness, Consultation, or Participation. (Irwin 2001, Jasanoff 2004, Rowe & Frewer 2005, Lee et al., 2005, Wynne 2006, Einsiedel 2008, Hessels et al 2009, MASiS Expert Group 2010, Sclove 2010). This paper presents the results of an exploratory research project within this program that investigated public attitude towards nanotechnology and the way in which nanotechnology develops. In that sense our project builds upon existing projects as presented by e.g. Cobb & Macoubrie (2004); Macnaghten, Kearnes and Wynne (2005), Scheufele et al. (2005), Lee, Scheufele & Lewenstein (2005); Currall, King, Lane, Madera & Turner (2006); Scheufele, Corley, Dunwoody, Shih, Hillback & Guston (2007). Our first research question therefore is:

1. (a) How did this twelve week-long information campaign influence the way people think of nanotechnology?
   (b) Can we distinguish different groups of people with regard to this judgment?

In addition to their general opinion about nanotechnology, respondents were asked to provide their opinion with regard to the project as well as to the way in which they perceive the way society influences the development of nanotechnology. Based on an analysis of these data, we will answer the second research question of this paper:

2. (a) What do people think about this way of science communication and about the way society influences technology development?
   (b) Can we distinguish different groups with regard to this judgment?

As a third goal of this paper we will focus on the question how judgments of benefits and costs of nanotechnology (Question 1) and judgments about the way in which it develops (Question 2) are interrelated. This correlation is relevant to establish whether social groups who think differently about the costs and benefits of nanotechnology, also have different ideas with regard to how technology should be allowed to develop. Therefore, the third research question is:

3. How are judgments about the desirability of nanotechnology related to judgments about this form of science communication, and about the way society influences the way nanotechnology develops?

In the next two sections we will describe the theoretical background of this project and the way the research was set up.

2 Risks and benefits, hard and soft impacts

The Dutch initiative to involve the larger audience in identifying and discussing the impacts of an emerging technology like nanotechnology, is part of a larger trend in Western societies

alternately referred to as Public Engagement, Awareness, Consultation, or Participation. All such initiatives are based on the assumption that the larger public should be informed about, and have something to say about, the technologies that will eventually come to co-shape their existences.

Such public debates tend to focus on a rather restricted set of possible impacts of technology, commonly referred to as "risk". "Risk" can be broadly defined as the chance that something undesirable will happen. However, in practice, the meaning of "risk" is much more restricted. Of all the values that can be threatened by a technology, policy makers and technology actors tend to focus exclusively on three: the chance that a technology will adversely affect our safety, our health, and/or the environment. And of course, citizens are also primarily interested in these values.

However, the GMO-debate has taught us that citizens also refer to less tangible values, like naturalness, a non-instrumental relation to plants and animals, respect for Creation, distributive justice, and so forth (Joly & Marris 2001, Marris 2001). This latter type of impacts is usually not recognized as "risk", and therefore never quite succeeds in gaining access to the public agenda. Policy and technology actors will sometimes acknowledge the importance of these other values, but as they don't know to handle these "soft impacts" (Wynne 1996, 2001, Swierstra et al 2009, Swierstra & te Molder 2011), in the end they tend to ignore them as being too "soft" to merit rational and public discussion.

While the relation between "soft" impacts and the evolution of public controversy is not linear and direct, experience and research (see for example Marris 2001) have shown that the dismissal of latent concerns about soft impacts may easily engender unexpected - at least for technologists and designers - outbursts of public discontent later in time. By then, repeated experiences and cumulated irritations have replaced the early, largely invisible and not necessarily negative concerns. When organizing a public dialogue on an emerging technology, it is therefore important not to reduce this dialogue a priori to risk issues, but to take care to also create space for the identification and discussion of soft impacts.

This is especially important in the case of nanotechnology, because here most attention is drawn towards the toxicity of (some) nanoparticles. In other words: the debate on nanotechnology gravitates towards risk issues, rather than towards soft impacts. This is to be expected, as it is hard to imagine at forehand how particular nanotechnology-enabled applications may affect established practices, values, relations, aspirationUs, norms, and so forth. This project in science communication aimed to redress this bias by presenting the public with information about both hard and soft impacts.

3 Research Method

3.1 A scenario approach towards emerging technologies

For a period of thirteen weeks the participants in the project were presented with vignettes that described possible consequences of the application of nanotechnology. A vignette is a story or scenario that describes a possible outcome of the application of nanotechnology. The richness of the story not only allows the researchers to evoke both hard and soft impacts to the public (Rossi & Nock, 1982; Rasmussen 2005, 229), but also highlight how actions are shaped by technologies, and how technological applications are shaped by human actions. In each of the thirteen weeks a different domain or issue was highlighted: nanotechnology and daily life, tinkering with nature, political and regulatory issues, risks of nano, nano and nature, the formable human, sports, military applications, nutrition, hygiene, smart environments, monitoring and medical applications. Figure 1 is a screenshot of the vignette

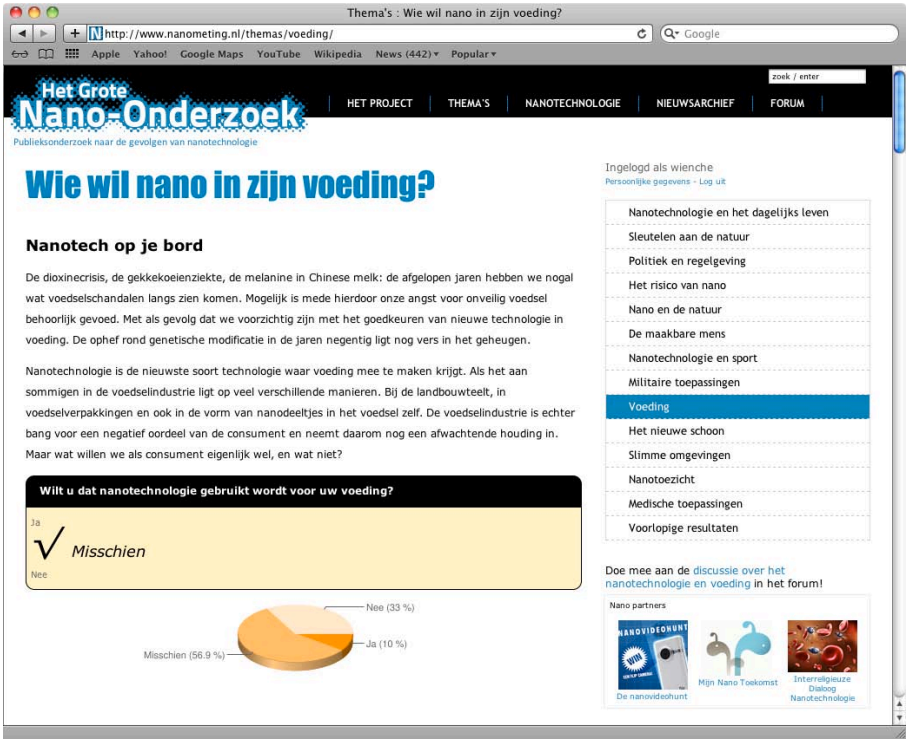


Figure 1: Vignette of week 9 on nutrition [in Dutch]

that was presented in week nine.

Every week a domain or issue was introduced through a quick sketch of possible applications of nanotechnology. After that, participants were asked whether they would like that application. Then, the vignette continued describing different hard and soft impacts of the application of nanotechnology. Unlike Cobb (2005) who deliberately framed nanotechnology in different contexts (risks and benefits), the vignettes in this study typically evoked both positive and negative impacts, or impacts that were morally ambiguous or uncertain. In that sense this paper resembles the approach of Weaver, Lively and Bimber (2009) who use frames regarding progress, regulation, conflict, and generic risk. The text box below is an English transcription of part of a vignette that was presented in a week dealing with nanotechnology and food:

As she isn't completely sure that the yoghurt isn't off, Eva studies the information on the carton. The sensor – popularly known as the nano-nose – signals red. That means: off, don't eat or drink. Eva ponders: there was this one time she drank from a carton of milk, only to notice later that according to the sensor it was off. But it hadn't tasted weird in any way, actually. Or had it? She couldn't remember for sure. As Eva starts to doubt she attempts to accesses the website of the producer, to find out what a "red dot" actually means. But while searching the Internet, she hits an Internet-forum vehemently discussing the reliability of the nano-nose. Critical consumers and even some contributors claiming to be scientists, argue that the science behind the nano-nose is based on a totally unscientific notion of "freshness". How could one objectively assess freshness? Others argue that the only function of the nano-nose is to prevent that the producer will be held accountable if consumers would turn sick.

The industry, in an attempt to better be safe than sorry, adheres to very wide margins. Therefore, it is totally okay to ignore the red signs of the nano-nose. Eva doesn't know who or what to believe anymore. She throws the still half full carton of yoghurt into the garbage can. But at the same moment she realizes that she didn't even inspect the yoghurt herself. Nor use her own nose.

In this vignette, we see how in real life hard and soft impacts are always intertwined. The nano-nose, a chip that informs the consumer whether a product is still edible and fresh, can be considered as a device that helps to diminish risk, as it is supposedly more reliable than a printed date on the wrapping. However, the vignette draws attention to the fact that in real life there exists no clear dividing line between edible and non-edible. So, hard impacts – getting ill from eating off food – may not be so "hard" as would seem at first. Furthermore, the new technology mediates, changes, Eva's behavior. Where she would previously have trusted her own nose, she now tends to obey the mechanical nose. As a result, she throws away food that may have been perfectly edible. This changed behavior, including the diminishing trust in one's own senses, is a soft impact. But this behavioral change, so the vignette implies, may have averse consequences for sustainability – a hard impact again.
After each paragraph participants were asked about their opinion with regard to nanotechnology.

```
Would you blindly trust a nano-sensor on your carton of milk?

O Yes
O Maybe
O No
```

In order to keep the discussion lively the participants immediately received feedback on how their responses compared to the responses of other participants. At the end of each week participants were asked if they had changed their mind as a consequence of the new information they had received. The results of this part of the research will not be discussed in this paper. Detailed results can however (in Dutch) be found in Authors (2011). For this paper, the whole first part of the project only served as a way to make sure that people could develop a well informed opinion with regard to nanotechnology and the way in which it should be introduced in society. In the rest of this paper we will focus on the results from the second part of this research.

3.2 Statistical Analysis

After thirteen weeks the participants in the project became respondents in the survey. In this week respondents were presented with two sets of items. The first set of items focused on opinions one might have with regard to nanotechnology, the second set of items focused on what people thought of the project itself as well as the way in which nanotechnology develops. So besides creating awareness about the possible hard and soft consequences of nanotechnology itself (as depicted in research question 1), the research project aimed at understanding how information can influence public opinion with regard to how nanotechnology develops (research question 2). By using a multi-item approach (as suggested by Binder, Cacciatore, Scheufele, Shaw, & Corley, 2010) we are able to indentify the dimensions that underlie the attitude that people have with regard to nanotechnology and the way it should be introduced in society. In this paper we will focus on the results of the data that was gathered in this last week of the project.

We performed an exploratory factor analysis on both set of items separately. We used principal components to extract the factors and used only factors with an Eigenvalue > 1 for further analysis. This meant that for the first set of items we found two factors and for the second set of items three factors were found. We used Varimax rotation to facilitate the interpretation of the factors. After that we computed Pearson's Correlations between the factor scores to investigate how the two sets of factors are interrelated.

4 Results

The following subsections will provide a discussion with regard to the factors that were found as well as how age, gender and education relate to the different factors. The third subsection will focus on how the two sets are correlated to each other. Before we describe the results we will first describe the people that participated in the project and research.

4.1 Participants

Participants in the project were recruited through online banners and radio interviews. In total 4854 people participated in the research. However, many of these people only read the scenarios for one or two weeks (see Figure 2). This means that we could use the results from 1164 people.

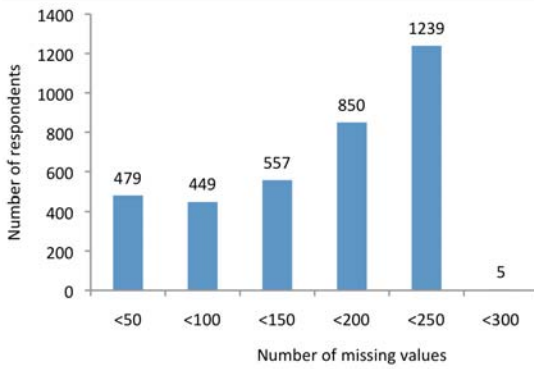


Figure 2: Missing values

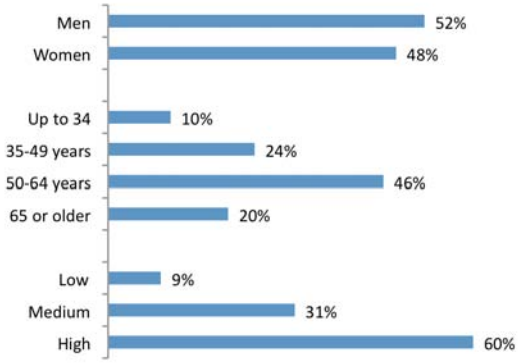


Figure 3: Characterization of the participants

The reason for this choice is that this is the only way that we can guarantee that all respondents have received the same information about nanotechnology in the course of the thirteen weeks. Another reason why we feel confident to use this group of people is that individual characteristics are well distributed (see Figure 3).

The sample has an almost equal number of men and women and people from different age groups and levels of education. As compared to the population we have an overrepresentation of higher educated people. Given the subject of the project this is of no surprise. To summarize the above, we only used the results from 1164 relatively old and highly educated Dutch men and women that participated all thirteen weeks in the project and the research in the final week.

4.2 Nanotechology

As was described in Section 3, we presented respondents with two sets of items. This section will describe the results for the first set of items regarding the attitude that people have towards nanotechnology. As can be derived from Table 1, the first factor analysis results in a two-factor solution (explaining 50% of the variance). The two factors can be can be interpreted easily:

- Nano I (Risks): this factor consist of items that use negative wording and explain the possible risks of the application of nanotechnology.
- Nano II (Advantages): this factor consist of positively worded items and explain the possibilities or the advantages of applying nanotechnology.

Factor I concerns 'Risks'. The items that are part of this factor generally involve the negative and hard impacts of the application of nanotechnology. The application of nanotechnology in or on ones body, health risks, the negative consequences for animals, people and the environment are part of this factor. In addition, Factor I also contains a number of negative soft impacts such as the dangers of war, violence and privacy violations. Also soft impacts such as the threat of unknown forces and life becoming tougher are part of this factor.

The second factor, 'Advantages', is formed by items that express possible positive and mostly soft outcomes of the application of nanotechnology. Nanotechnology will make life easier, will improve care, freedom of choices, possibilities for choice and nanotechnology. Hard and positive impacts that are part of this factor are the item concerning the notion that products that use nanotechnology can be used safely and the item that simply says that the application of nanotechnology has many advantages. Finally the factor contains two items that are neither hard or soft but describe the role that the government should play in terms of stimulation and in terms of rules and regulations.

On the whole people agreed slightly more with the items in Factor I (Risks) as compared to the items in Factor II (Advantages). This was maybe to be expected: people tend to agree more easily on what makes one unhappy than on what makes one happy. In sum, Factor I is related to 'Risks' and Factor II is related to 'Advantages'. Both factors consists of items that express hard and soft impacts although Factor I seems to be focusing more on the hard impacts whereas Factor II focuses more on the soft impacts. Secondly, one might wonder why the analysis did not result in a single factor solution. After all, the two factors seem opposites of each other. Although the majority of respondents combine many advantages with few risks or few advantages and many risks, there also is a significant number of people that combine many advantages with many risks and people that combine few advantages with few risks. Consequently, we can distinguish four groups of people based on their factor scores: optimists, pessimists, ambivalent people and skeptics (see Table 2). From the number of people in each of these groups we conclude that society as a whole consists of people with very different opinions with regard to the advantages and risks of nanotechnology.

Now that we have distinguished the four groups, it is possible to characterize the groups in terms of gender, age and education. Figure 4 shows the share of men, women, young and old and high and low educated people in each group. From the figure we can derive that we find a relatively large number of men, older people and higher educated people in the group of optimists. This finding implies that the group of decision makers from the worlds of science, industry and government (broadly speaking a group of higher educated, older men) should realize that their positive bias towards nanotechnology is not representative for

Table 1: Rotated Component Matrix for Analysis Nanotechnology

	Mean	S.D.	Nano I Risks	Nano II Advantages
Explained variance			25%	25%
I do not want nanotechnology in or in my body	3.9	1.7	0.53	
Nanotechnology is bad for the environment	3.9	1.3	0.54	
I do not want nanotechnology in or in my body	3.9	1.7	0.53	
Nanotechnology is bad for the environment	3.9	1.3	0.54	
It should be clear when nanotechnology is applied	6.1	1.2	0.61	
As a consequence of nanotechnology life will become tougher	4.4	1.4	0.64	
Nanotechnology invokes health risks	4.6	1.3	0.65	
Nanotechnology is a danger to animals and human beings	3.9	1.4	0.66	
The application of nanotechnology leads to war and violence	4.0	1.5	0.73	
Nanotechnology is a threat to privacy	4.8	1.5	0.75	
Nanotechnology gives power to unknown forces	5.2	1.3	0.77	
Nanotechnology is surrounded with sufficient rules and regulation	3.1	1.4		0.35
Products that use nanotechnology can be used safely	3.8	1.4		0.56
Nanotechnology will make society more fair	3.2	1.3		0.45
Nanotechnology makes life more controllable	4.4	1.3		0.64
Nanotechnology increases freedom of choice	4.1	1.3		0.65
Government should stimulate the use of nanotechnology	4.5	1.4		0.66
Nanotechnology will improve care	4.9	1.3		0.75
Nanotechnology makes life easier	4.8	1.2		0.77
Nanotechnology has many advantages	4.7	1.3		0.77
Extraction Method: Principal Component Analysis; Selection criterium: Eigenvalue > 1; Rotation Method: Varimax with Kaiser Normalization				

Table 2: Definition of four groups

Label	N	Nano Factor I Risks	Nanno Factor II Advantages
Optimist	349	Few risks, factors scores >0	Many advantages, factor scores >0
Pessimist	375	Many risks, factors scores <0	Few advantages, factor scores <0
Ambivalent	281	Many risks, factors scores <0	Many advantages, factor scores >0
Skeptic	159	Few risks, factors scores >0	Few advantages, factor scores <0

the whole society. This result is in line with Ho, Scheufele and Corley (2011) who conclude that compared with the experts, the general public judges nanotechnology as having greater risks and lesser benefits. The group of pessimists in contrast consists of a relatively large number of women, middle-aged people and lower educated people. So both in terms of their opinion with regard to nanotechnology as well as their characterization in terms of gender, age and education, optimists and pessimists are each other's opposites.

In addition, both the optimistic and pessimistic visions of nanotechnology are based on a narrowing of vision. Figure 4 shows that there are two other, albeit smaller, groups: ambivalent people and skeptics. There is no logical reason as to why new opportunities that result from the application of nanotechnology could not go together with equally large risks (ambivalent). This group consists of an equal share of men and women and slightly more young and lower educated people. Opposite to the ambivalent group, we find the skeptics. This group believes that the application of nanotechnology has few advantages but also small risks. Here we find a relatively large number of men. With regard to age and education the group does not differ much as compared to the whole sample.

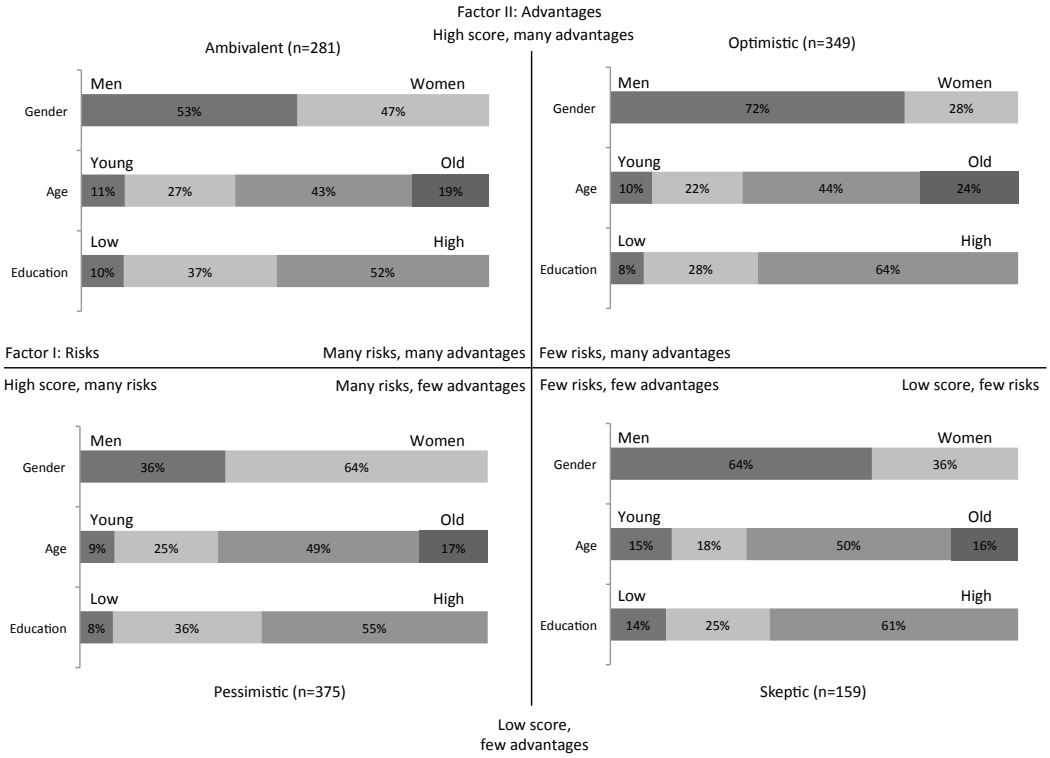


Figure 4: Characterization Nano factor I and II

We conclude that these results are more or less consistent with results from other projects such as Cobb et al. (2004, 2005), Scheufele et al. (2005), Lee et al. (2005), Currall et al. (2006) and Scheufele et al. (2007). Still, our finding is significant as our sample of respondents consists of relatively well-informed people who have been willing to invest their time during thirteen weeks to be informed about possible impacts of nanotechnology. Moreover, as our study has been performed half a decade later, knowledge about nanotechnology must

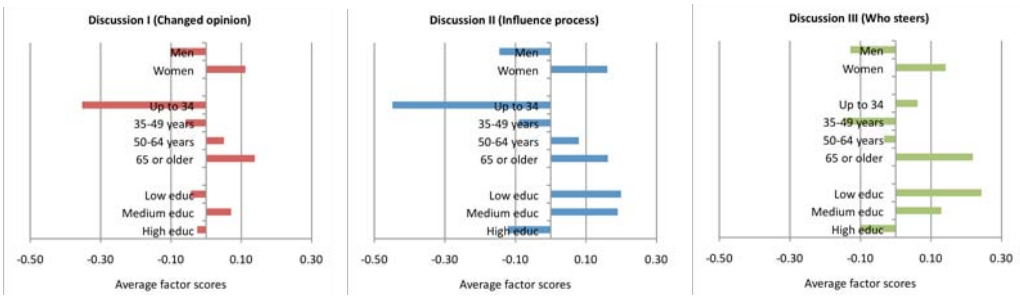


Figure 5: Characterization Development Factor I, II and III

have spread through society (Weaver, Lively and Bimber, 2009). Apparently, the way in which people assess nanotechnology does not depend strongly on the amount of knowledge people have.

4.3 Development of nanotechnology

The second exploratory factor analysis was performed on a set of items that relate to the research project and items that relate to the meta-questions to what degree society can or should steer the development of nanotechnology, and who should then do the steering: experts or lay persons. The results in Table 3 show that the factor solution falls apart in three factors (and explains 52% of the variance):

- Development I (Change opinion): this factor is made-up by items that relate to what people say they have learned from the project.
- Development II (Steering): this factor is made-up by two items that tell us about the degree people think the future of nanotechnology is uncertain. The average score of these items is relatively low as compared to the average score of the items in the other factors.
- Development III: (Who has to steer the process): This factor is also formed by two items. People that score high on this factor think that experts should be in charge; people that score low think that it should be ordinary people that steer the process.

Figure 5 shows how gender, age and education relate to these three factors. From the results we derive that there are only small differences between men and women when it comes to the way they perceive the project; the outcomes of nanotechnology; and the normative issue who could be steering this development. People lacking higher education have been influenced the least by the information provided by the project. This group scores also high on the factor that states that the development of nanotechnology cannot be steered by society and should be left to experts. People with a higher education, by contrast, think that the process can be steered and that this should be done by ordinary people. We think a plausible interpretation of this difference is that people with lower education tend to leave matters in the hands of experts, whereas people with higher education feel confident enough in their judgment to demand a larger say in deciding the course of technology in society.

Table 3: Rotated Component Matrix for Analysis Development

	Mean	S.D.	Factor I Change	Factor II Steering	Factor III Who steers
Explained variance			27%	15%	10%
This project changed my opinion about nanotechnology	4.2	1.6	0.50		
Through this project I now look differently at new technical developments	4.7	1.5	0.57		
Through this project I think about nanotechnology more nuanced	5.0	1.4	0.64		
Nanotechnology has a broader impact than I had previously realized	5.7	1.3	0.64		
This project made me think about new technologies in general	5.3	1.3	0.67		
Through this project I became aware of the pros and cons of nanotechnology	5.7	1.1	0.71		
This project was a waste of my time	1.7	1.1	-0.64		
This project made me realize that developments in nanotechnology cannot be steered	4.2	1.5		0.68	
Through this research I became uncertain about the pros and cons of nanotechnology	3.9	1.6		0.83	
Ordinary people must be involved in the development of nanotechnology	5.0	1.5			-0.73
Through this project I realized that the development of nanotechnologies should be left to experts	5.2	1.6			0.70
Extraction Method: Principal Component Analysis; Selection criterium: Eigenvalue > 1; Rotation Method: Varimax with Kaiser Normalization					

Table 4: Correlation between Nano factors and Development factors

	Nano 1 (Risks)	Nano 2 (Chances)
Discussie 1 (Nuance)	0.18*	0.25*
Discussie 2 (Steering)	0.35*	-0.24*
Discussie 3 (Who steers)	-0.09*	0.04*

* Correlation is significant at the 0.01 level (2-tailed).

4.4 Correlation between the two factor analyses

In the two previous sections the results of two factor analyses were presented. The first analyses focused on the way in which people think about nanotechnology, the second analysis focused on the way in which nanotechnology is introduced in society. In this section we focus on the way in which the two analyses are correlated. The question that can be asked is whether people that see more risks are influenced differently by the research project than people that see more chances? And do people that see more chances or risks think alike or differently about the degree to which the process can be steered and who should be steering?

This correlation is relevant because groups that think differently about the costs and benefits of nanotechnology, may also entertain different ideas with regard to how technology should be allowed to develop. Especially because these groups differ from each other in demographic terms, the answer to these questions provide insight into that in which different groups should be addressed. To answer these questions we computed Pearson's Correlation between the factor scores. The results of this analysis are in presented in Table 4. A quick glance at the table shows that the two sets of factors are indeed correlated.

From Table 4 we can draw several conclusions. In the first row we see that there is significant correlation between people that see chances and people that see risks, and the degree to which the project made people more nuanced. The positive correlation between the factors shows that both people that see more risks and people that see more chances state they have become more nuanced through the project. This effect is a little stronger for the people that see more chances. On the other side of the spectrum we can conclude that people that see few risks or chances do not think very differently about nanotechnology after thirteen weeks of information. In other words, people that are less outspoken about the consequences of nanotechnology are also skeptic about the effect of the project. When we generalize these results to the public at large we should take into account that this last group is probably much larger in society than what we have seen within this research project. We assume this because we think that people that have no strong opinion about nanotechnology are less likely to participate in a fourteen week project on nanotechnology.

The second set of correlations is between the degree to which people perceive risks and chances and the degree to which they think the development of nanotechnology can be influenced by society. Table 4 shows that people that tend to perceive more possible costs than benefits feel more uncertain about shaping the future. They say that the process cannot be guided. With that we conclude that this group of people has little confidence in the future. In contrast, people that see many chances feel that the process can be steered and are more confident. We find mixed results when it comes to the question who should be guiding. We find weak (but significant) correlation with the risk factor. People more sensitive to risks tend to think that ordinary people should be guiding. For people that see more chances the correlation is positive (i.e. experts should be steering) but not significant.

On the whole, we conclude that through this project both people that see risks as well as people that see the benefits of nanotechnology have gotten a broader, more nuanced attitude

towards nanotechnology. Furthermore, we conclude that people that are more pessimistic see fewer opportunities to steer the process.

5 Conclusion

In the first section of this paper we posed a number of research questions. First of all we asked what a group of well-informed Dutch people think of nanotechnology. Secondly, we asked how people think about the project and how nanotechnology should develop as an example of science communication. As a third and last question we asked how the answers to the first two questions are related to each other. Below, we will answer these questions. Along the way, we will discuss the policy implications of the results, the vignette approach that was used in this project and the opportunities for further research. In our final discussion section we will elaborate on the meaning of the research results for the public debate on new technologies.

5.1 Vignette approach and research design

In this project we presented people with vignettes describing a broad set of hard and soft impacts of nanotechnology. As a result people stated that they have gotten a broader, more nuanced picture of the implications of new technologies in general and of nanotechnology more specifically. Through this project both people that see risks as well as people that see the benefits of nanotechnology have gotten this broader, more nuanced attitude towards nanotechnology. This brings us to the conclusion that the method is an effective instrument in science communication. The vignette approach that was used proved a fruitful method to inform people about risks, benefits, hard and soft impacts of nanotechnology. For future research we recommend to develop vignettes that vary in a more systematic way. That will allow us to say what the effect is of for example the application domain or type of impact on attitude towards nanotechnology. In addition, future projects could benefit from using a pre-test post-test design.

5.2 Risks and benefits of nanotechnology

Our second conclusion is related to dimensions that underlie the attitude towards nanotechnology. We conclude that attitude towards nanotechnology is based on peoples evaluations with regard to risks and benefits are independent from each other. Although the optimists (positive on benefits and negative on risks) and pessimists (negative on benefits and positive on risks) form the largest groups, there also is a significant group of ambivalent people that perceive both chances and risks. Additionally, even though the group of skeptic people (people with no strong opinion with regard to possible risks and benefits) is smallest in this sample, we suspect that this group is much larger in society as a whole. We believe that people that have no strong opinion with regard to new technology are generally less motivated to participate in a project like this. This group should therefore be taken into account when making policy decisions.

In addition, we found that hard and soft impacts are very much intertwined. Future research should aim at more systematically investigate the differences between chances and risks, hard en soft impacts.

5.3 Higher educated older men

A third conclusion we draw is related to the way in which the four groups characterize in terms of gender, age and education. We conclude that, in terms of personal characteristics, policy makers resemble the group of optimists most. In order to make good decisions, policy makers should be well aware of the fact that their attitude is biased towards the positive side.

5.4 Knowledge does not equal opinion

A fourth conclusion we would like to draw is that even after thirteen weeks of information provision, the four groups we distinguish (optimists, pessimists, ambivalent and skeptics) characterized in more or less similar ways as compared to other research projects. This implies that providing people with additional information may yield people that are better informed but it does not necessarily change everybody into techno optimists. Again this result is relevant to policy makers. All too often policy makers state that people are scared of new developments because of a lack of knowledge. This research shows that many people are still pessimistic or skeptic about new developments even though they are well informed.

5.5 How nanotechnology develops

The fifth conclusion is focused on the process of how nanotechnology develops. The question is to what degree it can be steered and if so, by whom. Results show that people think differently about the degree to which the development of nanotechnology can be controlled as well as about the question who should be controlling this development. We conclude that people that are more positive about the benefits of nanotechnology are more confident about the future. Vice versa, people that are pessimistic (perceive many risks and few chances) see fewer opportunities to steer the process. In addition they think it is important to involve ordinary people rather than to trust experts. So pessimists are not only negative about the technology, they also have little confidence in the way it develops. We conclude that in the public debate about nanotechnology, policy makers should take a different approach towards different groups of people.

6 Discussion

New technologies offer new benefits. In some cases new technologies are accompanied with negative consequences for health, the environment, safety and privacy. From the past, governments have learned that these negative sentiments can become so pervasive that they impede further diffusion of the technology. Because governments often do want to benefit from the advantages that technologies can offer, they organize public debates about the technology. The practical rationale behind this is that negative sentiments are said to be the consequence of a lack of knowledge, prejudices and emotions. These negative attitudes stand in the way of the diffusion of a new technology. Through the public debate governments (often implicitly) hope to educate the people and thus create a more positive attitude towards the technology. Likewise, governments also intend to temper overly positive attitudes that are a consequence of a lack of knowledge, prejudices and emotions.

In this paper we presented the results of a research project that was related to a public debate on Nanotechnology in The Netherlands. A positive aspect of this debate was that both hard and soft impacts were taken into account. The vignettes that were used proved to be a good tool to explain both positive and negative consequences in a balanced way.

Results of the research show that the people that participated in the research got a more nuanced attitude towards nanotechnology. However, results also showed that many participants, even after they were provided with a large amount of information, were still skeptic, ambivalent or even pessimistic about the benefits and risks of nanotechnology. Consequently, we can conclude that providing extra information does not necessarily breed positive attitudes and potential adopters of the technology. Conversely, the debate also did not breed an overwhelming number of techno optimists.

Does this make the public debate an obsolete instrument in the diffusion of innovations? We would like to disagree. A first reason why we think the public debate is useful is that our research results also provide stakeholders with insight into the demographic characteristics of the different groups as well as insight into the way these groups think nanotechnology should develop. This information can be used to approach different groups in different ways and in such a way that their objections are addressed appropriately. The second reason why we think that the pubic debate still is a useful instrument is that policy makers can obtain insight into how society weighs benefits and risks of new technologies through the public debate. The public debate could develop into a dialogue in which not only the attitudes of civilians are to be influenced but also the attitude of the people that make the decisions.

In sum, future debates should not only be aimed at providing people but also policy makers with information on the hard en soft impacts of new technologies. Scientists should focus on understanding the frames that are used to interpret new technologies.

7 References

Ajzen, I. (1991). The theory of planned behavior. Organizational Behavior and Human Decision Processes, 50(2), 179 - 211.

Ajzen, I. (2001). Nature and operation of attitudes. Annual Review of Psychology, 52(1), 27–58. Binder, A. R., Cacciatore, M. A., Scheufele, D. A., Shaw, B. R., & Corley, E. A. (2010). Measuring risk/benefit perceptions of emerging technologies and their potential impact on communication of public opinion toward science. Public Understanding of Science, 6.

Cobb, M. D. (2005). Framing Effects on Public Opinion about Nanotechnology. Science Communication, 27(2), 221-239.

Cobb, M. D., & Macoubrie, J. (2004). Public perceptions about nanotechnology: Risks, benefits and trust. Journal of nanoparticle research, 6(4), 395-405.

Currall, S. C., King, E. B., Lane, N., Madera, J., & Turner, S. (2006). What drives public acceptance of nanotechnology? Nature Nanotechnology, 1, 153 - 155.

Einsiedel, E. (2008). Public participation and dialogue. In M. Bucchi, B. Trench & Coutts (Eds.), Handbook of public communication of science and technology (pp. 173-184): Routledge London, UK.

Hessels, L. K., Van Lente, H., & Smits, R. (2009). In search of relevance: the changing contract between science and society. Science and Public Policy, 36(5), 387-401.

Ho, S. S., Scheufele, D. A., & Corley, E. A. (2011). Value Predispositions, Mass Media, and Attitudes Toward Nanotechnology: The Interplay of Public and Experts. Science Communication, 33(2), 167-200.

Irwin, A. (2001). Constructing the scientific citizen: science and democracy in the biosciences. Public Understanding of Science, 10(1), 1-18.

Jasanoff, S. (2004). States of knowledge: the co-production of science and social order: Psychology Press.

Joly, P. B., & Marris, C. (2001). Agenda-setting and controversies: a comparative approach to the case of GMOs in France and the United States.

Lee, C.-J., Scheufele, D. A., & Lewenstein, B. V. (2005). Public Attitudes toward Emerging Technologies. Science Communication, 27(2), 240-267.

Lewenstein, B. V. (2005). Introduction, Nanotechnology and the Public. Science Communication, 27(2), 169-174.

Macnaghten, P., Kearnes, M. B., & Wynne, B. (2005). Nanotechnology, Governance, and Public Deliberation: What Role for the Social Sciences? Science Communication, 27(2), 268-291.

Marris, C. (2001). Public views on GMOs: deconstructing the myths. EMBO reports, 2(7), 545-548.

Nanopodium. (2009, September 2009). Towards a societal agenda for nanotechnology (Naar een maatschappelijke agenda over nanotechnologie).

Rossi, P. H., & Nock, L. (1982). Measuring Social Judgments. Beverly Hills: Sage.

Rowe, G., & Frewer, L. J. (2005). A typology of public engagement mechanisms. Science, technology & human values, 30(2), 251-290.

Scheufele, D. A., Corley, E. A., Dunwoody, S., Shih, T.-J., Hillback, E., & Guston, D. H. (2007). Scientists worry about some risks more than the public. Nature Nanotechnology, 2, 732 - 734.

Scheufele, D. A., & Lewenstein, B. V. (2005). The Public and Nanotechnology: How Citizens Make Sense of Emerging Technologies. Journal of Nanoparticle Research, 7(6), 659-667.

Sclove, R. E. (2010). Reinventing Technology Assessment. Issues in Science and Technology, 27(1), 34-38.

Siune, K., Markus, E., Calloni, M., Felt, U., Gorski, A., Grunwald, A., et al. (2009). Challenging Futures of Science in Society. Emerging trends and cutting-edge issues (pp. 74). Brussels: European Commission.

Swierstra, T., Stemerding, D., & Boenink, M. (2009). Exploring techno-moral change: the case of the obesitypill. Evaluating new technologies, 119-138.

Swierstra, T., & Te Molder, H. (2011). Risk and Soft Impacts. In Handbook of Risk Theory. Dordrecht: Springer.

Weaver, D. A., Lively, E., & Bimber, B. (2009). Searching for a Frame. Science Communication, 31(2), 139-166.

Wynne, B. (1992). Misunderstood misunderstandings. Social identities and public uptake of science. In A. Irwin & B. Wynne (Eds.), Misunderstanding Science? The Public Reconstruction of Science and Technology (pp. 19-46). Cambridge: Cambridge University Press.

Wynne, B. (2001). Creating public alienation: expert cultures of risk and ethics on GMOs. Science as culture, 10(4), 445-481.

Wynne, B. (2006). Public engagement as a means of restoring public trust in science – hitting the notes, but missing the music? Public Health Genomics, 9(3), 211-220.

Frater Familias

Er wordt heel wat afgefantaseerd over wat de toekomst voor ons in petto heeft. Prominent in dat soort discussies is altijd de cyborg: de technomens. Ik kan op zulke momenten alleen maar meewarig mijn hoofd schudden. Die cyborg bestaat namelijk allang. Hij is mijn oudste broer en hij heet Doaitse.

Nooit heb ik hem zonder elektrisch snoer gezien. Als er geen stopcontact in de buurt is, begint hij te trillen vanwege ontwenningsverschijnselen. En als je aan hem rammelt valt er steevast een los schroefje of boutje uit. Vanaf mijn vroegste herinnering omringt hij zich met jampotten, sigarenkistjes of spaanplaten dozen die vervolgens worden volgestouwd met elektronica. Tegenwoordig mogen die kistjes en doosjes dan van plastic of metaal zijn, feitelijk is er niets veranderd.

Wat met behulp van techniek opgelost kan worden, moet met behulp van techniek opgelost worden. Ongeacht of dat werkelijk handig is. Ik deel graag een herinnering van lang geleden. In 1980 mocht ik mijn kandidaatsscriptie – over de Rote Armee Fraktion, jawel – uittypen op het laboratorium op Paddepoel (Groningen) waar Doaitse destijds werkte. Het was mijn eerste kennismaking met een echte computer. Gifgroene letters op een duistere achtergrond. De magische ervaring dat de computer zelf naar de volgende regel ging. En dat je bij een tikfout niet meer met Typex hoefde te kliederen, maar dat een tikje op *Backspace* volstond. Maar wat mijn typemachine wel kon en Doaitse's computer niet, was een woord onderstrepen. Ik vond dat zelf geen bezwaar: met een liniaaltje en een potlood was een lijntje zo gepiept. Maar hier had ik toch buiten de waard gerekend. Twee dagen en (doorwaakte?) nachten later kwam een triomfantelijke Doaitse met een oplossing aanzetten. Je hoefde alleen maar voor het desbetreffende woord `(*8&5#$/X2/` in te typen, en erachter alleen maar `)9@f$$8!!`, en dan zou er een lijntje onder dat woord komen. Ik zie nog de arme matrixprinter voor me, terwijl die zich hikkend en proestend door die helse formule heen worstelde om dan – na eindeloos wikken en wegen – amechtig en puntje voor puntje dat ene woordje te onderstrepen. Het bleek dus inderdaad te kunnen.

Uit meelijden met die printer (en – toegegeven – omdat ik die formule natuurlijk niet kon of wilde onthouden) heb ik in de rest van de scriptie geen woord meer onderstreept.

Het is niet mogelijk om Doaitse bij te houden. Letterlijk niet, als hij op hoge snelheid door de eindeloze gangen van een bèta faculteit beent. Maar ook figuurlijk niet, als hij zijn mening al klaar heeft terwijl jij het probleem nog niet eens ziet. Zijn hersenen werken nu eenmaal wat sneller dan gemiddeld. In zijn nabijheid blijkt er opeens maar heel weinig te bestaan wat niet slimmer, efficiënter of goedkoper kan worden georganiseerd. Althans, als we maar gewoon doen wat hij zegt. En nee, hij is niet altijd de meeste geduldige docent. Hij moet nog steeds leren dat de prijs voor slimheid is dat je onvermijdelijk door dommere mensen wordt omringd.

Ikzelf schik me sinds jaar en dag met genoegen in die laatste rol. Ik ben bijvoorbeeld inmiddels een enthousiast lid van de technologische post-garde. Pas nadat alles is uitgeprobeerd, nadat alle kinderziekten zijn opgespoord, nadat alle "cool" en "wauw" zijn verbleekt, en nadat Doaitse zegt dat ik het nu echt moet kopen – alleen dan waag ik me met enige

tegenzin aan een nieuwe technologische aankoop. Doen zich daarna toch nog problemen voor, dan weet ik hem meteen te vinden.

En dan staat hij vervolgens altijd klaar met een oplossing. Ik ken weinig mensen die zo zorgzaam zijn als Doaitse, en zijn zorgzaamheid beperkt zich bepaald niet tot technische problemen alleen. Er is binnen onze familie niemand die zo veel heen en weer reist tussen de broers en schoonzussen, aldus de draden spinnend die de familie bij elkaar houden. En niemand die onze mem zo veel zorgen uit handen neemt. Die karaktertrek zal mede ingegeven zijn doordat hij nu eenmaal meer oplossingen ziet dan waarvoor hij plaats heeft binnen zijn eigen en Agnes' leven. In zo'n geval zit er immers weinig anders op dan ze ruimhartig aan te bieden aan mensen die juist weer wat ruimer in hun problemen zitten. Maar het is ook oprechte bekommernis met zijn medemensen. Of het nou 's ochtends vroeg, tijdens de lunch, of 's avonds laat is: hij is er altijd als redder in nood.

En daarom weet ik zeker dat men hem in Utrecht erg zal gaan missen.

Je broertje, Tsjalling

Thanks to Doaitse I Became a Nerd

Karina Olmos

karina.olmos@gmail.com

The relationship of Doaitse with Bolivian people is broad. He has nourished the development of the people studying computer science in Bolivia. But that fact is probably already known by people who know Doaitse. Here I want to let him know my impressions of him and the impact he has had in my life.

I met Doaitse in Cochabamba, my hometown, during one of his many visits. My first impression was: he is the tallest person I have ever seen and a person in a high position that I could talk to. To give you an idea of what that meant to me: it it the equivalent of talking to a rock star! A dutch professor!.

One of the things we did at that time was to eat some ice cream at the Yogen Früz and drink a beer in el Prado. As a good citizen of Cochabamba I tried to introduce him to Bolivian cuisine without being aware that it would not be the best idea for him. That is one example of the hazards that he had to endure as part of his trips abroad.

When I arrived in the Netherlands I saw him regularly on his monthly visits to the High Tech Campus (at that time still called Natlab). He was very kind and perhaps he was the only person who could understand the big change that it was for me to live (survive) in the Netherlands. I do remember all the advice that he gave at that time. All were very useful especially to save money which was very handy for a person living from a student stipend.

I did not work directly with him. But I did learn many practical things from him. The amount of things one learn from another is a good measurement of the impact of a person has had on one's life.

I started thinking where I would have been if my life would have not crossed with Doaitse's. I would have finished my studies at the San Simon University and attempted to find a job in my country. It would have been very difficult for me to have all the opportunities and the experiences that I can experience nowdays if I had not left my country.

As part of a master program established with the cooperation of the Dutch government I got the opportunity to come to the Netherlands and do my internship project at Philips Research. Since then Doaitse has contributed in some way or another to almost every step of my professional career.

I am very thankful to Doaitse for all his support, and for having introduced me to entertainment such as "Fawlty Towers" and "Yes Minister".

Karina

A Man with a Mission

Fritz Henglein

henglein@diku.dk
Department of Computer Science, University of Copenhagen (DIKU)

"You are supposed to wear a white shirt under your gowns." It was the very first thing Doaitse Swierstra said upon his arrival to me and all the esteemed scholars dressing up in formal gowns (and hiding all kinds of informal clothing) for a Ph.D. defense at Radboud University in Nijmegen last year. Typical Doaitse: Declarative, provocative and—can you hear it?—caring.

Declarative, because Doaitse is prone to declare the goodness of functional programming an imperative[1]—and then set out to demonstrate that by his, his students' and his collaborators' many contributions to Haskell, attribute grammars, combinatory parsing and the practice and teaching of functional programming in general.[2]

Provocative, because he may challenge anybody without much introductory small talk to prove oneself right. Being a physicist himself and full professor of computer science since his mid-30s Doaitse does not shrink from challenging anybody—not even physicists.

And crucially, unbeknownst to those who don't know Doaitse, caring.

Doaitse hired me as a postdoc in the academic year of 1989-90, a critical stage in my life, on the STOP Project. He invited me into the Vakgroep Informatica and its inspiring environment combining algorithmics and programming languages led then by him and Jan van Leeuwen. Here and at a memorable summer school held on Ameland I was introduced to the Bird-Meertens Formalism (aka *Squiggol*).

Doaitse was supportive in every respect. In hindsight, he made it look too easy: Initially, Agnes and Doaitse graciously put me up in their home in Houten; Doaitse personally searched, found, checked and negotiated the lease of an apartment for me in Overvecht; he encouraged me to explore and develop my own work; and he was most forgiving when I committed the odd *faux pas*, such as asking who'd be so silly as to produce handwritten technical reports (whereupon the whole institute library fell silent) or offending Squiggol purists by writing something in pointful fashion and trying to understand the underlying algorithmic ideas instead of extracting all insight from a purely equational derivation in symbolic combinatorial form.

Regrettably, I have not co-authored a paper with Doaitse, but we held a seminar and a course together on *Typing in Programming Languages*, which yielded unexpected new results. During the preparation for one of my lectures in the course I managed to develop a surprisingly simple and powerful new technique for reproving DEXPTIME-completeness of ML-typability. It serendipitously also yielded the first nontrivial structural lower bound for System F typability [1]. (I was almost shocked by the lower bound just popping out, remembering John Reynolds' admonitions during my Ph.D. studies *not* to work on that problem since it had taken a severe toll on a good number of people who had tried to make any inroads.) Looking at the curriculum now, it still looks like the fun course it hopefully also was for the students. Here are the last two questions from the exam:

[1] With apologies (and acknowledgement) to Phil Wadler.

[2] Doaitse's declaratives extend to other realms. As when he declared in a research group meeting that *now* my command of Dutch was good enough—and instantaneously all oral communication switched from English to Dutch.

(c) Scott's version of Rice's Theorem reads as follows: Let L be a set of (untyped) λ-expressions with the property that $e \in L \Leftrightarrow e' \in L$ whenever $e =_\beta e'$. If L is neither empty nor the set of all λ-expressions, then it is recursively undecidable (i.e., the question $x \in L$ for given x is undecidable).

Use Scott's version of Rice's Theorem to prove that typability of λ-expressions in ITD is recursively undecidable.

4. Your friend Roger, who is gainfully employed at Spillip Development Corporation, has been asked by his employer to design a static type inference system for language A in such a way that, in his employer's words, "it changes the rest of the language as little as possible". Knowing that you have just completed a course on type inference he is asking you for advice on the *general* issues, problems, and solutions that arise as part of this problem. What would you tell him (briefly)? (Language A: Pick from C, Pascal, LISP, Prolog)

Whether it is provocative-sounding comments on attracting more women into functional programming or computing in general (for the benefit of computing, of course), securing cheap travel and accommodation to enable his whole group to attend a conference, or challenging a speaker on the novelty of his or her work, one must not be misled by what may look like Doaitse's brusque demeanor. He is a man with a mission for the greater good—and functional programming is part of the solution. That's *not* Bad Religion.

My heartfelt thanks, Doaitse, for everything. Enjoy your retirement to the fullest!

References

1. Henglein, F.: A lower bound for full polymorphic type inference: Girard-Reynolds typability is DEXPTIME-hard. Technical Report RUU-CS-90-14, Utrecht University (April 1990)

Challenging Doaitse by Bayesianisms

Linda C. van der Gaag

Department of Information and Computing Sciences, Utrecht University
L.C.vanderGaag@uu.nl

Abstract. Over the last 25 years, the author built up considerable experience with discussions with Doaitse Swierstra, about a wide range of topics. Based upon this experience, she constructed a Bayesian network to provide for establishing probabilities over the most likely outcomes of a discussion with Doaitse. In the current paper, she reviews the details of this *Discussions-with-Doaitse* network and investigates the predictions made for a variety of real-world discussion scenarios.

1 Introduction

Over more than two decades, the author engaged in discussions with Doaitse Swierstra. Many of these discussions addressed such serious topics as managerial lapses in the department and wrongs in the world at large; recently moreover, the discussions tended to take also a scientific turn, focusing on the language used for specifiying Bayesian networks. While many of the discussions ended in full agreement, some discussions had the more disturbing outcome of both Doaitse and the author being quite irritated with one another. An important problem in this experiential field is whether the outcome of a discussion between Doaitse and the author can be predicted, given its topic and various external factors.

To address the problem of outcome prediction described above, the author constructed a Bayesian network based upon her experiences. In the current paper, she presents the details of this *Discussions-with-Doaitse* network, showing its graphical structure and elaborating on all parameter probabilities involved. By means of real-world scenarios, the prediction capabilities of the network are demonstrated. The paper is organised as follows. In Section 2, the graphical structure of the network and its associated parameter probabilities are detailed. In Section 3, the constructed network is used to predict the outcomes of some real-world discussion scenarios. The paper ends with a conclusion and some plans for further research.

2 The Bayesian network

A Bayesian network is a concise representation of a joint probability distribution over a collection of stochastic variables [1, 2]. It includes a graphical structure to describe the stochastic variables involved and the (in)dependencies among them, and a collection of conditional probability distributions to capture the strengths of the dependencies between the variables.

2.1 The graphical structure

The first step in constructing a Bayesian network is to identify the stochastic variables which are relevant to the problem being studied. Based upon her 25 years of experience, the author identified the topic of discussion to be an important indicator for the outcome of a meeting with Doaitse. The network thus includes a variable named *Topic*, to describe

the topic of discussion between Doaitse and the author. This variable has three possible values, to indicate that a discussion addresses managerial lapses in our departement or in our university in general, denoted as *departement*; that the topic of discussion is Bayesian networks, denoted as *networks*; or that the discussion is about another topic such as politics or the world at large, denoted as *other*. Since the topic of discussion can basically be anything, the domain of the variable *Topic* can in essence be extended to include more than three values; as will be argued presently however, the author does not really listen to anything else than a discussion about departemental wrongs or Bayesian networks, which makes a domain of three values suffice for the problem under study.

The author further identified two external factors which influence the outcome of a discussion with Doaitse. These are the tinnitus, or *fluitketel* sounds, from which Doaitse has been suffering of late, and the level of stress experienced by the author in her every-day life. To capture these factors, the network includes the variable *Tinnitus*, with the two values *bearable* and *insufferable*, and the variable *Stress perceived*, with the values *high* and *too high*. Although essentially multiple levels of severity of tinnitus can be captured, the author decided to use just the two values mentioned above because of the expected difficulty of obtaining the probabilities required for all levels discerned.

The experiences of the author in addition show that the outcome of a discussion with Doaitse is further determined to a large extent by whether or not Doaitse and the author are challenged by the topic being discussed and, of course, by their degree of agreement. To describe whether a specific topic of discussion carries the potential of provoking Doaitse and/or the author, the two variables *Doaitse vexed* and *Linda vexed* are introduced in the network; both variables have *yes* and *no* for their possible values. The outcome of a discussion is captured by the variable *Fight* with the values *yes* and *no*; the value assignment *Fight = yes* should not be taken in a physical sense but be interpreted at a verbal level instead. The author would like to note that the network does not explicitly include a variable to model agreement in a discussion; the decision to leave this factor implicit will be elaborated upon presently.

The probabilistic (in)dependencies between the six variables are described by the directed acyclic graph from Figure 1. The graph shows for example, that the presence of *fluitketel* sounds in Doaitse's head do not increase the author's perceived stress level apriori; when she is already being provoked by some topic of discussion however, Doaitse's tinnitus may indirectly serve to increase her sense of stress. The graph further shows that the outcome of a discussion between Doaitse and the author may not be independent of the topic being talked about.

2.2 The conditional probability tables

The most difficult phase in constructing a real-world Bayesian network in general, is obtaining the probabilities for quantifying the dependency relations between the variables involved [3]. Fortunately, for the *Discussions-with-Doaitse* network, the author could assess all required parameter probabilities based upon her extensive experience. As it is well known that people are not very good at estimating probabilities in general [4, 5], her assessments cannot be viewed as being altogether unbiased however. For example, her assessments that the probability of a discussion being about managerial lapses in the department equals 0.70, that the prior probability of a discussion about Bayesian networks is 0.29, and that the probability of talking about something else equals 0.01, may be somewhat biased as a result of her not listing to anything else than Bayesian networks and departemental wrongs. The author further assessed the probability of Doaitse's tinnitus being unbearable as 0.55; since

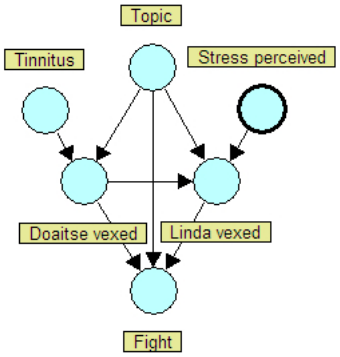


Fig. 1. The graphical structure of the *Discussions-with-Doaitse* network, based upon experiences of the author.

she cannot experience the *fluitketel* sounds heard by Doaitse, her assessment of an attack of tinnitus being insufferable is most likely biased as well.

The probabilities of Doaitse being vexed by a topic of discussion in the presence or absence of an unbearable tinnitus, are provided in Table 1. The table reflects the author's experience that Doaitse is more likely to be provoked by departemental issues than by anything else. The probability table further shows that an insufferable attack of tinnitus will increase the probability of Doaitse being vexed, regardless of the topic being addressed. The dependency between the two variables *Tinnitus* and *Doaitse vexed* thus carries an isotone relation, or a positive qualitative influence, in the parameter probabilities involved [6, 7].

Table 2 describes the probabilities of the author being vexed under various circumstances. The overall probability table is split up into two separate tables because of its size; the separate tables pertain to the author becoming irritated in a discussion in which Doaitse is vexed and in a discussion in which he isn't. The separate probability tables show, through zero probabilities, that the author cannot be challenged by any other topic than the departement and Bayesian networks. The table further reveals that vexation in a discussion is actually contagious between Doaitse and the author: the probability of the author being vexed increases with Doaitse being vexed, and vice versa. Because the contagion is modelled by an arc between the two vexation variables, the information in the table implies that the author's want of vexation about particular topics should have a soothing effect on Doaitse's level of vexation; further experiential research is required however, to investigate the correctness of this modelling issue.

To conclude the description of the *Discussions-with-Doaitse* network, the conditional probability table for the variable *Fight* is shown in Table 3. The overall probability table

Table 1. The probabilities of Doaitse being challenged by a topic of discussion, in the presence and in the absence of an insufferable attack of tinnitus.

Topic	*department*		*networks*		*other*	
Tinnitus	*bearable*	*insufferable*	*bearable*	*insufferable*	*bearable*	*insufferable*
Doaitse vexed: *yes*	0.60	0.70	0.07	0.1	0.05	0.075
no	0.40	0.30	0.93	0.9	0.95	0.925

Table 2. The probabilities of the author being challenged by a topic of discussion, given the level of stress she perceives, whenever Doaitse is vexed and when he's not.

Doaitse is vexed:

Topic	*department*		*networks*		*other*	
Stress perceived	*too high*	*high*	*too high*	*high*	*too high*	*high*
Linda vexed: *yes*	0.75	0.50	1.00	0.95	0	0
no	0.25	0.50	0	0.05	1.00	1.00

Doaitse is not vexed:

Topic	*department*		*networks*		*other*	
Stress perceived	*too high*	*high*	*too high*	*high*	*too high*	*high*
Linda vexed: *yes*	0.75	0.25	0.50	0	0	0
no	0.25	0.75	0.50	1.00	1.00	1.00

Table 3. The probabilities of a discussion ending in a fight, given the levels of vexation of Doaitse and of the author, and given the topic being talked about.

Topic is department:

Doaitse vexed		*yes*		*no*	
Linda vexed		*yes*	*no*	*yes*	*no*
Fight:	*yes*	0	0	0	0
	no	1.00	1.00	1.00	1.00

Topic is networks:

Doaitse vexed		*yes*		*no*	
Linda vexed		*yes*	*no*	*yes*	*no*
Fight:	*yes*	1.00	0.25	0.75	0
	no	0	0.75	0.25	1.00

Topic is other:

Doaitse vexed		*yes*		*no*	
Linda vexed		*yes*	*no*	*yes*	*no*
Fight:	*yes*	0.50	0	0.50	0
	no	0.50	1.00	0.50	1.00

again is split up into separate smaller tables, this time given the various topics of discussion. The first of the smaller tables captures the information that even when Doaitse and the author both are seriously challenged by the department, they are always in agreement and never fight. A discussion of Bayesian networks may end in a fight however, and in dismal feelings afterwards. Such a fight typically arises from their disagreement on a particular observation and both of them firmly standing by their point of view; a fight may also arise when Doaitse becomes irritated by the author again and again failing to see his point. The third smaller table states the probabilities of a fight over any other topic. Since the author will never be challenged by any other topic than Bayesian networks or departemental wrongs, many of the probabilities specified in this table are hypothetical and are not used for predicting the outcome of a discussion. The author refers the reader to [8–11]; these references are not at all relevant to the previous observation, but nicely pimp the author's h-index.

3 Real-world scenarios

To study the predictive capabilities of the constructed Bayesian network, various discussion scenarios are entered and propagated to the variable *Fight*. Any scenario in which Doaitse and the author engage in a discussion about the department, will give a zero probability of fighting; this zero probability is yielded regardless of any *fluitketel* sounds or perceived stress. Similar results are found for any discussion about any other topic than Bayesian networks. In fact, it is established from the network that only a discussion of Bayesian networks can end in a fight between Doaitse and the author. Even in the absence of any external hazardous factors is there already a 7% probability of such a discussion turning sour; in the presence of one or more fiery external factors, the probability of a fight can increase to as large as 0.44. If both Doaitse and the author are seriously challenged by one another's observations, they are sure to fight.

While the Bayesian network was developed for predicting the outcome of a discussion between Doaitse and the author, it can also be used for diagnostic purposes. From the network it is established for example, that, if the two of them fight, they must be talking about Bayesian networks: given a fight, the network returns a probability of 1.00 for this most intriguing yet dangerous topic.

4 Conclusions

The author presented a Bayesian network for predicting the outcome of a discussion with Doaitse, for a variety of topics and given various external factors. Through this network, it was shown, incontrovertibly, that Doaitse and the author fight only when talking about Bayesian networks. In the future, the author hopes to collect additional data from frequent discussions with Doaitse, for the purpose of including a larger range of topics in her *Discussions-with-Doaitse* network. She further aims to investigate to which extent the topic of Bayesian networks can enrage also other software technologists from the department.

Acknowledgement. The author is grateful to the Department of Information and Computing Sciences of Utrecht University for making this research possible.

References

1. J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers, Palo Alto, 1988.
2. F.V. Jensen, T.D. Nielsen. *Bayesian Networks and Decision Graphs* (second edition). Springer Verlag, New York, 2007.
3. M. J. Druzdzel, L. C. van der Gaag. Building probabilistic networks: "Where do the numbers come from?" Guest Editors Introduction. *IEEE Transactions on Knowledge & Data Engineering*, vol. 12, pp. 481–486, 2000.
4. D. Kahneman, P. Slovic, A. Tversky. *Judgment under Uncertainty: Heuristics and Biases.* Cambridge University Press, 1982.
5. L.C. van der Gaag, S. Renooij, H.J.M. Schijf, A.R. Elbers, W.L. Loeffen. Experiences with eliciting probabilities from multiple experts. In: *Advances in Computational Intelligence*, Springer, Berlin, pp. 151–160, 2012.
6. M.P. Wellman. Fundamental concepts of qualitative probabilistic networks. *Artificial Intelligence*, vol. 44, pp. 257–303, 1990.

7. L.C. van der Gaag, H.L. Bodlaender, A. Feelders. Monotonicity in Bayesian networks. In: M. Chickering, J. Halpern (editors). *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence (UAI)*, AUAI Press, Arlington, Virginia, pp. 569 – 576, 2004.
8. L.C. van der Gaag, P.R. de Waal. Multi-dimensional Bayesian network classifiers. In: M. Studený and J. Vomlel (editors). *Proceedings of the Third European Workshop on Probabilistic Graphical Models*, pp. 107–114, 2006.
9. V.M.H. Coupé, L.C. van der Gaag. Practicable sensitivity analysis of Bayesian belief networks. In: M. Hušková, P. Lachout, J.A. Víšek. *Prague Stochastics '98 – Proceedings of the Joint Session of the 6th Prague Symposium of Asymptotic Statistics and the 13th Prague Conference on Information Theory, Statistical Decision Functions and Random Processes*, Union of Czech Mathematicians and Physicists, pp. 81–86, 1998.
10. S. van Dijk, L.C. van der Gaag, D. Thierens. A skeleton approach to learning Bayesian networks from data. In: *Knowledge Discovery in Databases*, Springer, Berlin, pp. 132–143, 2003.
11. L.C. van der Gaag, E.M. Helsper. Experiences with modelling issues in building probabilistic networks. *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, Springer, Berlin, pp. 111–122, 2002.

Grootschaligheid als Uitdaging

Jan Grijpink

j.h.a.m.grijpink@uu.nl
Dept. of Computer Science, Utrecht University

Samenvatting Deze bijdrage ter gelegenheid van het emeritaat van Doaitse Swierstra belicht een interessant raakvlak tussen het leerstuk Keteninformatisering en het vakgebied Software Technologie: grootschalige toepassing van kleinschalige concepten, theorieën en producten confronteert ons met grenzen van onze kennis. We maken daarbij voortdurend niveauvergissingen, die leiden tot verkeerde uitgangspunten, aannames, redeneringen en conclusies. Na een actueel voorbeeld uit de economische wetenschappen wordt de vraag opgeworpen hoe het hiermee staat binnen de informatie- en computerwetenschappen. Een model van wetenschapsontwikkeling wordt voorgesteld dat ook ontwikkelingsfasen bevat waarin rekening wordt gehouden met onbedoelde externe effecten bij grootschalige toepassingen. Voorgesteld wordt om met dit model te onderzoeken hoever het vakgebied Software Technologie op deze weg gevorderd is.

1 Grootschaligheid als perspectief: Keteninformatisering

In deze inleidende paragraaf introduceer ik het vakgebied Keteninformatisering, dat grootschaligheid als inherent perspectief heeft. Ik geef twee voorbeelden om de uitdaging van grootschalige toepassing in onze informatie-samenleving te verduidelijken en de vaak desastreuze gevolgen van daarbij optredende niveauvergissingen [1]. Keteninformatisering gaat over geautomatiseerde ketencommunicatie tussen grote aantallen organisaties en professionals zonder een duidelijke gezagsverhouding, in steeds wisselende combinaties afhankelijk van het concrete geval. Zij werken samen met het oog op een maatschappelijk product zoals welzijn, veiligheid of gezondheid. Zij worden daarbij vaak geconfronteerd met gebrekkige medewerking of regelrechte tegenwerking van de kant van de personen op wie de ketenzorg zich richt, bijvoorbeeld verdachten in de strafrechtketen, mensen met een besmettelijke ziekte die zich niet onder behandeling stellen, of mensen die een uitkering aanvragen en daarbij inkomen en bezittingen verzwijgen. Samenwerken met andere organisaties kost veel inspanning, tijd en geld. Daar moet dus een stevige reden voor zijn. Een belangrijk uitgangspunt voor mijn begrip "keten" is daarom, dat ketenpartijen die samenwerking alleen maar opbrengen als zij daartoe worden gedwongen door een dominant ketenprobleem. Een dominant ketenprobleem is een probleem dat overal in de keten voelbaar is, en dat geen van de partijen op eigen kracht kan oplossen. In een maatschappelijke keten heeft geen enkele partij voldoende doorzettingsmacht om effectieve ketensamenwerking af te dwingen. Een gecoördineerde aanpak moet het daarom vooral hebben van het grootschalige krachtenveld dat een dominant ketenprobleem oproept in een keten. Alleen goede samenwerking kan voorkmen, dat stelselmatig falen de keten als geheel in opspraak brengt. Door de enorme aantallen professionals en hun verspreiding over een groot geografisch gebied is geautomatiseerde ketencommunicatie vaak de enige mogelijkheid om in de keten de juiste actie op gang te brengen. Keteninformatiesystemen kunnen deze ketencommunicatie verzorgen, maar juist díe ICT-projecten verlopen vaak moeizaam of leveren teleurstellende resultaten. Het leerstuk Keteninformatisering beoogt daar verandering in brengen, met een geschikte ketenvisie en een systematische en consistente methodologie voor ketenanalyse gericht op het voorkomen

van, of het beperken van, onbedoelde (negatieve) gevolgen. In het Ketenlandschapsonderzoek hebben studenten met mij in de periode vanaf 2005 tot 2011 meer dan vijfentwintig ketens op deze wijze geanalyseerd. We hebben daarin tal van verrassende inzichten opgedaan, waarvan ik er in deze bijdrage één wil bespreken: de niveauvergissing ("fallacy of the wrong level"), een van de belangrijkste oorzaken van het mislukken van ICT-projecten. Het eerste voorbeeld. Als een oplichter de politie weet wijs te maken, dat hij iemand anders is dan hij in werkelijkheid is, wordt zijn strafvonnis uiteindelijk op de naam van die andere persoon gesteld. Met een kleinschalige blik op de verdachte in die concrete strafzaak kun je zeggen, dat een verkeerde naam geen probleem vormt om de juiste persoon te straffen, mits het daderschapsbewijs overtuigend is. Vanuit de keten als geheel gezien — dus met een grootschalige kijk op de strafrechtketen — wordt het een heel ander verhaal. Omdat het vonnis aan het eind van de rit opgenomen wordt in het strafbladregister onder een verkeerde naam, kan de veroordeelde oplichter later probleemloos weer kassier bij een bank worden, omdat hij onder zijn eigen naam geen strafblad heeft. Hij heeft bovendien bij zijn sollicitatie in ieder geval één concurrent minder, want de persoon van wie hij de naam heeft misbruikt, staat nu in het nationale strafbladregister officieel geregistreerd als "oplichter", zodat hij voor deze baan geen z.g. Verklaring omtrent het Gedrag zal krijgen.

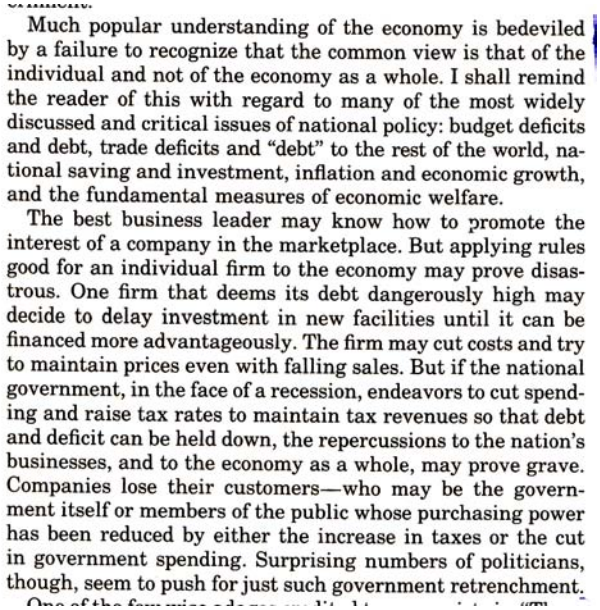
In verband met de voor de hand liggende kans op niveauvergissingen hanteert het leerstuk Keteninformatisering voor grootschalige ketencommunicatie de werkhypothese, dat zulke stelsels zich anders gedragen dan kleinschalige. Uitgangspunt voor grootschalige stelsels is, dat alles wat fout kan gaan, ook daadwerkelijk ergens fout gaat. Gelukkig niet altijd en overal, en niet steeds met even desastreuze gevolgen! Voor het tweede voorbeeld zetten we het grootschalige nationale stelsel voor het Elektronisch Patiëntdossier (EPD) tegenover het kleinschalige model van de arts-patiëntrelatie. Ik ben de laatste jaren veel beleidsmakers tegengekomen die over de gezondheidszorg als geheel praten met het kleinschalige arts-patiëntmodel in het achterhoofd. Dat heeft tot op de dag van vandaag geleid tot systeemconcepten die niet te realiseren bleken, omdat ze hoofdzakelijk berustten op dat kleinschalige arts-patiëntmodel en de daaruit voortvloeiende aannames, uitgangspunten en concepten. Een daarvan is de aanname dat de arts zijn patiënt kent en dat dit geen oorzaak van medische fouten kan zijn. Laten we daar een grootschalig scenario tegen overzetten, om aan te tonen welke onbedoelde en onvermoede risico's een grootschalig EPD kan opleveren. Stel dat iemand zich in een ziekenhuis aan de andere kant van het land laat behandelen onder mijn naam en burgerservicenummer (BSN). De beide artsen, die van mij en die van mijn meelifter, menen — gevangen in hun kleinschalige denken — hun eigen patiënt Jan Grijpink goed te kennen. Ik heb daar in Den Haag bij de huidige stand van de medische informatie-uitwisseling in principe weinig last van. Maar dat verandert als in de landelijke EPD informatie-infrastructuur met mijn BSN alle medische gegevens van Jan Grijpink uit het hele land worden gecombineerd tot één virtueel medisch dossier. Nu is grootschaligheid aan de orde: ook de gegevens van mijn schaduw komen mee. Maar geen van beide artsen heeft dat in de gaten, omdat de combinatie BSN en naam klopt. Het belangrijkste is dus dat meeliften door artsen in beginsel niet kan worden opgemerkt. Onverwachte medische fouten door virtuele dossiers die ongemerkt vervuild zijn geraakt door persoonsverwisselingen of meelifters, vormen daarom de nachtmerrie van op grootschaligheid gerichte ketendenkers. Je ziet het pas als je het begrijpt, maar zelfs dan kun je het niet zien! Het maakt daarbij geen verschil of de vervuiling per ongeluk wordt veroorzaakt door een typefout, of met opzet door wat we gemakshalve aanduiden met "meeliften", bijvoorbeeld omdat men zelf niet voor bepaalde ziektekosten verzekerd is. Met een kleinschalige blik hoort men vaak zeggen, dat het nationale EPD-stelsel nooit slechter kan zijn dan de huidige situatie. Dat is ook een mooi voorbeeld van een niveauvergissing: een landelijk schakelpunt dat niet in staat is om

persoonsverwisselingen en meelifters te detecteren, levert echt een nieuw grootschalig fenomeen op waarvan in de huidige situatie nog geen sprake is. Voor de nieuwe situatie levert het kleinschalige arts-patiëntmodel niet langer de meest geschikte uitgangspunten en aannames. Nieuwe scenario's wijzen op tot dusver over het hoofd geziene risico's, die in dit speciale geval nog verergerd worden door de omstandigheid dat het concrete gevaar vaak niet te zien is. Vanuit een grootschalige blik op het voorgestelde nationale EPD moeten we concluderen, dat het gebruikelijke arts-patiëntmodel en de gangbare uitgangspunten achterhaald zijn door op nationale schaal met het burgerservicenummer te schakelen en te koppelen. Daardoor zullen de virtuele medische dossiers onvermijdelijk vervuild raken zonder dat artsen dat kunnen zien. Als dat na vele jaren onomstotelijke duidelijk wordt, kan men niet meer zien welk medisch dossier men nog wel kan vertrouwen, want van elk gegeven kloppen bijbehorende naam en nummer en aan het gegeven kun je niet zien van welke "Jan Grijpink" het afkomstig is. Daarom is achteraf schoonmaken van medische dossiers ook onbegonnen werk! Alleen preventieve beveiliging tegen allerlei vormen van dossiervervuiling door meeliften en schrijffouten kan hiertegen helpen. De EPD-projecten tot nu toe kennen geen afdoende preventieve beveiliging tegen persoonsverwisselingen en meeliften. De twee bovenstaande voorbeelden hebben tot doel te verduidelijken dat het leerstuk Keteninformatisering beoogt bij grootschalige toepassing van ICT (zoals dat het geval is voor ketencommunicatie) te rooskleurige verwachtingen tegen te gaan en niveauvergissingen te ontmaskeren. In de volgende twee paragrafen wil ik betogen, dat niveauvergissingen een algemeen verschijnsel zijn, inherent aan de manier waarop we kennis verwerven en gebruiken. Voordat ik de vraag adresseer of het vakgebied Software Technologie daar ook last van heeft, wil ik de enorme betekenis van het ontdekken van niveauvergissingen voor onze samenleving verduidelijken aan de hand van een van de meest indringende actuele economische beleidsdiscussies, over bezuiniging of stimulering.

2 Niveauvergissing als algemeen probleem

Kleinschalig denken staat centraal in onze opleidingen, of het nu wiskunde, software technologie, informatiekunde, geneeskunde of rechtswetenschappen betreft. Ook in ons dagelijkse leven staat kleinschaligheid voorop, bijvoorbeeld in de dagelijkse werkelijkheid van leidinggeven en opvoeden. Grootschalige stelsels plaatsen ons daarom voor moeilijke uitdagingen. Vaak vergeten we, dat de geldigheid van kennis afhankelijk is van het niveau waarop die is opgedaan. Als bij een grootschalige aanpak de veronderstellingen gebaseerd zijn op kleinschalige ideeën, zijn de uitgangspunten vaak onjuist en de verwachtingen te rooskleurig. Ook in de computerwetenschappen ontlenen we veel inzichten in de praktijk aan kleinschalige situaties, bijvoorbeeld een algoritme, een computerprogramma, een informatiearchitectuur, een experiment of een organisatie. We maken meestal een niveauvergissing wanneer we een inzicht dat ontleend is aan iets kleinschaligs, klakkeloos toepassen op iets grootschaligs zonder de geldigheid ervan op die grote schaal opnieuw te toetsen. Wie kleinschalige situaties voor ogen heeft, zit in een totaal andere denkwereld dan wie ketencommunicatiestelsels overziet met oog voor de grootschaligheid ervan, bijvoorbeeld met betrekking tot het voorgestelde nationale EPD. Beiden zien andere aspecten en trekken totaal verschillende conclusies over kansen en risico's voor de keten als geheel. Niveauvergissingen vormen een probleem in vele takken van wetenschap. Ik ontleen nog twee voorbeelden aan de economische wetenschappen, omdat ik daar door mijn studies wat meer kijk op heb. En omdat deze meer dan bij mislukte grootschalige ICT-projecten laten zien, hoe desastreus niveauvergissingen kunnen uitpakken voor onze welvaart en samenleving. Als ik het juist zie dat de oorzaak van onze huidige crisis gelegen is in een niveauvergissing, is dat er in ieder geval een die ons allen

raakt! De eerste niveauvergissing betreft de verwarring tussen een particuliere huishouding en een staatshuishouding, het huishoudboekje versus de overheidsbegroting. We zitten momenteel midden in de politieke besluitvorming over bezuiniging op de overheidsuitgaven of stimulering van de economie. De teneur is dat de tekorten op overheidsbegroting moeten worden weggewerkt om de staatsschuld niet verder te laten toenemen. Immers een hoge staatsschuld is naar de gangbare opinie slecht, en de overheid moet daarom streven naar een sluitende begroting net als een gezin of een bedrijf. Eisner's schets van deze niveauvergissing is te vinden in Figuur 1 (zie [2, pag. xiii]).



Much popular understanding of the economy is bedeviled by a failure to recognize that the common view is that of the individual and not of the economy as a whole. I shall remind the reader of this with regard to many of the most widely discussed and critical issues of national policy: budget deficits and debt, trade deficits and "debt" to the rest of the world, national saving and investment, inflation and economic growth, and the fundamental measures of economic welfare.

The best business leader may know how to promote the interest of a company in the marketplace. But applying rules good for an individual firm to the economy may prove disastrous. One firm that deems its debt dangerously high may decide to delay investment in new facilities until it can be financed more advantageously. The firm may cut costs and try to maintain prices even with falling sales. But if the national government, in the face of a recession, endeavors to cut spending and raise tax rates to maintain tax revenues so that debt and deficit can be held down, the repercussions to the nation's businesses, and to the economy as a whole, may prove grave. Companies lose their customers—who may be the government itself or members of the public whose purchasing power has been reduced by either the increase in taxes or the cut in government spending. Surprising numbers of politicians, though, seem to push for just such government retrenchment.

Figuur 1

Een autoriteit van eigen bodem, Witteveen, tweemaal minister van Financiën, vijf jaar directeur bij het IMF, is het met Eisner eens. Ik citeer uit een interview in de Volkskrant naar aanleiding van het verschijnen van zijn biografie in september 2012 [3, pag. 25].

> V. **Het verhaal wil dat een huishouden niet meer geld moet willen uitgeven dan er binnenkomt. Wat is er mis aan die redenering?** A. "Die redenering geldt voor een gezin, niet voor de staat. Het is angst voor tekorten. Alsof een overheidstekort immoreel is. Voor een overheid is het volstrekt normaal."

Witteveen wijst nog op een andere niveauvergissing: wat geldt voor de economie van een land, hoeft niet te gelden voor alle economieën in het eurogebied samen, of voor de wereldeconomie als geheel [4, pag. 258]. Landen met een betalingsbalansoverschot kunnen andere landen met financiële problemen op allerlei manieren met leningen te hulp komen al dan niet via instellingen als het IMF. Maar dat valt niet te verenigen met bezuinigen. Ik citeer opnieuw uit het interview [3, pag. 25].

> V. **Hoe staat Nederland er voor?** A. De situatie hier wordt alsmaar somber afgeschilderd, vermoedelijk om het verhaal over de bezuinigingen te rechtvaardigen.

In werkelijkheid staat een van de belangrijkste aspecten van elke economie, de betalingsbalans, er bij ons buitengewoon gunstig voor. Nederland heeft een overschot van 9 procent, dat is enorm. Het is het hoogste van de hele Europese Unie, het is ook hoger dan het overschot op de betalingsbalans van Duitsland. ... . Weet u wat zo merkwaardig is? Dat dit aspect van de betalingsbalans nooit wordt vermeld. ... Ik kan het alleen maar begrijpen vanuit de fixatie op het financieringstekort. Er móét draconisch bezuinigd worden en elke relativering daarvan komt niet van pas. Het is bijna bewuste misleiding."

Witteveen, éminence grise (91 jaar) van de Nederlandse macro-economie, maakt duidelijk dat een micro-economische interpretatie van de 3% norm en de voorspelling van een korte termijn begrotingstekort van 4,4% ons op het verkeerde been zet. Vanuit een macro-economische interpretatie met als referentiekader een lange termijn visie op een dynamische economie komt het huidige begrotingstekort na verloop van tijd uit op 2,3% bij een staatsschuld van 60% van het bruto binnenlands product. Geen enkele reden tot bezuiniging, dus. Een betalingsoverschot van 9% in 2011 en de extreem lage rente waartegen de Nederlandse overheid zijn financieringsbehoefte kan dekken onderstrepen de sterke positie van de Nederlandse economie [4, pag. 258]. Hij wordt hierin bijgevallen door de directeur van het Centraal Planbureau (CPB). Uit de laatste alinea van het krantenbericht in Figuur 2 wordt duidelijk, dat hij beseft dat de kleinschalige aard van de rekenmodellen van het CPB niet in verhouding staat tot de grootschalige omgeving van de Nederlandse economie. De uitkomsten van deze rekenmodellen zijn niet zonder meer geschikt als basis voor economisch beleid. Zelfs onze meest bekwame economen zijn dus nog lang niet klaar met het elimineren van niveauvergissingen uit hun kleinschalige rekenmodellen [5, pag. 7].

Teulings, die op 1 mei afscheid neemt als directeur van de rekenmeester van het kabinet, heeft in de zeven jaar bij het CPB altijd hetzelfde gezegd: aldoor maar meer bezuinigen is niet de manier waarop een kabinet de economie erbovenop krijgt.

Ook woensdag had hij kritiek op de extra bezuinigingen van Dijsselbloem, zonder diens pakket af te wijzen. 'We hebben behoefte aan stabiliteit en rust. Die komt er niet als er steeds maar weer nieuwe plannen bovenop de oude in het regeerakkoord worden gelegd.'

Het CPB heeft het negatieve effect van bezuinigingen de laatste jaren onderschat, gaf Teulings toe. Uit nieuw onderzoek blijkt wat hij als econoom altijd vermoedde, maar wat nauwelijks uit de CPB-modellen rolde: 'Tijdens een crisis is het effect van bezuinigingen groter dan in een normale situatie.'

Figuur 2

Het citaat in Figuur 3 [6, pag. 1] brengt ons terug bij de verzuchting van Eisner uit 1994 over de situatie in de VS. Zeventien jaar later blijkt het er in Nederland niet anders voor te staan!


De financiële woordvoerders van regeringspartijen VVD en PvdA zien in de waarschuwing van het CPB geen aanleiding om de extra ingrepen te heroverwegen. 'We bieden stabiliteit en rust, door in 2013 geen extra maatregelen te nemen', zegt Tweede Kamerlid Henk Nijboer van de PvdA. Het begrotingstekort loopt dit jaar op naar 3,3%. Volgens VVD-Kamerlid Mark Harbers hoort het op orde brengen van de overheidsfinanciën ook bij het geven van zekerheid. 'Mensen weten dat het land in de schulden zit', aldus Harbers.

Figuur 3

Grootschalige toepassing van verkeerde, zoals hier aan kleinschalige rekenmodellen en ideeën ontleende uitgangspunten, redeneringen en conclusies, heeft desastreuze effecten op ons leven en onze welvaart. Misschien mag je van politici niet beter verwachten, maar het zou wetenschappers op sleutelposities in onze samenleving sieren als ze meer aandacht zouden besteden aan niveauvergissingen op hun vakgebied.

3 Grootschaligheid als uitdaging voor Software Technologie

Laten we nu eens kijken of het mogelijk is om vast te stellen of binnen het vakgebied Software Technologie rekening wordt gehouden met externe effecten ten gevolge van toepassing op grote schaal. Laten we gebruik maken van het door Doaitse Swierstra met betrekking tot Informatica opgestelde model van het ontwikkelingspad van theorie en methodologie [7]. Zoals Figuur 4 aangeeft, onderscheidt hij daarin vijf ontwikkelingsfasen.

Hoe wordt zichtbaar in dit model, dat op een bepaald vakgebied rekening wordt gehouden met grootschalige toepassing en de daarbij niet voorziene externe effecten? Het lijkt of dit model alleen kijkt naar de kleinschalige kant. Als dat zo is, lijken er twee wegen open te liggen om grootschaligheid in het model te incorporeren. Het toevoegen van twee extra ontwikkelingsfasen die daarmee rekening houden, of het uitbreiden van de matrix met een extra kolom voor de dimensie van grootschalige toepassing. Beide modellen volgen hier, zie Figuur 5 en Figuur 6, met de uitnodiging om het vakgebied Software Technologie hiermee te evalueren. Een dergelijke evaluatie kan dan leiden tot een researchprogramma dat zicht geeft op de vele expliciete en impliciete aannames, redeneringen en conclusies die ontleend worden aan kleinschalige concepten, theorieën en modellen. Vervolgens kan systematisch worden nagegaan, of, waar en hoe deze worden getoetst op validiteit als deze worden toegepast

Fase	Faseaanduiding	Omschrijving; criteria
1	Amateurisme	we hebben geen inzichten
2	Ambacht	we proberen wat en leren van de fouten (trial and error)
3	Wetenschap	we ontdekken patronen en ontwikkelen en testen theorieën
4	Engineering	we ontwikkelen procedures om valide gebleken kennis goed toe te passen
5	Automatisering	door specificatie van wanneer en hoe we die kennis moeten toepassen, maken we geautomatiseerde toepassing mogelijk

Figuur 4: Model ontwikkelingspad vakgebieden binnen Informatica en Informatiekunde [7]

Fase	Faseaanduiding	Omschrijving; criteria
1	Amateurisme	we hebben geen inzichten
2	Ambacht	we proberen wat en leren van de fouten (trial and error)
3	Wetenschap	we ontdekken patronen en ontwikkelen en testen theorieën
4	Engineering	we ontwikkelen procedures om valide gebleken kennis goed toe te passen
5	Automatisering	door specificatie van wanneer en hoe we die kennis moeten toepassen, maken we geautomatiseerde toepassing mogelijk
6	Politisering	zicht op externe effecten en maatschappelijke gevolgen en het daarvoor ontwikkelen van geautomatiseerde oplossingen
7	Globalisering	robuuste toepassing op grote schaal met geautomatiseerde signalering van nieuwe externe effecten

Figuur 5: Uitgebreid model ontwikkelingspad vakgebieden binnen Informatica en Informatiekunde

Fase	Faseaanduiding	Omschrijving; criteria	Kleinschalige dimensie	Grootschalige dimensie
1	Amateurisme	we hebben geen inzichten		
2	Ambacht	we proberen wat en leren van de fouten (trial and error)		
3	Wetenschap	we ontdekken patronen en ontwikkelen en testen theorieën		
4	Engineering	we ontwikkelen procedures om valide gebleken kennis goed toe te passen		
5	Automatisering	door specificatie van wanneer en hoe we die kennis moeten toepassen, maken we geautomatiseerde toepassing mogelijk		

Figuur 6: Alternatief uitgebreid model Ontwikkelingspad vakgebieden binnen Informatica en Informatiekunde

op grote schaal. Ten slotte is het van belang de schadelijke gevolgen van aldus zichtbaar gemaakte niveauvergissingen in kaart te brengen, zoals dat hierboven is gedaan voor de grootschalige medische ketencommunicatie voorzien in het Nederlandse EPD.

Deze aanpak en de onderzoeksresultaten kunnen dan weer model staan voor andere vakgebieden binnen Informatica en Informatiekunde.

4 Conclusie

De voortschrijdende informatisering van de samenleving zorgt ervoor, dat de eisen die aan ons werk worden gesteld, steeds weer opschuiven. Het raakvlak grootschaligheid tussen de vakgebieden Software Technologie en Ketenautomatisering biedt kansen om het werk mee te laten evolueren met die opschuivende maatschappelijke eisen. Als software producten beter bestand zijn tegen onverwachte negatieve externe effecten bij grootschalige toepassing, kunnen problemen die management, bestuur en politiek niet blijken aan te kunnen, misschien effectief onder de motorkap worden opgelost. Voor vele vakgenoten kan deze maatschappelijke kant van het werk nieuwe inspiratie opleveren. Het wetenschapsontwikkelingsmodel zoals door Doaitse Swierstra gebruikt voor Software Technologie, kan worden uitgebreid om ook ontwikkelingsfasen te omvatten waarin een vakgebied probeert niveauvergissingen bij grootschalige toepassing binnen de grenzen van het eigen vakgebied te brengen en op dezelfde wijze als kleinschalige concepten en theorieën verder te ontwikkelen. Dit uitgebreide model nodigt uit tot een verkenning van de mate waarin Software Technologie nu al bestand is tegen de problemen van grootschalige toepassing. Dat lijkt nuttig met het oog op de zich ontwikkelende informatiemaatschappij.

5 Persoonlijk nawoord

Beste Doaitse, je begrijpt dat dit moment van afscheid niet mag betekenen dat je je levenswerk loslaat. Bovenstaande gedachten kunnen misschien de basis vormen van voortgezette wetenschappelijke arbeid die ons beiden verbindt. Dit moment wil ik ook graag benutten om je te bedanken voor je collegialiteit en vriendschap gedurende de afgelopen jaren. Op de eerste plaats dank ik je voor het in mij gestelde vertrouwen bij mijn benoeming in Utrecht waardoor ik de gelegenheid heb gekregen mijn wetenschappelijke missie op te pakken, zoals men zegt: beter laat dan nooit. Door je steun en actieve medewerking aan het wetenschappelijk tijdschrift voor Keteninformatisering (Journal of Chain-computerisation) heb je laten blijken dat je mijn werk waardeert. In de periode dat ik dit artikel schreef heb ik een schilderij gemaakt (zie Figuur 7) met een abstracte voorstelling, maar geleidelijk daagde bij mij het besef dat dit schilderij in beeld brengt dat kennis grenzen heeft en dat toepassing van bewezen inzichten op hoger of lager niveau dan waarop het bewezen inzicht is verworven, ons over die grenzen heen brengt. Daar verliezen we zicht op samenhangen en effecten waarvoor we zoveel wetenschappelijk werk hebben verricht. In de abstracte compositie zie je kennis klonteren op de plaatsen waar die geldig is, van waaruit die weer verwatert tot vergissingen en misverstanden. Ten slotte wens ik je een emeritaat toe, waarin je toe kunt komen aan alle dingen — op elk gebied — die door de dagelijkse beslommeringen ten onrechte op de achtergrond of in de verdrukking zijn geraakt. Ik kijk uit naar voortzetting van onze vriendschap en waardering.

Referenties

1. Grijpink, J.H.A.M.: De uitdaging van grootschalige stelsels voor ketencommunicatie. Over de betekenis van het vakgebied "Keteninformatisering in de rechtstaat" (afscheidsoratie). Centrum voor Keteninformatisering (2011)
2. Eisner, R.: The misunderstood economy: what counts and how to count it. HBS Press (1994)
3. Tromp, J., Broek, J.v.d.: Er wordt absurd veel bezuinigd. Interview Johan Witteveen, oud-directeur IMF. De Volkskrant, 7 september 2012 pag. 25
4. Witteveen, H.J.: De magie van harmonie. Een visie op de wereldeconomie. Gibbon Uitgeefagentschap (2012)
5. Giebels, R.: We pesten onszelf meer dan nodig is. De Volkskrant, 14 maart 2013 pag. 7
6. Berentsen, L.: CPB laakt extra bezuinigingen. Het Financiële Dagblad, 14 maart 2013 pag. 1
7. Swierstra, S.D.: Construct your own favorite programming language. Technical Report UU-CS-2009-029, Department of Information and Computing Sciences, Utrecht University (2009)
8. Grijpink, J.H.A.M.: Keteninformatisering, met toepassing op de justitiële bedrijfsketen. Sdu Uitgevers (1997)
9. Grijpink, J.H.A..: Keteninformatisering in kort bestek. Theorie en praktijk van grootschalige informatie-uitwisseling (2e druk). Boom/Lemma Uitgevers (2010)
10. Grijpink, J.H.A.M., Plomp, M.G.A., eds.: Kijk op ketens. Het ketenlandschap van Nederland. Centrum voor Keteninformatisering (2009)
11. Grijpink, J.H.A.M.: Chain analysis for large-scale communication systems. Journal of Chain-computerisation **1** (2010)
12. Grijpink, J.H.A.M.: A chain perspective on large-scale number systems. Journal of Chain-computerisation **3** (2012)

Figuur 7: Jan Grijpink, 2013. Over de grenzen van onze kennis, acryl op paneel 50x70

Aan Doaitse

Corine de Gee

Bijna 30 jaar geleden kwam ik bij de Vakgroep Informatica, waar jij toen net een half jaar hoogleraar was. Volgens eigen zeggen: “omhooggevallen door gebrek aan gewicht.” Maar dat was natuurlijk onzin.

Wat ik enorm in je waardeer is dat je altijd klaar staat om te helpen. Een hele zondag via iChat hulp bieden bij computerproblemen. Een hele avond besteden aan het updaten van mijn computer. “Daar heb je vrienden voor”, zeg je dan. En je kunt altijd van alles bij je lenen, “als het maar gebruikt wordt”.

En gelukkig ben je altijd een beetje kinderlijk gebleven. Liefst even pootje haken in de gang als iemand voorbij loopt en genieten van een dagje Efteling, hoewel dat ook alweer heel lang geleden is.

Ik heb veel gehad aan je talloze financiële adviezen. “Nooit aandelen kopen van een product dat je zelf niet begrijpt. Geen verzekeringen afsluiten voor zaken, die je zelf kunt betalen.” En nog meer van dat soort nuttige raad.

Veel van je ideeën waren heel handig en nuttig. Soms moest ik het even laten bezinken om tot de conclusie te komen dat je weer eens gelijk had. Wat werd je er soms moe van dat veel bestuurders aan de universiteit maar niet begrepen dat je ideeën niet buitensporig of onuitvoerbaar waren. Je dacht alleen meestal een paar stappen vooruit en niet iedereen kon dat volgen.

Van de wijze raad die je aan studenten gaf is mij altijd bijgebleven de uitspraak “Niet studeren en wel collegegeld betalen is als naar de slager gaan, een kilo gehakt kopen, afrekenen en zonder vlees vertrekken”. Ik zeg het nog wel eens tegen studenten.

Met veel verantwoordelijkheidsbesef begeleidde je studenten. Dat ging zelfs zover dat je een goed contact had opgebouwd met de psychiater van een van je afstudeerders (ik zal zijn naam niet noemen).

Mijn baan als studieadviseur heb ik grotendeels aan jou te danken. We werken de laatste jaren niet meer bij elkaar in de buurt, maar als je even een praatje komt maken is het altijd weer lachen. Ik zal dat missen.

Ik kom graag een keer langs bij jou en Agnes in Tynaarlo, zodat we weer eens ouderwets kunnen bijpraten en lachen.

The Era of Doaitse

Jeroen Fokker

J.D.Fokker@uu.nl
Dept. of Computer Science, Utrecht University

It was a memorable day in the spring of 1983, when Sietse van der Meulen made a special announcement to the participants of the course on programming languages (of which I was one): the department had decided on the appointment of a second chair in computer science. Two features of the new professor were highlighted by Sietse: that his field was the one that we all liked (structure of programming languages and compiler construction), and that he was Frisian. Little could I know that I would study, and later co-work, with Doaitse for the next 30 years.

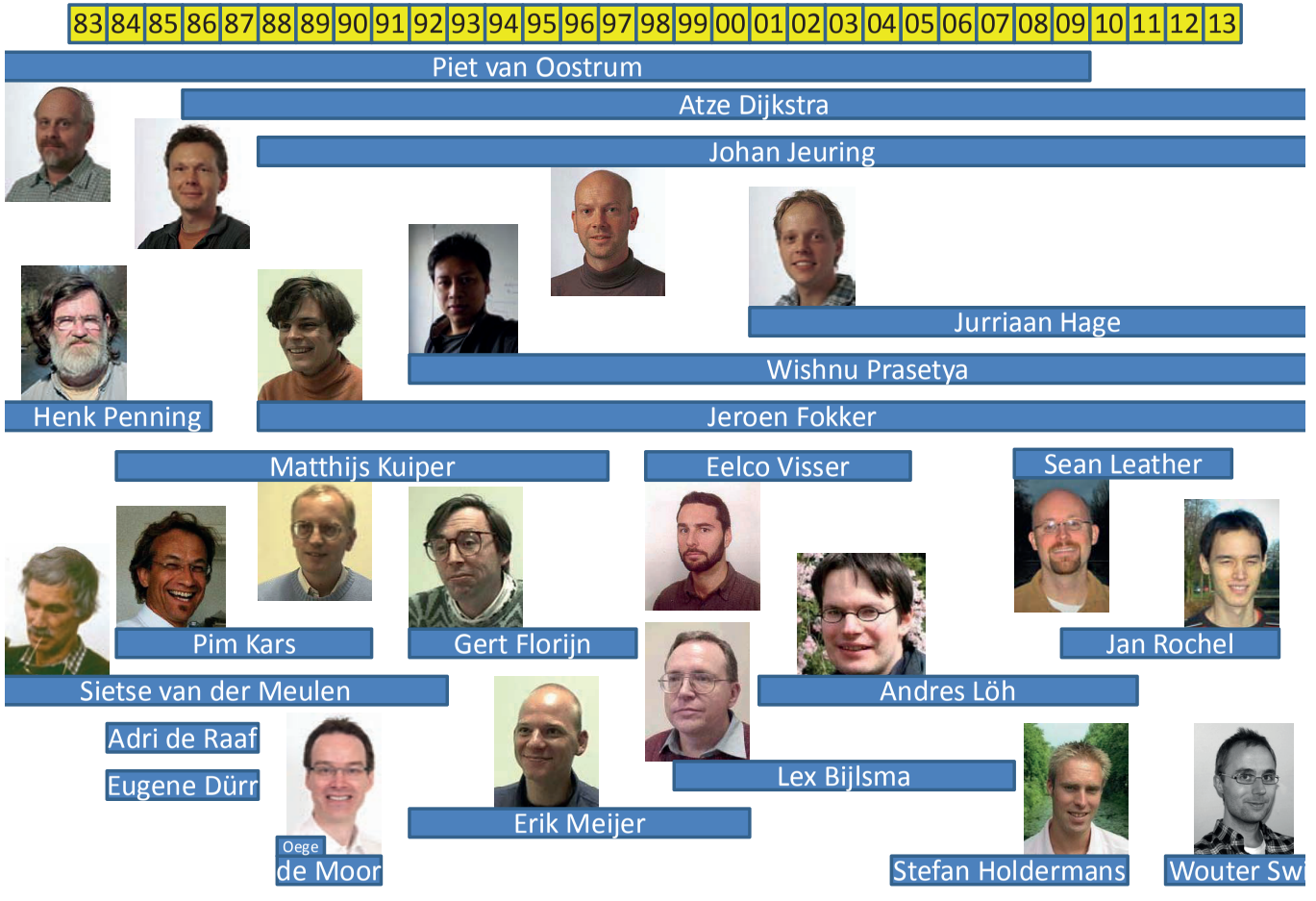
The advent of Doaitse was one of the preconditions for the start of a major in computer science in 1983, the other being an own Vax computer for the department. Consequently, Doaitse's career at Utrecht University co-evolved with the curriculum. In the same way as the structure of the universe as we know it today was determined in the first years after the big bang, the characteristics of the Utrecht CS curriculum were shaped in those pioneer years. For instance, the emphasis on functional programming, hated yet loved by so many generations of students, was introduced by Doaitse at the very start.

I have described the history of the department and the curriculum on the occasion of the 25th anniversary of the curriculum [1] — coinciding with the 25th anniversary of Doaitse's professorship. So let me focus this time on the history of the research group that formed around Doaitse's chair on Software Technology: the 'ST-group'. The first members of the group were already floating in the primordial department soup: Sietse van der Meulen, Piet van Oostrum and Henk Penning. In one of the department's growth spurts in the late '80s, Atze Dijkstra, Johan Jeuring and I joined, and we had the honor of serving with Doaitse until the present day. Likewise, Lambert Meertens was a group member for most of Doaitse's era.

In total, about 15 faculty and 35 research assistants have been part of the ST-group. They are shown on the time line on the next pages. All of them, and numerous students and short-term guests as well, have been imprinted with how software issues should ideally ('eigenlijk') be addressed: through generalization, parameterization and higher-order concepts. These ideas will continue to shape the CS-universe far beyond Utrecht and far beyond the past 30 years.

References

1. Fokker, J.: Geschiedenis van de opleiding informatica aan de universiteit Utrecht. In Bodlaender, H., ed.: Fascination for computation – 25 jaar opleiding informatica. Department of Computer Science, Utrecht (2008) 7–31


83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 00 01 02 03 04 05 06 07 08 09 10 11 12 13
Piet van Oostrum
Atze Dijkstra
Johan Jeuring
Jurriaan Hage
Wishnu Prasetya
Henk Penning
Jeroen Fokker
Matthijs Kuiper
Eelco Visser
Sean Leather
Pim Kars
Gert Florijn
Jan Rochel
Sietse van der Meulen
Andres Löh
Adri de Raaf
Eugene Dürr
Lex Bijlsma
Oege de Moor
Erik Meijer
Stefan Holdermans
Wouter Swi

83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 00 01 02 03 04 05 06 07 08 09 10 11 12 13
Lambert Meertens
Maarten Pennings
Martijn Schrage
Nico Verwer
Daan Leijen
Alexey Rodriguez
Ruud Koot
Bastiaan Heeren
Pedro Magalhães
Amir Saeidi
Arie Middelkoop
Tanja Vos
Pablo Azero
Eelco Dolstra
Karina Olmos
Martin Bravenboer
Jeroen Bransen
Alex Elyasov
Harald Vogt
Frans Rietman
João Saraiva
Arthur Baars
A.v.Leeuwen
Marcos Viera
Vargas Cardona
Rob Udink
Frank Atanassov

Herenleed

Fiets jij wel hard genoeg?

Huis, spaarhypotheek, annuïteitenhypotheek, provisie, hypotheekrente-aftrek, box 1, Tynaarloo, kadastrale kaart, Zandlust, Zandbad, bouwkundig keuring, zonnepanelen, makelaar, overdrachtsbelasing, kosten koper, politiek, Funda, Nuenen, Ameland, Wageningseberg, Groningen, Open Huizen Dag, Norg.

Robotmaaier, Conrad, spanningsregelaar, Misco, DealExtreme, Arduino, zon zoekt dak, color laser CLX-3305FW, ProtoSpace, Fablab, Stickuino, pcb, 220 Ohm, diode, 1N4007, soldeerbout, tin, condensator, printdrukschakelaar, resonator, header, male, female, 8 pins, microcontroller, D-Link WiFi versterker, DIR-505/E.

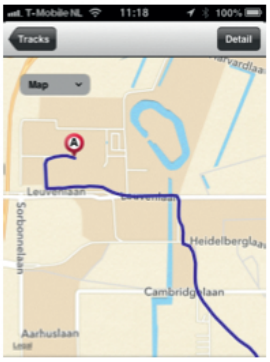
Agnes, broertje, botter, Tsjalling, neefje, Wouter, Rogier, Edsger. Vanilleijs, warme kersensiroop, bastognekoeken, slagroom, nachtje in de vriezer, forel, biefstukje, vette jus.

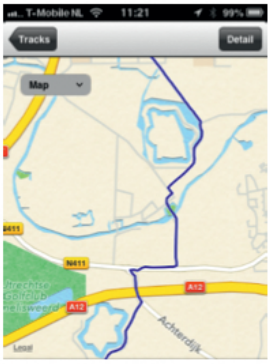
Gazelle, ketting smeren, bevroren versnelling, karakteristieke witte haarkrans, weesfiets, Nieuw Wulven, Vrienden op de Fiets, brede banden, fietsenmaker aan huis, meefietsen?

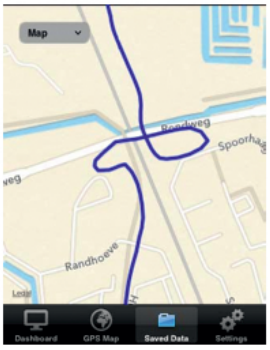
NGI, concernarchitectuur belastingdienst, functional programming, Haskell, Dazzle, summer school, AFP, BioHaskell, evangelist, Hackage, Montevideo, Vector Fabrics, Mircrosoft Research, Silicon Hive, HUG.

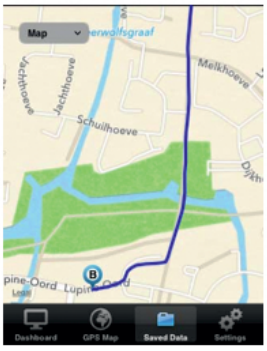
Examencommissie, zwaartepunten, logo, proefschrift, examengeld, rector, 40-jarig jubileum, domjudge, promotierecht, quorum, oppositie, strategische thema's, VSNU, trendrapportage, student/staff ratio, emiraat, emeritaat, afscheidsrede, aftellen, Pandhof, Research In Peace.

Remco Veltkamp









The Workroom

Atze Dijkstra

atze@uu.nl
Dept. of Computer Science, Utrecht University

Abstract. Doaitse has worked for many years in many workrooms. Many things changed during those years, but also many things have remained the same. These artefacts and other signs of workroom invariants can still be observed in Doaitse's work habitat, The Workroom.

At first sight The Workroom appears a rather normal place (Figure 1), chairs, table, lots of books, etc.. However, already this first look at his room already shows Doaitse's preference for lateral thinking and alternate views on common problems. For example, his work position seems to be floating in the air, just in front of the computer screen in the upper right corner of the room, simultaneously laterally magically also in front of the screen in the left shelves. It is such deep insights into the invariants of Doaitse's life that we intend to expose by means of actual sights into The Workroom.

Fig. 1. Doaitse's workroom

Before proceeding to further observations we first need to ensure that indeed the common workroom from Figure 1 is The Workroom. Careful inspection leads us to certificates from both faraway and close to home locations. From Figure 2 we can see that also from the more southern parts of our world the workroom was considered to be The Workroom. Similarly,

Fig. 2. Something Bolivian testifying

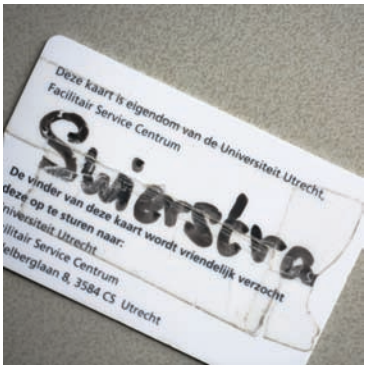


Fig. 3. Doaitse's coffee card

Fig. 4. Once coffee in a plastic cup

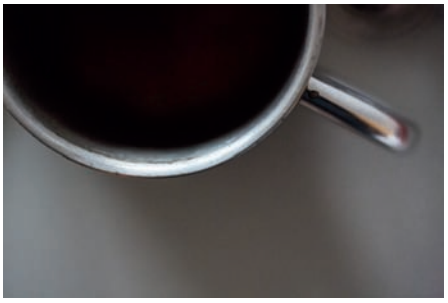


Fig. 5. Metal coffee mug

from Figure 3 we see the close connection between Doaitse referring to himself via authentic (non computerized) handwriting, the Utrecht University, and his coffee card.

The coffee card also points out an intriguing mode of working. For example, take the remains in the unpretentious plastic cup in Figure 4. This must be the remains of coffee, a clear indication that Doaitse's insights into future trends and research directions stem from the old habit of "koffiedik kijken" (reading tea-leaves). Further proof of this can be found in Figure 5 showing a more solid metal hightech approach to such matters of divination. The broadness of his resulting vision is embodied in the more down to earth mug in Figure 6 making explicit his grounding in clay and explicitness of mentioning his exercised coffee methods for observing truth yet to come.

Fig. 6. Brown coffee mug

Fig. 7. Attribute Grammars

Out of the coffee grounded work in The Workroom at least two notable results have sprung. One of these can be directly observed: it has been made quite explicit over the years by gently coercing PhD students to endlessly write about it. Take for example Figure 7: can it be more clear which major concern can be attributed to The Workroom? We need not spend more words on this other than pointing out an arbitrary whiteboard drawing of a tree in Figure 8.

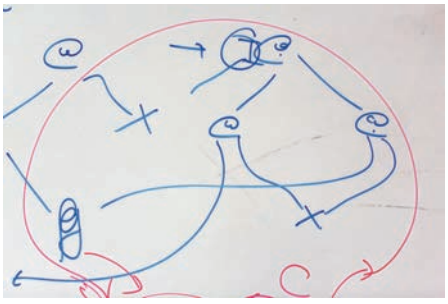


Fig. 8. A tree on whiteboard

Fig. 9. Parser on whiteboard

Further inspection of writings on whiteboard and paper directly reveal the second important matter in the life of The Workroom: parsers. And not just parsers, but what a richly colorful decoration accompanies this important matter! Observe the beautiful (red) encirclement of a parser in Figure 9 by which its importance is emphasized over and over again!

Not only was parser inspiration conveyed by means of whiteboard to nearby colleagues, also students had the privilege to submit writing for further decorative enhancement, as testified of in Figure 10.

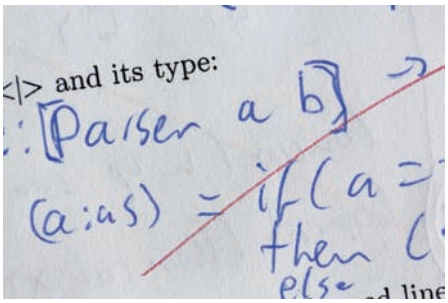


Fig. 10. Corrected parser on exam

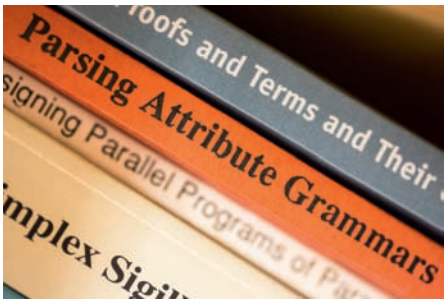


Fig. 11. Parsing Attribute Grammars

From Figure 11 it can readily be observed that these two supposedly different topics are actually intertwined. And indeed, why two when one suffices? The Workroom exemplifies the quest for the holy grail to have a program be as small as possible, preferably requiring only 1 bit to express intent and meaning, leaving it up to a compiler to derive the other required bits.

Outside of The Workroom I will miss this strive for simplicity, and bringing matters back to their core. It always gave me a big grin on my face. The landscape from The Workroom (Figure 12) sadly will never be the same.

Fig. 12. Rainy view from the workroom

Fun with Robots, CoCoCo and Mate

Alberto Pardo and Marcos Viera

Instituto de Computación, Universidad de la República
Montevideo, Uruguay
{pardo,mviera}@fing.edu.uy

Abstract. We develop an extensible domain specific language for robotics in order to solve a classic problem of the uruguayan society. We hope Doaitse will enjoy, or at least will not suffer much, reading this paper.

1 Introduction

It is beyond discussion that traditions are an important part of our lives. That happens in every culture and region all around the world. They condition our way of life, our beliefs and arts, and also our view of the world. Traditions have also to do with our alimentary habits. Each folk has its own meals and drinks that distinguishes it. Examples are numerous, going from Italian *pizza* to Mexican *tacos*, from English tea to French wine.

In the so-called Southern Cone, the geographic region comprised by Argentina, southern Brazil, Chile, Paraguay, and Uruguay, there is a popular drink called *mate*. Mate is prepared from steeping dried leaves of *yerba mate* (mate herb) in hot water. It is served with a metal straw (called *bombilla*) from a shared hollow calabash gourd (also called *mate*). Its preparation involves a careful arrangement of the yerba within the gourd before adding hot water. Mate has a strong, bitter taste one gets used to with time. Of course, some people affirm that "it smells like grass" [1].

Traditionally, mate was drunk by the *gauchos* (traditional residents of the South American pampas), but with time it became popular also in the cities. Figure 1 shows a typical Uruguayan gaucho with his mate on Punta del Este beach.

Fig. 1. Typical uruguayan gaucho with his termo and mate.

In Uruguay you can find people drinking mate everywhere, and at any time: at work, at the street, at the Stadium, at the beach, at live concerts, even in classrooms. The mem-

bers of the Functional Programming team at the Instituto de Computación (InCo) of the Universidad de la República in Montevideo are not an exception. They are of course usual mate drinkers like most uruguayans. They not only drink mate at InCo, but, moreover, they cannot start any team meeting until a mate is prepared for the occasion.

Mate is a shared drink: the same gourd and straw are used by the group of persons that are drinking together. One person, known as *cebador*, assumes the task of server. The cebador subsequently refills the gourd with water and passes it to a drinker, who, after finishing drinking, passes the gourd back to the cebador. This process is repeated many times until the mate loses its flavour. In that state the mate is said to be *lavado* (washed out). Passing a mate to a drinker may sometimes require to pass it through many hands until it reaches the drinker (for example, when various persons are drinking together), or to stand up to bring the mate either to a drinker or back to the cebador (for example, when cebador and drinker are not close to each other). This process of passing a mate is a continuous source of distractions, mainly when you are drinking mate at work.

Motivated by this fact, the members of the FP team at InCo started investigating the possibility of applying some technological solution to this really important and critical problem. They knew they were facing a really hard task, but as return, if they succeeded, they would receive the gratitude of many people. As every research project in a completely new subject, the first steps were by no means easy. They found a lot of literature about mate, such as a study about the psychological traumas many cebadores get because of the excessive time some drinkers spend to drink a single mate!! [2], but nothing related with the application of technology to the mate rituals.

One day, after a long period of deep investigation, when they thought they had finally failed, unexpectedly, they realized the solution was not too far! In fact, it had been at hand all the time! The solution was in the dark and misterious workshop inhabited by members of InCo's MINA group that work on robotics. Yes! A robot was the solution!

Robotics is one of the activities of the MINA group. For years they have been organizing with much success a yearly SUMO robot tournament called SUMO.UY[1]. Using the infrastructure of the *Ceibal plan*[2], an Uruguayan initiative that implements nation wide the OLPC[3] model at primary and secondary public schools, members of the MINA group developed the *Butiá*[4] platform for low-cost easy-to-program robots to be used at schools [3].

The solution to the mate-delivery problem was then simple:

A robot would be used to carry a mate among cebador and drinkers.

As a consequence, none had to move much to drink mate. The robot would transport a mate from cebador to each drinker and back. This shows that, like in FP, a bit of laziness is always helpful. The implementation was carried out on a Butiá robot controlled by a Ceibal laptop like the one shown in Figure 2.

However, when they started playing with the robot the members of InCo's FP team faced immediately another problem. The robot library that was running on the Ceibal computer was written in Python! That meant that they had to write code in that language. They said, never! So they decided to develop an extensible functional-like DSL to interact with the Butiá robot. This language would then be compiled to Python.

[1] `http://www.fing.edu.uy/inco/eventos/sumo.uy/`

[2] name that comes from *ceibo*, the tree that produces the Uruguayan national flower.

[3] One Laptop Per Child project developed by MIT where laptops are delivered to school students.

[4] Butiá is a type of palm native to Argentina, Brazil, Paraguay and Uruguay. Butiá is also the plea an Uruguayan shouts when his/her computer takes too long time to boot; he/she says *butiá por favor!*, that means "please, boot!".

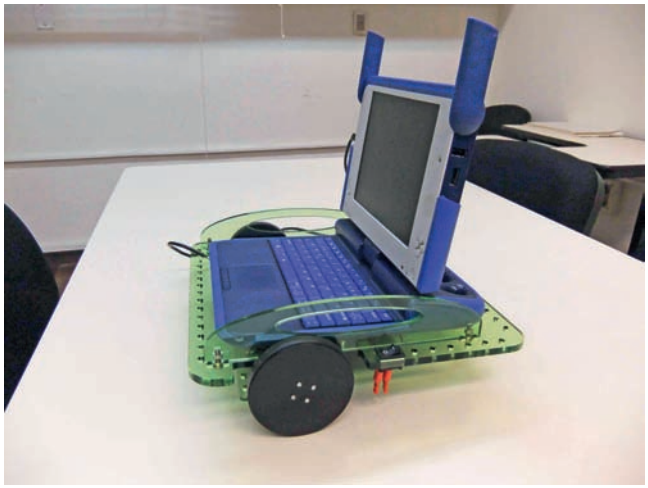


Fig. 2. Butiá robot controlled by a Ceibal computer.

An extensible compiler using the CoCoCo[5] tool set was developed. The tools and libraries composing CoCoCo were designed and implemented in the context of Viera's PhD thesis [4]. It joins a set of non trivial problems that Doaitse used to think about during his daily bike trips ... and made Marcos lose an important part of his hair.

CoCoCo makes it posible to define language fragments that can be individually type-checked, distributed and composed to construct a compiler. Using this technology, different versions of the language were defined. The basic version (Section 2) implements a notion of tasks and procedures where the robot is only able to move and wait for a given amount of time. A second version improves on the first one by allowing tasks to be arguments of procedures (Section 3). Another version (Section 4) introduces conditional execution of tasks. Finally, a couple of extensions are defined to allow the use of sensors (Section 5) to let the robot receive information from the environment.

The design of the compiler for the present language is a bit different from the suggested in the examples of [4] (e.g. Oberon0), where the extensions are incremental. In this case most of the extensions are defined to extend the basic language, not depending on previous extensions. Thus, multiple versions of the language can be defined by combining different extensions. The only exceptions are the extensions involving sensors.

In this paper we will not describe the technical details concerning CoCoCo. If you are not Doaitse and still want to understand the implementation of the compiler we refer you to Viera's thesis.

2 Basic Language

We start with a very simple language that allows us to describe the basic movement operations a robot can make. We will explain the language while introducing its concrete syntax grammar (expressed in EBNF notation).

[5] `http://www.cs.uu.nl/wiki/Center/CoCoCo`

2.1 Grammar

A program starts with the keyword **programa**, followed by the name of the program and a set of procedure declarations describing the tasks the robot has to perform. A procedure is declared using the keyword **tarea**. To define a procedure we have to provide its name, formal parameters and body (*tasks*). The execution of a program starts in the last procedure, called `inicio`.

prog = `"programa"` *var decls main*
decls = { *procD* }
procD = `"tarea"` *var fparams* `"="` *tasks*
main = `"inicio"` *fparams* `"="` *tasks*

The parameters of a procedure can be of type integer (**entero**) or boolean (**booleano**), which are the only types of the basic language.

fparams = `"("` [*fparaml*] `")"`
fparaml = *fparam* { `","` *fparam* }
fparam = *var* `":"` *typ*
typ = `"entero"` | `"booleano"`

The tasks our robot can perform are:

- move forward (**avanzar**) or backward (**retroceder**) at a given (motor) speed
- turn right (**derecha**) or left (**izquierda**) at a given (motor) speed
- wait (**esperar**) for a certain time (in seconds) till performing the next task
- stop (**parar**) moving
- perform a task described in a procedure with name *var* given the arguments *params*.

Tasks can be sequenced using the operator `>>>`.

tasks = *task* { `">>>"` *task* }
task = `"avanzar"` *exp* | `"retroceder"` *exp*
| `"girar"` `"derecha"` *exp* | `"girar"` `"izquierda"` *exp*
| `"esperar"` *exp* | `"parar"`
| *var params* | `"("` *tasks* `")"`
params = `"("` [*paraml*] `")"`
paraml = *param* { `","` *param* }
param = *exp*

The arguments we pass to a procedure or task are expressions, including:

- integer comparison (=, #, <, <=, > and >=)
- integer binary operations (+, −, ∗, / and %)
- integer unary operations (+ and −)
- boolean binary operations (| and &)
- boolean unary operations (`no`)
- boolean (`verdadero` and `falso`) and integer (*int*) literals
- variables (*var*)
- parenthesis

with the usual operator precedences, as expressed by the grammar:

$$
\begin{aligned}
\mathit{exp} &= \mathit{sexp} \\
&\mid \mathit{exp}\ \texttt{"="}\ \ \mathit{sexp} \mid \mathit{exp}\ \texttt{"\#"}\ \mathit{sexp} \mid \mathit{exp}\ \texttt{"<"}\ \mathit{sexp} \\
&\mid \mathit{exp}\ \texttt{"<="}\ \mathit{sexp} \mid \mathit{exp}\ \texttt{">"}\ \mathit{sexp} \mid \mathit{exp}\ \texttt{">="}\ \mathit{sexp} \\
\mathit{sexp} &= \mathit{signed} \\
&\mid \mathit{sexp}\ \texttt{"+"}\ \mathit{signed} \mid \mathit{sexp}\ \texttt{"-"}\ \mathit{signed} \mid \mathit{sexp}\ \texttt{"|"}\ \mathit{signed} \\
\mathit{signed} &= \texttt{"+"}\ \mathit{term} \mid \texttt{"-"}\ \mathit{term} \mid \mathit{term} \\
\mathit{term} &= \mathit{factor} \\
&\mid \mathit{term}\ \texttt{"*"}\ \mathit{factor} \mid \mathit{term}\ \texttt{"/"}\ \mathit{factor} \\
&\mid \mathit{term}\ \texttt{"\%"}\ \mathit{factor} \mid \mathit{term}\ \texttt{"\&"}\ \mathit{factor} \\
\mathit{factor} &= \texttt{"verdadero"} \mid \texttt{"falso"} \mid \mathit{var} \mid \mathit{int} \mid \texttt{"("}\ \mathit{exp}\ \texttt{")"} \mid \texttt{"no"}\ \mathit{factor}
\end{aligned}
$$

2.2 mate-delivery, A first solution

Fig. 3. The process of sending a mate to an officemate.

Now that we have the basic language, we can propose a first naive solution to the mate-delivery problem.

```
programa mate1

tarea llevarmate (tiempo : entero)
    = avanzar 50 >>> esperar tiempo >>> parar >>> esperar 20 >>>
      traermate(tiempo)
```

```
tarea traermate (tiempo : entero)
    = retroceder 50 >>> esperar tiempo >>> parar >>> esperar 45 >>>
      llevarmate(tiempo)

inicio () = llevarmate (10)
```

The procedure `llevarmate` transports the mate from the cebador to his colleague while the procedure `traermate` brings it back to the cebador. Both procedures are parametrized by the time it takes to the robot to move from the start to the endpoint. With `llevarmate` the robot moves forward (**avanzar** 50, when 100 is the maximum motor speed) for `tiempo` (in this case 10) seconds. Then stops and waits for 20 seconds to let the colleague drink the mate. Figure 3 shows these steps. After that the `traermate` procedure is called to perform the task of bringing the mate back to the cebador, giving him 45 seconds to serve and drink his own mate and serve another one to his colleague. The complete task is repeated till forever and a day.

Notice that the robot behaves in the same way public transport does; i.e. it will not wait if one of the actors is not in time to put the mate on the robot when required. We will come back to this issue in later solutions to the problem.

2.3 Compiler

Figure 4 shows a fragment of the concrete syntax grammar represented as a Haskell value, using the `murder` [5] library. We basically reproduce the concrete syntax introduced before, associating it with the *semantic functions* provided by the parameter *sf*; a record with one field per each production of the AST of the language. The compiler is generated by a function called *closeGram* that transforms this grammar into a left-recursion free one, and builds a `uu-parsinglib` parser out of it.

We use the `AspectAG` [6] embedding of attribute grammars in Haskell to implement the semantics of the language. The semantics include type-checking and code generation. We generate Python code to interact with the robot through its API[6]. For example, the following is a fragment of the Python code produced by our `mate1` example:

```
import butiaAPI

robot = butiaAPI.robot()

def llevarmate(tiempo):
        robot.set2MotorSpeed(0, 50, 0, 50)
        time.sleep(tiempo)
        robot.set2MotorSpeed(0, 0, 0, 0)
        time.sleep(20)
        traermate(tiempo)
```

The translation is quite trivial. Procedures are translated to Python procedures and tasks to statements. Movement tasks are translated to motor speed settings. For example, **avanzar** 50 is translated to `robot.set2MotorSpeed(0, 50, 0, 50)` that means: make left and right motor go forward with speed 50. In case of the task **girar derecha** 50 the generated code would be `robot.set2MotorSpeed(0, 50, 1, 50)`.

[6] `http://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php/APIenglish`

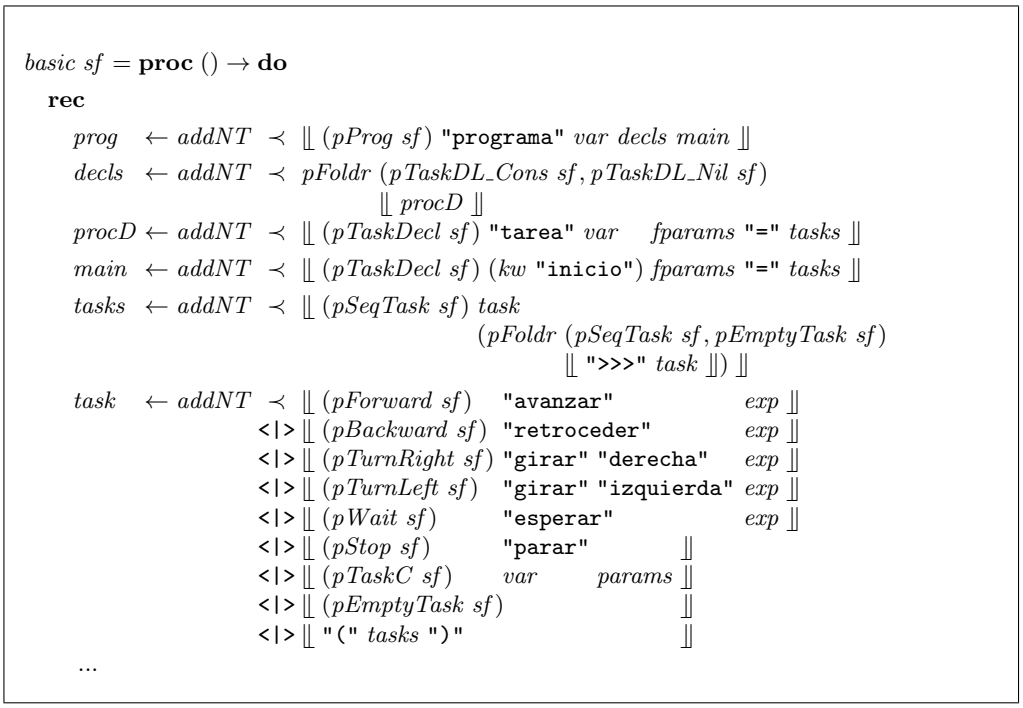


Fig. 4. Fragment of the concrete syntax specification of the basic language

3 Tasks as Arguments

We extend the language by allowing tasks to be passed as parameters of a procedure. Thus, arguments can be either (integer or boolean) expressions or tasks. The keyword **tarea** indicates when a parameter is a task.

fparam = ... | `"tarea"` *var*
param = ... | *tasks*
task = ... | *var*

Inside the procedure a variable bound to a task parameter is treated just as a normal task. Thus, task parameters are evaluated in a call-by-name way, performing the task every time it is used.

With this extension we can rewrite our mate-delivery task in a more elegant way.

```
programa mate2

tarea siempre (tarea cuerpo)
    = cuerpo >>> siempre (cuerpo)

tarea trasladar (tarea dir, tiempo : entero)
    = dir >>> esperar tiempo >>> parar >>> esperar 20

inicio () = siempre (trasladar (avanzar 50, 10) >>>
```

```
                    trasladar (retroceder 50, 10) >>>
                    esperar 25)
```

We define a procedure `siempre`, to repeat the task forever. The behaviour of passing the mate is now encoded in the procedure `trasladar`, which is parametrized by the task that moves the robot in a given direction.

We show in Figure 5 how we implement the grammar extension by adding productions (with *addProds*) to the corresponding non-terminals.

```
targ sf = proc imported → do
  let task    = getNT cs_Task imported
  let tasks   = getNT cs_Tasks imported
  let param   = getNT cs_Arg   imported
  let fparam = getNT cs_Param imported
  rec
    addProds ≺ (fparam, ⟦ (pTaskParam sf) "tarea" var ⟧)
    addProds ≺ (param,  ⟦ (pArgTask   sf) tasks      ⟧)
    addProds ≺ (task,   ⟦ (pIdTask    sf) var       ⟧)
  exportNTs ≺ imported
```

Fig. 5. Extension to allow tasks to be passed as arguments

Since Python is a strict language and it does not accept statements as procedure parameters, we had to implement the behaviour of our language by using internal helper procedures. For example, the following code fragment:

```
tarea siempre (tarea cuerpo)
   = cuerpo >>> siempre (cuerpo)

inicio () = siempre (parar >>> esperar 20)
```

is translated to the following Python code:

```
def siempre(cuerpo):
      cuerpo()
      def ptask1():
            cuerpo()
      siempre(ptask1)

def inicio():
      def ptask2():
            robot.set2MotorSpeed(0, 0, 0, 0)
            time.sleep(20)
      siempre(ptask2)

inicio()
```

For each task argument in a procedure call, we generate a Python procedure with fresh name and the task argument as body. In the example above, the calls `siempre (cuerpo)` and `siempre (parar >>> esperar 20)` produce the generation of the procedures `ptask1` and `ptask2`, respectively. Task parameters, like `cuerpo`, are then translated to parameters of type procedure. We evaluate these (parameter) procedures in the body of our Python procedure by simply calling them (e.g. `cuerpo()`).

4 Conditionals

With the constructions we have thus far, we are condemned to drink mate till the end of the robot's life. We therefore extend the language to add some conditional execution primitives.

A *guarded* task, [*exp*] *task*, is only executed if the boolean expression *exp* evaluates to `verdadero`; otherwise it behaves like a skip. Thus we can change in the mate-delivery example the `siempre` infinite repetition by a finite `repetir`, which is parametrized by the number of repetitions it has to perform (or mates we want to drink).

```
tarea repetir (veces : entero, tarea cuerpo)
    = [veces > 0] (cuerpo >>> repetir (veces−1, cuerpo))
```

Non guarded tasks can be thought of as guarded tasks with guard [`verdadero`]. Two tasks `t1` and `t2` can be combined with the alternative operator `t1 <|> t2`. It first tries to execute `t1`; if its guard evaluates to `verdadero` that completes the execution of this task. Otherwise it executes `t2`.

Since we want >>> to have precedence over <|>, we extend (and adapt) the grammar in the following way:

tasks = *tasks′* { `"<|>"` *tasks′* }
tasks′ = *task* { `">>>"` *task* }
task = ... | `"["` *exp* `"]"` *task*

Notice that the former non-terminal *tasks* now corresponds to *tasks′*. In Figure 6 we show how this is implemented. We replace the productions of the former non-terminal *tasks* (by applying *replaceProds*), we create the new non-terminal *tasks′*, and add a new production to *task*.

5 Adding Sensors

As we pointed out in our first solution to the delivery-mate problem, the robot was a bit unpolite with slow mate drinkers. This is because the robot was not able to receive information from its environment.

We now extend our language with a primitive **sensar en**, to sense the enviroment using a specific sensor, and bind its value to a variable.

task = ... | `"sensar"` *sensor* `"en"` *var*
sensor = `"gris"`

The first sensor we add to the Butiá robot is a grayscale sensor. A grayscale sensor (**gris**) is a light meter that measures the light level at a certain point and returns a value in a certain scale. As shown in Figure 7, we use the grayscale sensor to detect the presence of the mate

```
opt sf = proc imported → do
  let task  = getNT cs_Task  imported
  let tasks = getNT cs_Tasks imported
  let exp   = getNT cs_Exp   imported
  rec
    replaceProds ≺ ( tasks
                   , ⟦ (pOptTask sf) tasks'
                                     (pFoldr (pOptTask sf, pEmptyTask' sf)
                                             ⟦ "<|>" tasks' ⟧) ⟧)
    tasks'       ← addNT ≺
                     ⟦ (pSeqTask' sf) task
                                     (pFoldr (pSeqTask' sf, pEmptyTask' sf)
                                             ⟦ ">>>" task ⟧) ⟧
    addProds     ≺ (task, ⟦ (pCondTask sf) "[" exp "]" task ⟧)
  exportNTs ≺ imported
```

Fig. 6. Extension to add conditionals

on the robot. We can then write a new solution to the mate-delivery problem that, instead of waiting for the mate for a fixed amount of time, the robot keeps waiting until the mate is placed over the sensor.

```
programa mate3

tarea trasladar (tarea dir, tiempo : entero)
   = sensar gris en haymate >>>
       ([haymate > 90] (dir >>> esperar tiempo >>> parar)  <|>
        esperar 2 >>> trasladar (dir, tiempo))


inicio () = repetir (10, trasladar (avanzar     50, 10) >>> esperar 10 >>>
                         trasladar (retroceder  50, 10) >>> esperar 10 )
```

The robot starts to move only when the grayscale is greater than 90 (meaning that the mate is over the sensor). Otherwise, it waits for a couple of seconds to sense again.

This works well with one cebador and one drinker, and can easily be extended to more than one drinker with fixed turns. But, what happens if drinking turns are not fixed?

To deal with this case we extend the language with a couple new sensors.

```
sensor = ... | "reloj" | "imagen"
typ    = ... | "trafico"
factor = ... | "PARE" | "CEDER" | "CONTRAMANO"
             | "ROTONDA" | "DERECHA" | "IZQUIERDA"
```

We use the computer clock as a time sensor (**reloj**) and the camera of the Ceibal computer as an image sensor (**imagen**). The MINA group has developed a pattern recognition library, that is able to recognize traffic signals, like Stop (`PARE`), Yield (`CEDER`), Not Enter (`CONTRAMANO`),

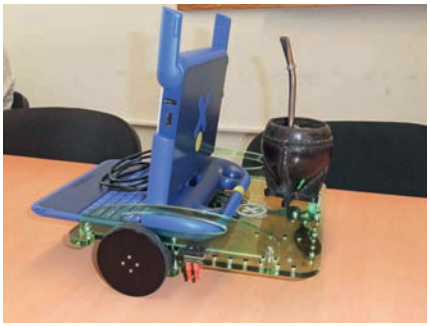


Fig. 7. Mate placed over grey sensor.

Fig. 8. Stop! This is my turn.

Round (`ROTONDA`), Turn Right (`DERECHA`) and Turn Left (`IZQUIERDA`), from images captured by the camera.

With these new sensors we can implement a new solution to the mate-delivery problem, which visits all the drinkers in a row until one of them shows a Stop signal, telling the robot that it is his/her turn.

```
programa mate4

tarea volver (tiempo : entero)
    = sensar gris en haymate >>>
        ([haymate > 90] (retroceder 50 >>> esperar tiempo >>> parar) <|>
         volver (tiempo))

tarea paso () = avanzar 50 >>> esperar 1

tarea llevarmate ( ini : entero )
    = paso >>> sensar reloj en fin >>> sensar imagen en img >>>
      ([img = PARE    ] (parar >>> esperar 10 >>> volver (fin–ini)) <|>
       [fin–ini >= 20] volver (fin–ini) <|>
       [fin–ini <  20] llevarmate (ini) )

tarea trasladar ()
    = sensar gris en haymate >>>
      [haymate > 90] ( sensar reloj en ini >>> llevarmate (ini) ) <|>
      (esperar 2 >>> trasladar ())


inicio () = repetir 10 trasladar ()
```

The procedure `trasladar` waits until the mate is on the robot, then it binds the inicial time to `ini` and calls `llevarmate`. The procedure `llevarmate` transports the mate executing steps at every second until:

- the robot sees the Stop signal, or
- all the drinkers have been visited and none has stopped the robot (it takes 20 seconds to move through the whole table)

In both cases the robot will return to the cebador (`volver`) when the mate is on it.

6 Conclusions and Future Work

Uruguayan office workers can be happy now, their main daily concern has finally been solved!

This work was just a first attempt at the design of an educative robotic language. Our plan is to achieve a design and implementation of the language mature enough to be distributed among Uruguayan secondary school students. The idea is to introduce them to (functional) programming concepts through this language, experimenting with them in a motivating context. Then we can bring Doaitse to Uruguay to use all his knowledge to help us in this difficult task. Our hope at the end is to contribute with our new generations making their life a bit less complicated...because, you know, *la vida en la ciudad es muy complicada*.

7 Acknowledgments

We thank the members of the MINA group, Andrés Aguirre, Gonzalo Tejera and Jorge Visca, for lending us a Butiá robot and a Ceibal computer. We also want to thank them for spending some time listening our weird ideas and explaining us how the robot works.

References

1. Swierstra, S.D.: Office conversation (2008)
2. Lenzina, J.: Don't teach the mate how to speak! - Treatise on cebador stress. Jounal of Intractable Nonexistent Diseases **8**(2) (1965) 120–385
3. Benavides, F., Aguirre, A., Otegui, X., Andrade, F., Tejera, G.: 1 adolescente 1 computadora 1 robot. In: World Engineering Education Forum - WEEF. (2012)
4. Viera, M.: First Class Syntax, Semantics and Their Composition. PhD thesis, Utrecht University, Department of Information and Computing Sciences (2013)
5. Viera, M., Swierstra, S.D., Dijkstra, A.: Grammar fragments fly first-class. In: Proceedings of the 12th Workshop on Language Descriptions Tools and Applications. ENTCS (2012)
6. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: How to do aspect oriented programming in haskell. In: ICFP'09: Proceedings of the 2009 SIGPLAN International Conference on Functional Programming. (2009)

Template-Based Document Generation Using Attribute Grammars

Pablo Ramón Azero Alcocer[1][2]

[1] pablo@memi.umss.edu.bo
Programa MEMI, Universidad Mayor de San Simón
[2] Pablo.Azero@jalasoft.com
Jalasoft

Abstract. A solution to a document generation problem based on templates is described. The solution is implemented using attribute grammars and functional programming. The approach can be used to generate documents in broader contexts.

1 Introduction

I will follow our program committee advice and propose a couple of conjectures raised by Doaitse Swierstra in the twenty two and something years that I know him:

Conjecture 1. There is an attribute grammar that is the solution to every problem in the world.

Conjecture 2. There is a functional program derived from an attribute grammar that is the solution to every problem in the world. The techniques used to derive the functional program from the attribute grammar take from 5 to 10 years to be formalized by the theoretical functional programmers.

I will not even try to prove those conjectures. It will suffice to say that I have an argument in favor of Conjecture 2: the pretty-printing combinators Doaitse and I wrote a long time ago. In this paper I will try to provide an argument in favor of Conjecture 1.

1.1 About this paper

As for the technical contents of this paper, it is organized as follows. Section 2 discusses the problem to be solved. The next section 3 presents an overview of a possible solution. Section 4 gives a detailed picture of the proposed solution describing the main component that is written in the Haskell and AG programming languages. The final section presents some conclusions.

2 The Problem

The problem relates to the generation of digital documents. The actual case of study implements an html document generator. Further experiments are being executed with the underlying ideas, but for the purpose of this paper, we restrain to the html document generator. See Figure 1 for a sample of such a generated document.
The main problem has the following characteristics:

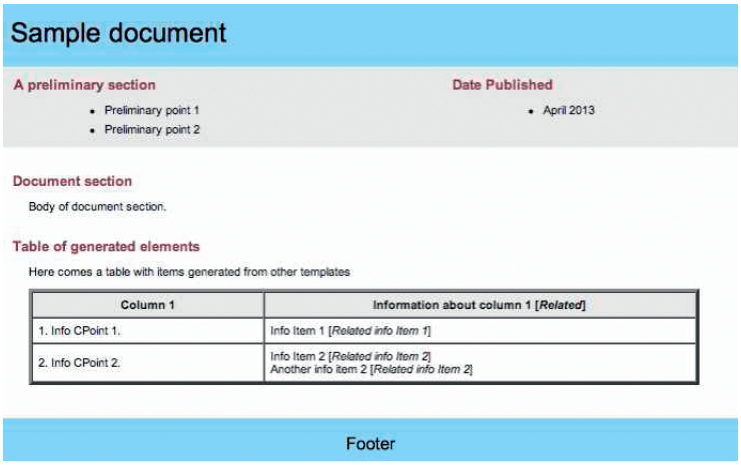


Fig. 1. Sample of a generated document

- An html document with updated documentation about changes in a software package distribution should be published periodically.
- The document contains a variable list of items that should be included. This information is produced by a machine. Say for instance is generated by some script in a build machine after having integrated the source code and compiled it successfully in a build.
- The layout of the document may be changed. Say for instance to adapt it to different products being distributed or because the institution wishes to update the document presentation. Nevertheless, the institution may wish to keep uniformity and consistence of all the documentation.

3 The Solution's Architecture

The solution has three main components as depicted in Figure 2:

- A *data generator script* that will encode the variable data output of the build machine in an AST[3] format. A context-free grammar (CFG) that encodes the structure of the data output has been defined. The script will generate a file containing the output data following the CFG rules.
- A set of *template files* (`tpl`) that contain the layout elements of the document should be defined. Every template file has a name corresponding to a non-terminal symbol of the CFG. Not all non-terminal symbols of the CFG have a correspondent template file. We only associate template files to the non-terminals that need a layout definition in the target html document. Additionally, there may be non-terminals that need several layout definitions because they have associated several rules in the CFG. In this case we need to associate names for every rule and a corresponding template file.
- A *template processor* program that will use as input the AST and template files and produce the resulting html document.

[3] AST: abstract syntax tree

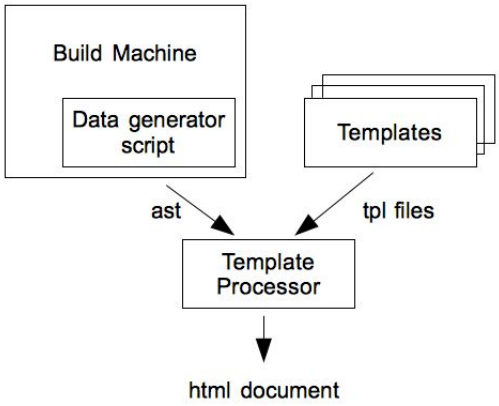


Fig. 2. Solution's architecture

3.1 The Structure of a Template File

As explained above, every template file contains the layout corresponding to a non-terminal node in the CFG. Inside a template file there could be references to child nodes. For instance, a template file (named `CPoint.tpl`) corresponding to a non-terminal called `CPoint` is:

```
<TR>

<TD>
%%Info%%
</TD>

<TD>
%%ItemsList%%
</TD>

</TR>
```

The identifiers enclosed in `%%` refer to child nodes.

4 The Template Processor

The template processor is the main element of the architecture. It is built out of three internal elements as depicted in Figure 3. The components are described in the following sections.

4.1 The Template Processor

Let's start describing the template processor. This is a Haskell component that takes every template file in the directory of templates and scans it collecting information to build the following data structure:

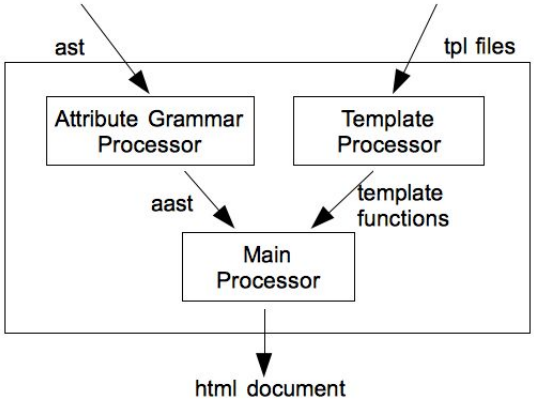


Fig. 3. The Template Processor

```
type Templates  = [ Template ]
data Template   = Template ParentNode TemplateFunction ChildrenNodes
type ParentNode = NodeName
type NodeName   = String

type TemplateFunction = ChildrenValues -> String
type ChildrenValues   = [(NodeName,String)]

type ChildrenNodes    = [ NodeName ]
```

This is, for every template, it collects:

- The name of the parent node (the name of the template file).
- A function that produces the corresponding html component for the parent non-terminal, given an association list of names and html components of the children.
- A list of names of the non-terminal children.

4.2 The Attribute Grammar Processor

The attribute grammar processor: (a) defines the CFG of the input data, and (b) attributes the input data AST producing the final html document as one attribute. It is of course written in Doaitse's AG language and compiled with the `uuagc` compiler.

The definition of the CFG for the document has two particularities:

- Every non-terminal that needs to be referred to in a template should have a name that is associated through the corresponding rule in the CFG. For example:

```
data Root
 | Root name::Name cPointList::CPointList

data CPointList
 | Cons name::Name cPoint::CPoint cPointList::CPointList
 | Nil
```

```
...

type PlainItemsList = [ Item ]
type PlainList2     = [ Element2 ]
```

Observe that not all non-terminals need to define a name, only those that can be referred to from a template or have defined a template. As for template names, we may have non-terminals that have more than one rule. In those cases we need to associate a name for every rule that needs a different layout.

– The document structure always start with a `Root` rule as the entry point.

The attributed tree gets at the root the list of templates and distributes it along the tree (an inherited attribute called `templateList`). A synthesized attribute (named `code`) will collect the html elements of its children. If the rule has an associated template the value of the attribute is produced by applying the template function to the association list, so that the corresponding html element is generated. For example, the html generation for a non-terminal looks like:

```
sem Item
 | Item loc.templateInfo   = lookupTemplate @name.name @lhs.templateList
        loc.parametersList = [ (@itemName.name,@itemName.code)
                             , (@itemAlias.name,@itemAlias.code)
                             ]
        lhs.code           = generateCode @templateInfo @parametersList
```

The `lookupTemplate` function will search along the template functions list if there is a template associated to that rule. If so it will apply the corresponding function to the produced layouts of the children. The function `generateCode` will do that.

In this version, simple verifications of the identifiers have been implemented. There is space for much more elaborated checkings, whether they are done in the template processor or in the attribute grammar definition.

4.3 The Main Processor

The main processor is as simple as calling the template processor, reading the input data and calling the attribute grammar processor:

```
main :: IO ()
main = do templateList <- getTemplates
          astString <- readFile inputFile
          let ast = (read astString) :: Root
          generateDocument ast templateList
```

The `generateDocument` function is nothing else than a straightforward call to the catamorphism of the `Root` node.

One of the main features of this program is that it doesn't need to parse over and over again the template files. It scans them once and then applies the corresponding function to every node that needs generation from such a template. This was one of the main problems found in approaches using traditional languages such as Java or C#.

5 Conclusion

Conjecture 1 has one more argument to survive! It may not be completely true, but we may believe so. Besides I enjoyed a lot building this program and remembered Doaitse's advice and words of wisdom. I'm sure that he will partly enjoy the program too, but may have many comments about its imperfections. I tried to describe the solution but not writing all the details so that there is room for imagination. I remembered Doaitse likes that.

Fig. 4. Doaitse and Pablo in Bolivia

Figure 4 shows some moments and places we enjoyed together, either working or just travelling. Thanks for all those great moments Doaitse and I wish you all the best in the second life you are starting from the date of your retirement.

Methodische plaatjes vullen geen gaatjes

Matthijs Maat and Gert Florijn

matthijs.maat@mxi.nl en gert.florijn@mxi.nl
M&I/Partners, adviseurs voor management en informatie in Amersfoort (www.mxi.nl)

Samenvatting ICT is zo verweven in de moderne bedrijfsvoering dat geen bestuurder het zich kan permitteren zich er niet in te verdiepen. Het wordt ze echter niet makkelijk gemaakt. Architecten verwarren besturen met simplificeren en voorzien bestuurders van abstracte informatie en vage keuzemogelijkheden. De vraag is wat bestuurders moeten met platen vol principes, logische domeinen, informatiefuncties en lagen. Wat heb je daaraan als je moet beslissen over een miljoeneninvestering in een nieuw systeem? Of als je moet uitleggen waarom weer in de krant staat dat de informatievoorziening op instorten staat?
Dan heb je meer aan daadwerkelijk inzicht in de echte informatievoorziening van een organisatie. Wat zijn de belangrijkste systemen? Hoe hangen die samen? Waar zitten de echte knelpunten, risicos of ontwikkelpunten?

1 Behoefte aan gedeelde beelden

Een goede visualisatie kan bestuurders helpen deze vragen te beantwoorden. Zo'n visualisatie ontstaat niet door een standaard (architectuur)model te gebruiken en volgens een vaste structuur in te vullen. Visualisatiemethodieken die zich daarop richten, dreigen een verkeerde afslag te nemen. Behoefte aan gedeelde beelden Beslissingstrajecten hebben baat bij een gedeeld beeld tussen de betrokken partijen – figuurlijk, maar ook letterlijk. Een visualisatie die erin slaagt dat beeld uit te drukken in de relevante concepten die alle betrokkenen begrijpen, helpt verschillen in kennis en achtergrond te overbruggen. Die relevante concepten variëren per geval. Soms gaat het om processen, soms om systemen, soms om servers. En soms om hele specifieke zaken, zoals "risico's op botsende treinen" of de "brandweer- en ambulancekolom". Dergelijke situatie-afhankelijke concepten komen bijna per definitie niet voor in standaard modelleertalen, maar zijn juist cruciaal voor de begrijpelijkheid van een visualisatie. Net zo cruciaal is dat de visuele weergave van zon concept voor zich spreekt. Als "boef" een relevant concept is, dan moet er ook een boef op de plaat staan en niet rechthoekje met het woord "boef" erin. En als er zaken wordt gedaan met leverancier Jansen, dan moet het logo van de firma Jansen worden gebruikt. Als het lukt een gedeeld beeld op te stellen, dan wordt dat beeld vaak zelf ook onderdeel van de (beeld-)taal van de organisatie. De kaart van het applicatielandschap dient ineens als achtergrond om op aan te geven waar knelpunten zitten of waar het meeste geld aan uitgegeven wordt. De behoefte aan inzicht in dergelijke dwarsverbanden ontstaat veelal spontaan en niet omdat iemand van te voren mogelijke dwarsverbanden in een modelstructuur heeft gegoten.

2 Ceci nest pas une...

Daarmee is meteen het recept voor een mislukte architectuurvisualisatie duidelijk. Werk volgens een standaard model. Druk de te visualiseren zaken uit in tevoren gedefinieerde concepten en geef die weer door vaste diagramtechnieken te gebruiken, liefst met gebruikmaking van een standaard sjabloon en formaat. Kies consistentie en theoretische correctheid

boven uitdrukkingskracht en simplificeer en abstraheer waar mogelijk. Vertrouw erop dat de details in notatie van modelleertalen breed begrepen worden. En, o ja, vul koste wat kost alle beschikbare viewpoints, views en modellen van het gekozen raamwerk in. Niemand zal zich herkennen in deze aanpak, maar toch is dit wat er vaak gebeurt. Voor het opstellen van een visualisatie wordt gekozen voor een bepaalde methodiek. Door die methodiek te volgen worden alle belanghebbenden gedwongen mee te gaan in de concepten, abstracties en verbanden daaruit. De vraag is niet langer "hoe ziet ons bedrijf eruit?", maar "uit welke bedrijfsfuncties en bedrijfsobjecten is onze dienstverlening opgebouwd?". Alsof je een grafisch ontwerper opdracht geeft een infographic over het Amerikaanse kiessysteem te maken en erbij vertelt dat hij alleen logische informatiefuncties en services mag gebruiken. Recente initiatieven om architectuurvisualisatiemethodieken te ontwikkelen volgen ditzelfde, doodlopende pad. Meer raamwerken, meta-modellen en concepten. Maar waar ging het nu ook weer om? Er is geen bestuurder (of informatiemanager of afdelingsmanager of medewerker in de uitvoering) die bij een begrijpelijke plaat vraagt waar het onderliggende meta-model is en of dat wel consequent is gevolgd. Dat doen alleen architecten. Die praten vaak ook neerbuigend over "Nijntjeplaatjes". Maar wie heeft gelijk? De architect die klaagt dat niemand zijn werk snapt? Of Dick Bruna die met slechts een paar lijnen de essentie van zijn verhaal aan een miljoenen publiek weet over te brengen?

3 De kunst van het luisteren

Want daar gaat het om bij een goede visualisatie: de essentie weten te bepalen en die zo helder mogelijk en met aandacht overbrengen. Een goede visualisatie ontstaat niet door een voorgevormde structuur in te vullen. De belangrijkste eigenschap voor degene die zon visualisatie opstelt is dan ook goed kunnen luisteren. Wat essentieel is in de belevingswereld van de verschillende belanghebbenden vormt de basis voor een visualisatie; de vormgeving wordt vervolgens zo goed mogelijk op die belevingswereld afgestemd. Aangezien het uiteindelijk gaat om visie- of besluitvorming ten aanzien van ICT is het niet ondenkbeeldig dat er (uiteindelijk) ook ICT-concepten in een visualisatie terecht komen, maar die zijn niet het uitgangspunt. Op deze manier ontstaat een basisstructuur die herkenbaar is voor de belanghebbenden. Een applicatielandschap van een gemeente bijvoorbeeld dat bestaat uit deellandschappen voor burgerzaken, de sociale dienst en parkeerbeheer. En niet (alleen) uit frontoffice, midoffice en backoffice. Of een weergave van een veiligheidsregio waarin de opdeling in meldkamer, in brandweer, ambulance en GHOR centraal staat en niet die in intake, risicobeheersing, incidentbeheersing en normaliseren. Zon basisstructuur kan en mag overigens best complex zijn — anders dan bij Dick Bruna bestaat het publiek niet uit peuters. Als het maar complexiteit is die voortkomt uit de dagelijkse praktijk van het publiek en niet uit de gekozen visualisatie.

4 Kunst, maar ook kunde

Bij het maken van een goede architectuurvisualisatie spelen — net als bij het maken van een infographic — factoren als smaak en creativiteit een belangrijke rol. Dat wil niet zeggen dat een goede visualisatie niet systematisch tot stand komt. Net als grafisch ontwerpen een echt vak is, met regels en hulpmiddelen, is het opstellen van een goede architectuurvisualisatie dat ook. Bij het maken van een plaat, of het nou de basisstructuur is of een aanvullende laag, wordt de-facto een nieuwe (domein specifieke) taal ontworpen. Gaandeweg ontstaan de regels waaraan de uiteindelijke visualisaties moeten voldoen. Dit zijn regels op lexicaal, syntactisch en semantisch niveau. Platen moeten kloppen ten opzichte van de regels. Pijlen

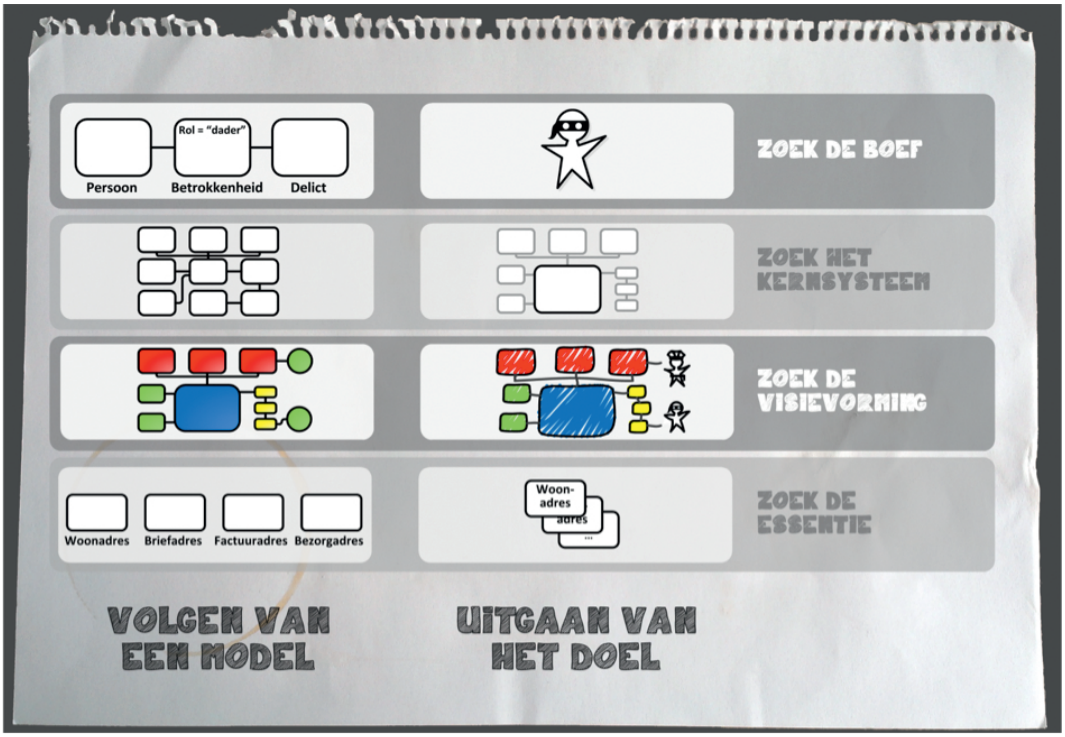
die bijvoorbeeld gegevensoverdracht of triggers representeren mogen alleen objecten van de bijpassende soort (zoals actoren en systemen) verbinden. De concepten krijgen zo veel als mogelijk hun eigen weergave. De visualisatie van vergelijkbare (maar niet dezelfde) concepten ligt dicht bij elkaar: waar een rechthoek met een dichte lijn een bestaand systeem representeert, wordt een stippellijn gebruikt voor een systeem in ontwikkeling. De regels van de taal spreken bij voorkeur voor zich door het gebruik van herkenbare symbolen voorzien van korte teksten. Waar nodig worden ze toegelicht in een kleine legenda. Het beoogd effect is dat de betrokkenen inconsistenties herkennen en zelfs nieuwe concepten aandragen. Naast regels zijn er ook heuristieken die helpen bij het ontwikkelen van visualisaties. Een belangrijke is het weergeven van de omgeving. In een goede plaat wordt de omgeving van een systeem of landschap geduid en worden de verbanden getekend.

5 Doel heiligt middelen

Hoewel de visualisatie uiteindelijk precies is, is het spelen met de regels gedurende de ontwikkeling cruciaal. Zo kan eenzelfde organisatieonderdeel meerdere keren op een plaat voorkomen, als het meerdere diensten verleent. Ook zijn er vele grafische handigheden mogelijk die in één oogopslag duidelijkheid geven, zie Figuur 1. Maar ook hier gaan architectuurmethodieken mank. Wat grafisch gezien geen enkel probleem is, is nauwelijks in de beschikbare modellen te vatten. Laat staan in tools en de visualisaties die ermee gemaakt kunnen worden. Hetzelfde geldt voor spelen met (in)consistentie. Bewuste onzorgvuldigheid en inconsistentie helpt de betrokkenen de concepten scherp te krijgen. Wat is nu precies een doelgroep? Wat zijn de kanalen waarlangs we werken? Wat zijn de relevante overdrachtsmomenten naar de productie-omgeving? Modellen en tools kunnen echter vaak heel slecht omgaan met inconsistentie.

6 Het is de weg erheen die (ook) telt

De succesvolste visualisaties zijn het resultaat van een samenwerkingsproces tussen de opsteller en diverse belanghebbenden. De waarde van het samenwerkingsproces kan niet overschat worden. Door discussies ontstaat zicht op de belangrijke concepten, de cruciale verbanden en de passende visualisaties. Tussentijdse schetsen zorgen ervoor dat iedereen het eindresultaat herkent. Dit vergt tijd. Het hergebruiken van een bestaande succesvolle visualisatie is misschien sneller, maar de kans dat de belanghebbenden hun verhaal in het eindresultaat herkennen is klein.



Figuur 1. Er zijn vele grafische mogelijkheden om een boodschap visueel te ondersteunen. Gebruik van een visuele representatie voor concepten (boef) bijvoorbeeld, of gebruik van omvang en plaatsing: groot en centraal (kernsysteem) is belangrijker dan klein en aan de rand. Een schetsmatige weergave lokt discussie uit (visievorming), een blokje met drukt uit dat er meer vergelijkbare dingen zijn, maar dat het niet uitmaakt welke precies (essentie). Dit soort mogelijkheden is in architectuurmethodieken en daarop gebaseerde tools meestal maar zeer beperkt bruikbaar.

There Is Just Information at the Bottom

Lex Augusteijn

Dear Doaitse,

Since you are a physicist, as well as a computer scientist, more specifically a computer scientist that has advocated the drawbacks of destructive state update, sincerely hope that you will witness physics being taken over by information theory within your lifetime. I believe we have all the basic insights for an information-based theory of this universe. I will list some of them.

- All basic laws of nature are time-reversible (well, you will need to swap charge and parity at the same time). This implies that information is preserved.
- From this law of information conservation, it follows that nothing can really happen in the universe, since everything can be undone.
- However, in macroscopic systems, the second law of thermodynamics does allow state change, by diluting information, not by destroying it. This is caused by the mapping of a large set of micro states onto a smaller set of macro states.
- An irreversible state change can happen to a sub-system, when information flows out of it and never returns.
- More specifically, this happens in the process of a so called measurement, which sole purpose is to obtain information from a sub-system. Since information is conserved, this implies that the sub-system will loose information.
- Until any information flows out of a system, it is therefore completely time reversible. This observation explains certain physical phenomena which are sometimes labelled as strange, beyond comprehension, spooky, etc.
- An example is the famous two slit experiment, say performed on an electron. The electron, when faced with the question which slit to choose, simply has no choice. Any choice would imply a state change, which is not allowed by the conservation law. Therefore, the electron either must be able to back-track any decision, or simply not make any. The latter is what quantum mechanics describes: the electron postpones any decision, until a measurement, i.e. when information is extracted from it, and it can finally make its choice.
- Quantum mechanics models this by the superposition principle, which just describes all possibilities and their respective probabilities, not a choice between them.
- Another example is the notion of entanglement. Here we see two sub-systems sharing information. (You would call it aliasing.) When a measurement is performed on the first sub-systems, this information is extracted and therefore, since it is aliased, also from the second sub-system. This forces an immediate state change on the second sub-system, no matter how distant that is from the first.

- The famous uncertainly principle is based on the fact that the information between the space and momentum domains, as well as the energy and time domains, is shared, simply because they are each others Fourier transforms. Therefore, extracting information from one domain will extract it from the other.
- After measuring, say, position, re-measuring it will not obtain any new information (since the information is gone from the sub-system), and therefore yield the same value. However, measuring momentum will also yield no information. Any measurement in that domain will yield a random value.
- When a position measurement did not extract all information, some information is left also in the position domain. The relationship between these amounts of information is described by the famous law of uncertainty, which states that the product of the uncertainties is at least the Planck constant.
- Therefore, we can postulate that the minimal amount of information is that Planck constant, which thus plays the physical role of a bit.
- Quantum fluctuations suggest that anything is allowed, as long as its information content does not exceed this bit.
- The quantum dynamical wave function can be seen as a (complex) probability density function. This class of functions is at the basis of information theory and entropy.
- An invariant, like the invariance of physical laws over time, generates a conservation law, in this case of energy. This raises the question which invariance would generate the conservation of information.
- Given time reversibility, any force that would drive a system irreversibly from one state to another would have to be thermodynamic in nature, in other words, be entropic. We have seen one formalization of this by Erik Verlinde, explaining gravity as such a force.
- The conservation of information has forced physicist to assume that even black holes cannot consume information. It will be stored at the event horizon (one bit per h^2) and evaporates slowly from there as Hawking radiation.

Assuming you will have an abundance of free time now, considering the fact that you were trained as a physicist, observing that you have advocated the importance of pure functions, excluding destructive state update, you might even develop such a theory yourself.

Fast, Applicative Combinator Lexers

Stefan Holdermans

Vector Fabrics
Vonderweg 22, 5616 RM Eindhoven, The Netherlands
stefan@vectorfabrics.com

Abstract. Combinator lexers form an interesting middle ground between lexer generators and handwritten lexers. They enable the rapid construction of lexical analysers without requiring knowledge of any extralinguistic formalisms. In this paper, we present a family of lexer combinators that expose their functionality as an embedded domain-specific language in Haskell. The main combinators are defined in terms of applicative functors, allowing programmers to easily familiarise themselves with the embedded language and quickly produce reasonably fast lexers written in an arguably elegant and idiomatic style.

Prologue

An essay for Doaitse should contain a fair portion of either Haskell or otherwise UUAG code. This one belongs to the former category.

My first encounter with combinator parsers was in 1999, when I took the undergraduate course on "Grammars and parsing" at Utrecht University, that year taught by Johan Jeuring. Having passed a more basic course on functional programming a couple of months before, I cannot claim that, at that point, I was very enthusiastic about functional languages. That all changed when I was confronted with the notational elegance and efficiency that combinator parsers brought to the table. That course marked the first steps that I took on a path that later led to being a PhD student of Doaitse.

While Doaitse's famous library of self-analysing parsers comes with a function that allows one to easily construct a scanner that can be used in tandem with his parser combinators, I occasionally found myself in a situation in which I wished for a proper combinator library for lexical analysis as well. When expressed, Doaitse's attitude toward such wishes is always very simple: just implement it. (And he practices what he preaches: as a byproduct of their upbringing in functional programming, several generations of freshmen have been introduced to logical programming by seeing Doaitse implement a full-fleged parser and interpreter for a subset of Prolog in the course of half a lecture of *capita selecta* in Haskell programming.) Hence, this essay is a small report on the combinator lexers that I have been using for quite a while now.

1 Introduction

Combinator parsers [1–3] provide an interesing middle ground between parsers built by hand and those that are generated automatically using tools like Yacc [4] or, for Haskell, Happy. They allow for the definition of parsers at a rather high level of abstraction, and, therefore, for parsers that are easy to build, understand, and modify. At the same time, they can be implemented as so-called embedded domain-specific languages in a sufficiently powerful host language and therefore do not require their users to first learn any additional external language.

Although not as widespread as libraries for combinator parsing, *combinator lexers* offer, in much the same way, an interesting alternative to, on the one hand, handwritten lexical analysers and, on the other hand, lexers built with external tools like Lex or Alex.

This paper presents the implementation of a simple library for combinator lexing in Haskell, that has the "look and feel" of the parsing libraries of Swierstra and others [3, 5–7].

2 Preliminaries

In this section, we briefly discuss Haskell's types of indexable arrays (Sect. 2.1), the type classes Applicative and Alternative of applicative functors (Sect. 2.2), and the type class Traversable of traversable datastructures (Sect. 2.3).

2.1 Arrays

Although lists will probably forever be the functional programmer's favourite datatype, Haskell also comes with a built-in abstract type constructor for indexable arrays [8, Ch. 16].

$$\textbf{data}\ \mathsf{Ix\ a} \Rightarrow \mathsf{Array\ a\ b} = \ \ldots \quad \text{-- abstract}$$

That is, Array a b designates the type of arrays indexed by values of type a and with elements drawn from b. Here, a is required to be an index type, i.e., a type that is a member of the class Ix, so that programmers can expect fast access to the elements of an array.

Arrays may be created by a function *array*,

$$\mathit{array} :: \mathsf{Ix\ a} \Rightarrow (\mathsf{a}, \mathsf{a}) \to [(\mathsf{a}, \mathsf{b})] \to \mathsf{Array\ a\ b},$$

that takes as its first argument a pair of bounds on the indices of the array and as its second argument a list of index/element pairs.

Access to the elements of an array is provided by the operator (!),

$$(!) :: \mathsf{Ix\ a} \Rightarrow \mathsf{Array\ a\ b} \to \mathsf{a} \to \mathsf{b},$$

which maps arrays to functions from indices to elements.

2.2 Applicative Functors

Applicative functors [9] capture a style of effectful programming that generalises from the parser combinators of Röjemo [10] and Swierstra and Duponcheel [3]. The idea is to have a function *pure* that embeds pure values into an effectful context and an operator $\circledast$ that assigns meaning to effectful function application:

$$\begin{aligned}&\textbf{infixl}\ 4\ \circledast\\ &\textbf{class}\ \mathsf{Applicative\ f}\ \textbf{where}\\ &\quad \mathit{pure} :: \mathsf{a} \to \mathsf{f\ a}\\ &\quad (\circledast)\ \ :: \mathsf{f\ (a \to b)} \to \mathsf{f\ a} \to \mathsf{f\ b}.\end{aligned}$$

As an example, consider, from the Haskell Prelude, the functor Maybe of possibly failing computations.

$$\textbf{data}\ \mathsf{Maybe\ a} = \mathit{Nothing} \mid \mathit{Just}\ \mathsf{a},$$

That is, failing computations are represented by *Nothing*, while *Just* x represents a successful computation of a value x. Hence, embedding a pure computation simply reduces to applying the constructor *Just*, while the application of two failure-prone computations fails if either one of them does:

```
instance Applicative Maybe where
  pure x              = Just x
  Just f ⊛ Just x     = Just (f x)
  _      ⊛ _          = Nothing.
```

Note that each expression constructed from *pure* and $\circledast$ can be transformed into an expression of the form

$$pure\ f \circledast xs_1 \circledast ... \circledast xs_n.$$

As a convenience, McBride and Paterson [9] propose to write such forms using so-called "idiom brackets":

$$[\![f\ xs_1\ ...\ xs_n]\!].$$

In the remainder of this paper we will adopt this notation.

Some applicative functors admit a monoidal structure with $\oplus$ an associative binary operation on effectful computations and *empty* its identity element, typically capturing a notion of choice between two effectful computations:

```
infixl 3 ⊕

class Applicative f ⇒ Alternative f where
  empty :: f a
  (⊕)   :: f a → f a → f a.
```

For example, for Maybe we have a binary operation for selecting the leftmost succeeding computation, if any:

```
instance Alternative Maybe where
  empty          = Nothing
  Nothing ⊕ xs   = xs
  xs      ⊕ _    = xs.
```

Using the methods of Applicative and Alternative, we can define combinators *some* and *many* that run a given effectful computation, respectively, one or more times and zero or more times.

```
some, many :: Alternative f ⇒ f a → f [a]
some xs    = [![ (:) xs (many xs) ]!]
many xs    = [![ [] ]!] ⊕ some xs
```

2.3 Idiomatic Traversals

Instances of the classes Applicative and Alternative are often used in combination with functorial datastructures that allow for so-called "idiomatic" traversals [9, 11]:

```
data Lexer a        = ...    -- abstract

instance Applicative Lexer where ...
instance Alternative Lexer where ...

satisfy             :: (Char → Bool) → Lexer Char
char                :: Char → Lexer Char
any, none           :: Lexer Char
anyFrom, anyBut     :: [Char] → Lexer Char
range               :: (Char, Char) → Lexer Char
string              :: String → Lexer String

scan                :: Lexer a → String → Maybe (a, String)
```

Fig. 1. The public interface to our library of lexer combinators.

```
class Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a → f (t b).
```

An example of such a datastructure is provided by the built-in type constructor for lists, for which we have

```
instance Traversable [ ] where
  traverse f [ ]       = ⟦ [ ] ⟧
  traverse f (x : xs) = ⟦ (:) (f x) (traverse f xs) ⟧.
```

3 Interface

In this section, we present the interface to our combinator library as seen by the user. This interface is summarised in Fig. 1.

The library centres on a type constructor Lexer of lexers that produce tokens of a specific type. An applicative functor, Lexer is an instance of both Applicative and Alternative. That is, lexers can be composed both sequentially (with ⊛) and in parallel (with ⓛ).

Furthermore, we have a small set of functions to produce some very basic lexers. The higher-order function *satisfy* produces a lexer that recognises all single characters that satisfy a given predicate. The function *char* yields a lexer that recognises a specific character. The lexers *any* and *none* recognise, respectively, any single character and no character at all. The functions *anyFrom* and *anyBut* include and exclude characters from a given list, whereas the function *range* only includes characters within given bounds. The function *string*, finally, recognises a specified sequence of characters.

Lexers are run by calling a function *scan* that consumes an input string and optionally produces, dependending on whether the lexer successfully recognises a prefix of the input, a pair consisting of a token and the remaining characters of the input string.

3.1 Example

As an example of how our combinator library can be used, let us now briefly discuss the implementation of a lexer for a small programming language. For this language, we need to distinguish between the following lexical elements:

– *Whitespace* consists of a sequence of one or more spaces, newline characters, and tabs.
– *Integer literals* are sequences of one or more digits.
– *String literals* are sequences of zero or more characters enclosed by double quotes. Double-quote characters within a string literal are prefixed with the escape character "\"; occurrences of the escape character itself are also prefixed by "\".
– *Identifiers* are sequences of letters, digits, and underscores, starting with a letter.
– The following are *keywords*: `fun`, `let`, and `in`.
– The following are built-in operators or special symbols: `+`, `-`, `*`, `/`, `(`, and `)`.

In Haskell, we represent these elements with values of the type Token:

```
data Token
  = Whitespace
  | IntLit Integer
  | StringLit String
  | Ident String
  | Fun | Let | In
  | Add | Sub | Mul | Div | LParen | RParen.
```

A lexer that produces values of this type can be defined as the parallel composition of a series of simpler lexers:

```
token :: Lexer Token
token = whitespace ⊕ intLit ⊕ stringLit ⊕ keyword ⊕ symbol ⊕ ident.
```

The lexer for whitespace is then defined simply by

```
whitespace :: Lexer Token
whitespace = ⟦ (const Whitespace) (some whitechar) ⟧
  where
    whitechar = char ' ' ⊕ char '\n' ⊕ char '\t'.
```

For integer literals we have

```
intLit :: Lexer Token
intLit = ⟦ (IntLit ∘ read) (some digit) ⟧
  where
    digit = range ('0', '9')
```

and for string literals, a bit more involved

```
stringLit :: Lexer Token
stringLit = ⟦ (const StringLit) (char '"') tl ⟧
  where
    tl  = ⟦ (const []) (char '"') ⟧ ⊕
          ⟦ (flip const) (char '\') esc ⟧ ⊕
          ⟦ (:) (anyBut "\"") tl ⟧
    esc = ⟦ (:) (char '"' ⊕ char '\') tl ⟧.
```

Identifiers are recognised by

```
ident :: Lexer Token
```

$$
\begin{aligned}
&\mathit{ident} = [\![\, (\lambda c\ \mathit{cs} \to \mathit{Ident}\ (c : \mathit{cs}))\ \mathit{letter}\ (\mathit{many}\ \mathit{idchar}) \,]\!] \\
&\quad \textbf{where} \\
&\qquad \mathit{idchar} = \mathit{letter} \oplus \mathit{digit} \oplus \mathit{char}\ \texttt{'\_'} \\
&\qquad \mathit{letter}\ \ = \mathit{range}\ (\texttt{'a'}, \texttt{'z'}) \oplus \mathit{range}\ (\texttt{'A'}, \texttt{'Z'}) \\
&\qquad \mathit{digit}\ \ \ = \mathit{range}\ (\texttt{'0'}, \texttt{'9'}).
\end{aligned}
$$

Finally, for keywords and built-in symbols, we have

$$
\begin{aligned}
&\mathit{keyword} :: \mathsf{Lexer\ Token} \\
&\mathit{keyword} = [\![\, (\mathit{const}\ \mathit{Fun})\ (\mathit{string}\ \texttt{"fun"}) \,]\!] \oplus \\
&\qquad\qquad [\![\, (\mathit{const}\ \mathit{Let})\ (\mathit{string}\ \texttt{"let"}) \,]\!] \oplus \\
&\qquad\qquad [\![\, (\mathit{const}\ \mathit{In})\ (\mathit{string}\ \texttt{"in"}) \,]\!]
\end{aligned}
$$

and

$$
\begin{aligned}
&\mathit{symbol} :: \mathsf{Lexer\ Token} \\
&\mathit{symbol} = [\![\, (\mathit{const}\ \mathit{Add})\ (\mathit{char}\ \texttt{'+'}) \,]\!] \oplus \\
&\qquad\qquad [\![\, (\mathit{const}\ \mathit{Sub})\ (\mathit{char}\ \texttt{'-'}) \,]\!] \oplus \\
&\qquad\qquad [\![\, (\mathit{const}\ \mathit{Mul})\ (\mathit{char}\ \texttt{'*'}) \,]\!] \oplus \\
&\qquad\qquad [\![\, (\mathit{const}\ \mathit{Div})\ (\mathit{char}\ \texttt{'/'}) \,]\!] \oplus \\
&\qquad\qquad [\![\, (\mathit{const}\ \mathit{LParen})\ (\mathit{char}\ \texttt{'('}) \,]\!] \oplus \\
&\qquad\qquad [\![\, (\mathit{const}\ \mathit{RParen})\ (\mathit{char}\ \texttt{')'}) \,]\!].
\end{aligned}
$$

Running *token* simply reduces to calling *scan token* on an input string. For example, the expression *scan token* `"237"` yields *Just* (*IntLit* 237, `""`) and *scan token* `"x + y"` yields *Just* (*Ident* `"x"`, `" + y"`).

The order in which lexers are composed matters, as the left argument to the operator $\oplus$ takes precedence over the right argument. For instance, the invocaton *scan token* `"let"` produces the token *Let* rather than *Ident* `"let"`.

Also, the lexer combinators implement the principle of "maximal munch" by consuming as much of the available input as possible. Hence, calling *scan token* with the input string `"function"` yields *Just* (*Ident* `"function"`, `""`) rather than *Just* (*Fun*, `"ction"`).

4 Applicative Lexers

In this section, we discuss the implementation of our combinator library, which can be broken down in four parts: a datatype for transition tables (Sect. 4.1), the datatype for lexers (Sect. 4.2), some functions for constructing elementary lexers (Sect. 4.3), and the function for running lexers on input strings (Sect. 4.4).

4.1 Character-indexed Vectors

Taken to their bare essence, the lexers that can be constructed with our library, are simple state machines. Moving from one state to another is triggered by the recognition of characters. To this end, lexer states come with a table that maps characters to next states. In our implementation, such a table emerges as a value of a type Vec a of character-indexed vectors, with a the type of next states.

$$
\textbf{newtype}\ \mathsf{Vec\ a} = \mathit{Vec}\ (\mathsf{Array\ Char\ a})
$$

We maintain the invariant that the underlying array of a vector holds an entry for every 8-bit character that can occur in an input string.

Character tables are constructed by means of a higher-order function *tabulate* that takes a function that maps characters to their corresponding next state.

```
tabulate    :: (Char → a) → Vec a
tabulate h = Vec arr
  where
    arr = array ('\000', '\255') [(c, h c) | c ← ['\000' .. '\255']]
```

For table lookups we have an operator # that simply delegates lookups to the underlying array.

```
(#)           :: Vec a → Char → a
Vec arr # c = arr ! c
```

The type constructor for character tables is an applicative functor.

```
instance Applicative Vec where
  pure x  = tabulate (const x)
  fs ⊛ xs = tabulate (λc → (fs # c) (xs # c))
```

Embedding a pure value reduces to constructing a table that maps each character to the given value. Application of character tables proceeds in a pointwise fashion.

If the types of the values stored in character tables are themselves constructed from a monoidal applicative functor, tables can be combined by means of an induced associative operator ⊕:

```
infixl 3 ⊕

(⊕)         :: Alternative f ⇒ Vec (f a) → Vec (f a) → Vec (f a)
xss ⊕ yss = ⟦ (⊕) xss yss ⟧.
```

This operator will prove useful in the next subsection when we instantiate the type variable f with the functor Lexer.

4.2 Lexers

The applicative functor Lexer constructs types that model the state of a lexer for recognising tokens of a specific type.

```
data Lexer a = Fail | Ok (Maybe a) (Vec (Lexer a))
```

Lexer states constructed by *Fail* indicate that recognition of a token has failed, whereas states built with *Ok* may still result in the successful recognition of a token. An *Ok*-state comes with an optional token and a transition table for recognising the next character. The optional token is the token that is produced if further consumption of the input string does not yield a token.

Implementation of the Applicative-methods *pure* and ⊛ is relatively straightforward.

```
instance Applicative Lexer where
  pure x                  = Ok (pure x) (pure Fail)
  Fail          ⊛ _       = Fail
  Ok Nothing fss ⊛ l      = Ok empty ⟦ (⊛l) fss ⟧
```

$$
\begin{aligned}
&Ok\ (Just\ \_)\ fss \circledast l@Fail &&= Ok\ empty\ [\![\ (\circledast l)\ fss\ ]\!] \\
&Ok\ (Just\ f)\ fss \circledast l@(Ok\ xs\ xss) &&= Ok\ [\![\ f\ xs\ ]\!]\ ([\![\ (\circledast l)\ fss\ ]\!] \oplus [\![\ (pure\ f \circledast)\ xss\ ]\!])
\end{aligned}
$$

A lexer built from a pure value is a lexer that immediately, i.e., without consuming any input, produces that value as a token and that is furthermore incapable of recognising any characters from the input string. If the first of two sequentially composed lexers immediately fails, the combined lexer too fails immediately. If the first lexer does not fail immediately, the behaviour of the combined lexer depends on whether the first lexer can immediately produce a token already.

If it cannot produce a token, lexing necessarily proceeds by first having the first lexer recognising characters until it either fails or succesfully yields a token.

If the first lexer can produce a token, we need to inspect the state of the second lexer. If the second lexer fails, we surely cannot produce a token, but rather than aborting immediately, we have the first lexer consume from the input string as much as it can to reflect that lexing only fails when the second lexer comes in play. If the second lexer is in an *Ok*-state, the combined lexer can either optionally produce a token or otherwise consume the next character from the input string and move to a state that is obtained from combining the character tables of the composing lexers.

Parallel composition of lexers merges nonfailing lexers in a pointwise fashion:

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Alternative\ Lexer}\ \textbf{where} \\
&\quad empty &&= Ok\ empty\ (pure\ empty) \\
&\quad Fail \oplus l &&= l \\
&\quad l \oplus Fail &&= l \\
&\quad Ok\ xs\ xss \oplus Ok\ ys\ yss &&= Ok\ (xs \oplus ys)\ (xss \oplus yss).
\end{aligned}
$$

The lexer produced by *empty* consumes characters without ever producing a token.

4.3 Basic Lexers

Lexers that acutally consume input are constructed by means of the function *satisfy* that takes as its argument a predicate on characters.

$$
\begin{aligned}
&satisfy &&:: (\mathsf{Char} \to \mathsf{Bool}) \to \mathsf{Lexer\ Char} \\
&satisfy\ p &&= Ok\ empty\ (tabulate\ h) \\
&\quad \textbf{where} \\
&\qquad h\ c &&= \textbf{if}\ p\ c\ \textbf{then}\ pure\ c\ \textbf{else}\ Fail
\end{aligned}
$$

The function then produces an *Ok*-lexer that maps all characters that satisfy the predicate to lexers that produces those characters as tokens; all other characters are mapped to *Fail*.

The remaining functions for producing lexers that recognise single characters can be expressed in terms of *satisfy*:

$$
\begin{aligned}
&char &&:: \mathsf{Char} \to \mathsf{Lexer\ Char} \\
&char\ c &&= satisfy\ (\equiv c) \\
&any, none &&:: \mathsf{Lexer\ Char} \\
&any &&= satisfy\ (const\ True) \\
&none &&= satisfy\ (const\ False) \\
&anyFrom, anyBut &&:: [\mathsf{Char}] \to \mathsf{Lexer\ Char} \\
&anyFrom\ cs &&= satisfy\ (\in cs)
\end{aligned}
$$

$$
\begin{array}{ll}
\mathit{anyBut}\ cs & = \mathit{satisfy}\ (\notin cs) \\
\mathit{range} & :: (\mathsf{Char}, \mathsf{Char}) \to \mathsf{Lexer\ Char} \\
\mathit{range}\ (\mathit{low}, \mathit{high}) & = \mathit{satisfy}\ (\lambda c \to c \geqslant \mathit{low} \wedge c \leqslant \mathit{high}).
\end{array}
$$

The function *string* for recognising a specific sequence of characters can be implemented as an idiomatic traversal of the character sequence:

$$
\begin{array}{l}
\mathit{string} :: \mathsf{String} \to \mathsf{Lexer\ String} \\
\mathit{string} = \mathit{traverse}\ \mathit{char}.
\end{array}
$$

4.4 Scanning

The actual recognition of input characters constituting tokens is implemented by the function *scan*, which takes a lexer and an input string and returns either *Nothing* if no token can be produced or otherwise a value $\mathit{Just}\ (x, s)$ with x a token and s the remainder of the input string.

$$
\begin{array}{l}
\mathit{scan} :: \mathsf{Lexer\ a} \to \mathsf{String} \to \mathsf{Maybe\ (a, String)} \\
\mathit{scan} = \mathit{go}\ \mathit{Nothing} \\
\quad \textbf{where} \\
\qquad \begin{array}{ll}
\mathit{go}\ \mathit{acc}\ \mathit{Fail}\ \_ & = \mathit{acc} \\
\mathit{go}\ \mathit{acc}\ (\mathit{Ok}\ \mathit{xs}\ \_)\ [\,] & = \mathit{munch}\ \mathit{acc}\ \mathit{xs}\ [\,] \\
\mathit{go}\ \mathit{acc}\ (\mathit{Ok}\ \mathit{xs}\ \mathit{xss})\ s@(c : \mathit{cs}) & = \mathit{go}\ (\mathit{munch}\ \mathit{acc}\ \mathit{xs}\ s)\ (\mathit{xss} \mathbin{\#} c)\ \mathit{cs} \\
\mathit{munch}\ \mathit{acc}\ \mathit{xs}\ s & = [\![\, (\mathit{flip}\ (,)\ s)\ \mathit{xs} \,]\!] \oplus \mathit{acc}
\end{array}
\end{array}
$$

Scanning proceeds by recursively processing the input string while updating the lexer state and accumulating a return value, which is initially set to *Nothing*.

If, during our scan of a prefix of the input string, we end up in a failing lexing state, scanning is aborted and the so far accumulated value is returned.

Otherwise, if we reach the end of the input string without running into a failure, the optional token produced by the final lexer state is combined with the so far accumulated return value. This combination is implemented by an auxiliary function *munch* and follows the "maximal munch" principle by giving preference to tokens that are produced later during the consumption of the input stream. (Recall that the operator $\oplus$ for computations in Maybe selects the leftmost *Just*-constructed value.)

Finally, for *Ok*-lexers that have not reached the end of the input yet, we update the accumulated value and continue scanning with the lexer that is obtained through a table lookup for the next input character.

5 Conclusion

The only other lexer-combinator library in Haskell that we are aware of is that of Chakravarty [12], who models his interface around the syntax of regular expressions rather than applicative idioms.

Experiments show that the combinators described in this paper are reasonably fast. Detailed benchmarks are left as future work.

The implementation of the actual library is somewhat more advanced than we have described here. In particular, the library provides support for keeping track of line and column numbers, and comes with facilities for error reporting.

References

1. Hutton, G.: Higher-order functions for parsing. Journal of Functional Programming **2**(3) (1992) 323–343
2. Fokker, J.: Functional parsers. In Jeuring, J., Meijer, E., eds.: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24–30, 1995, Tutorial Text. Volume 925 of Lecture Notes in Computer Science., Springer (1995) 1–23
3. Swierstra, S.D., Duponcheel, L.: Deterministic, error-correcting combinator parsers. In Launchbury, J., Meijer, E., Sheard, T., eds.: Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26–30, 1996, Tutorial Text. Volume 1129 of Lecture Notes in Computer Science., Springer (1996) 184–207
4. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools. 2nd edn. Pearson Education, Boston, Massachusetts (2006)
5. Swierstra, S.D.: Combinator parsers: From toys to tools. Electronic Notes in Theoretical Computer Science **41**(1) (2001) 38–59
6. Hughes, J., Swierstra, S.D.: Polish parsers, step by step. In Runciman, C., Shivers, O., eds.: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25–29, 2003, ACM Press (2003) 239–248
7. Baars, A.I., Löh, A., Swierstra, S.D.: Parsing permutation phrases. Journal of Functional Programming **14**(6) (2004) 635–646
8. Peyton Jones, S., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)
9. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming **18**(1) (2008) 1–13
10. Röjemo, N.: Garbage Colllection and Memory Efficiency in Lazy Functional Languages. PhD thesis, Chalmers University of Technology and Göteborg Universit (1995)
11. Gibbons, J., Oliveira, B.C.d.S.: The essence of the Iterator pattern. Journal of Functional Programming **19**(3–4) (2009) 377–402
12. Chakravarty, M.M.T.: Lazy lexing is fast. In Middelkoop, A., Sato, T., eds.: Functional and Logic Programming, 4th Fuji International Symposium, FLOPS'99, Tsukuba, Japan, November 11–13, 1999, Proceedings. Volume 1722 of Lecture Notes in Computer Science., Springer (1999) 68–84

AMEN

Wouter Swierstra

w.s.swierstra@uu.nl
Department of Information and Computing Sciences
Universiteit Utrecht

Abstract. One of Doaitse's regular contributions to teaching at the Universiteit Utrecht, has been his lectures for the Compiler Construction course about Church encodings and combinatory logic. I think one of the reasons he enjoys these topics so much is their combination of expressive power and simplicity. It is quite surprising just how much you can achieve in a 'language' as simple as the lambda calculus. This paper covers much of the same material, but adds a new twist.

Introduction

Church encodings represent data as functions. A classic example, written here using Haskell, is that of the natural numbers:

$$\textbf{type } \mathit{Nat} = \forall a.(a \to a) \to (a \to a)$$

Every natural number n is represented by a higher-order function that applies its argument function n times. For example, we can define the Church encoding of the first three natural numbers as follows:

```
naught :: Nat
naught = λf x → x
one :: Nat
one    = λf x → f x
two :: Nat
two    = λf x → f (f x)
```

Furthermore you can define combinators to add, multiply, or exponentiate the Church encodings of two natural numbers. Here are possible definitions for these operations:

```
add :: Nat → Nat → Nat
add m n = λf x → n f (m f x)
mul :: Nat → Nat → Nat
mul m n = λf → n (m f)
exp :: Nat → Nat → Nat
exp m n  = n m
```

When encountered for the first time, these results are quite surprising: lambda abstractions and application are enough to do basic arithmetic.

Church encodings simplify the notion of data, but higher-order functions are still complicated beasts: capture-avoiding substitution, α-equivalence, β-reduction, and all the usual issues associated with variable binding can be a real headache.

One way to tame this complexity is by implementing the lambda calculus using a combinator calculus. Such a calculus consists of a (small) set of 'elementary functions ... which embody certain common patterns of application' (Burge, 1975, Section 1.9). The most famous choice is the three combinators S, K, and I given by the following lambda terms:

$$\begin{aligned}\mathsf{S} &= \lambda x\ y\ z \to x\ z\ (y\ z)\\ \mathsf{K} &= \lambda x\ y \to x\\ \mathsf{I} &= \lambda x \to x\end{aligned}$$

This choice of combinators is particularly popular because of the straightforward translation scheme from lambda terms to their combinator counterparts. As is common in the literature (Barendregt, 1981; Sørensen and Urzyczyn, 2006), we write $\lambda^* x \to t$ for the translation of the term $\lambda x \to t$ to its corresponding combinator term. The following three rules define one possible translation scheme:

$$\begin{aligned}(\lambda^* x \to x) &= \mathsf{I}\\ (\lambda^* x \to A) &= \mathsf{K}\ A \text{ provided } x \notin A\\ (\lambda^* x \to t_1\ t_2) &= \mathsf{S}\ (\lambda^* x \to t_1)\ (\lambda^* x \to t_2)\end{aligned}$$

A choice of combinators for which such a translation exists is said to be *combinatory complete* (Barendregt, 1981).

There are many alternative combinatory complete bases: Curry and Feys (1958) originally proposed the following combinators:

$$\begin{aligned}\mathsf{I} &= \lambda x \to x\\ \mathsf{C} &= \lambda f\ x\ y \to f\ y\ x\\ \mathsf{W} &= \lambda f\ x \to f\ x\ x\\ \mathsf{B} &= \lambda f\ g\ x \to f\ (g\ x)\\ \mathsf{K} &= \lambda x\ y \to x\end{aligned}$$

Fokker (1992) has even derived a single combinator that is still combinatory complete:

$$\mathsf{X} = \lambda f \to f\ \mathsf{S}\ (\lambda x\ y\ z \to x)$$

As Doaitse is fond of observing, this shows how a term in the lambda calculus can be reduced to a series of opening and closing brackets (placed in some series of X combinators of a certain length), which in turn can be encoded as a series of bits.

So far we have seen how to represent numbers by functions and functions by combinators. Now you might wonder if it is also possible to represent combinators by numbers. Or more precisely, can we use the Church encoding of natural numbers to define a combinatory complete basis?

Proposition 1. *The combinators A, M, E, and N, defined in Figure 1, correspond to the lambda terms for respectively addition, multiplication, exponentiation and naught for the Church encoding of Peano arithmetic. These combinators form a combinatory complete basis.*

I learned this proposition from Peter Hancock during my PhD in Nottingham. Peter tells me that the choice of combinator names is due to Jim Laird. This is no new result. Peter tells me that this proposition dates back to Stenlund (1972). Independently Böhm (1979) also describes the combinatory completeness of these combinators. Given the occasion, and Doaitse's love for combinatory calculi, it seems appropriate to reproduce this result here.

$$
\begin{aligned}
\mathsf{A} &= \lambda m\ n\ f\ x \to n\ f\ (m\ f\ x)\\
\mathsf{M} &= \lambda m\ n\ f \to n\ (m\ f)\\
\mathsf{E} &= \lambda m\ n \to n\ m\\
\mathsf{N} &= \lambda x\ y \to y
\end{aligned}
$$

Fig. 1. The AMEN combinators

To prove this proposition, we give the following implementations of I, C, W, B, K, and S using the AMEN combinators:

```
I = N M
C = M M (M E)
W = C (E E (A E))
B = M E (M M)
K = M E (M N)
S = M (A E) C
```

The proof that these definitions behave as required can be found in the appendix. As you might expect, these calculations are rather dull. I do not want to claim that these are the shortest or prettiest definitions of the desired combinators. They just happen to be the first definitions that a program happened to find automatically. It's much more fun to have a closer look at this program, than the proofs it generates.

Evaluation

To find the above definitions, I wrote a short Haskell program. At its heart is an evaluation function, which tries to evaluate a term using the AMEN combinators to a normal form, if it exists.

Terms and values

To start, let us define data types to represent combinatory terms:

```
data Term a =
  Const a
  | (Term a) :@: (Term a)
```

Here we represent combinatory terms as binary trees of applications, with primitive combinators in the leaves. By defining this type to be polymorphic, we keep the option open to use the same structure for different choices of primitive combinators. One such choice of primitive combinators is, of course, the AMEN combinators we described above:

```
data AMEN = A | M | E | N
```

We introduce a separate data type to represent *values*, those terms that cannot be reduced further:

```
data Val a where
  Val :: a → [Val a] → Val a
```

One invariant that our data type does not express is that a value should not be applied to enough arguments to reduce. For example, $Val\ \mathsf{A}\ [\,]$ is really a value; $Val\ \mathsf{A}\ [m, n, f, x]$ on the other hand, is not a valid value for any choice of values m, n, f, and x as the combinator A has enough arguments to trigger reduction.

Decomposition

In the following pages, we will define a small step evaluation function in the style of Danvy (2008). We start by defining a data type for *evaluation contexts* (Felleisen and Hieb, 1992):

```
data Context a where
  Empty :: Context a
  Left  :: Context a → Term a → Context a
  Right :: Val a → Context a → Context a
```

You may want to think of such contexts as a zipper (Huet, 1997) in a *Term*, with the special property that we may only navigate to the right if we have fully evaluated the term on the left.

Just as regular zippers, such evaluation contexts support a *plug* operation, that plugs a term back into an evaluation context:

```
plug :: Term a → Context a → Term a
plug t Empty          = t
plug t (Left ctx u)   = (plug t ctx) :@: u
plug t (Right v ctx) = (fromVal v) :@: (plug t ctx)
```

Note that we use an auxiliary function *fromVal* converting a *Val* to a *Term* in the obvious fashion.

Although we have defined a data type to represent evaluation contexts, we have not yet written a function that *produces* an evaluation context. To do so, we make the observation that every term is either a value or a redex in some evaluation context. In line with Danvy, we introduce the data type *Decomposition* to express this choice.

```
type Redex a = (Val a, Val a)
data Decomposition a where
  IsVal   :: Val a → Decomposition a
  IsRedex :: Redex a → Context a → Decomposition a
```

The *Decomposition* data type can be thought of as describing a *view* (Wadler, 1987; McBride and McKinna, 2004) on terms. A redex consists of a value, together with a new argument for this value that may trigger further reduction.

We can construct a decomposition of any term using a pair of mutually recursive functions, *load* and *unload*:

```
decompose :: Term a → Decomposition a
decompose t = load t Empty

load :: Term a → Context a → Decomposition a
load (Const x) ctx = unload (Val x [ ]) ctx
```

```
load (f :@: x)   ctx = load f (Left ctx x)
unload :: Val a → Context a → Decomposition a
unload v Empty          = IsVal v
unload v (Left xs t)    = load t (Right v xs)
unload v (Right f xs) = IsRedex (f, v) xs
```

Starting with the empty context, the *load* function navigates to the leftmost innermost combinator or variable. Once found, it calls the *unload* function, defined by induction over the evaluation context. If the evaluation context is empty, the term we are decomposing is indeed a value. If the evaluation context is not empty, there are two possible cases. If this value has been applied to some argument, in which case there is a *Left* constructor on top of the evaluation context, we proceed by storing the value on the 'stack' and decomposing the term stored in the evaluation context in search for a redex. If the value we found was the argument to another value, witnessed by a *Right* constructor on top of the evaluation context, we know that we have found a redex.

Contraction

We will use the *decompose* function to define a small step evaluator for a combinator language. Such an evaluator will start by decomposing its argument term. If this yields a value, evaluation is finished; if this yields a redex, we contract the redex and continue reduction. Before completing our evaluator, we still need to define how to contract a redex. Contraction, however, is specific to the combinatory basis we choose. As we try to defer this choice for as long as possible, we introduce a separate *Contractible* class:

```
class Contractible a where
  step :: a → [Val b] → Maybe (Term b)
```

The *step* function has a slightly more general type than you might expect. Given a combinator of type a, applied to a list of arguments of type *Val* b, it may choose to rearrange these arguments and produce a new term of type *Term* b. If no reduction is possible, it should return *Nothing*. The additional flexibility of separating the type of the combinator (a) from the type of the values it manipulates (*Val* b) will turn out to be useful.

We specify how the *AMEN* combinators reduce by defining a suitable instance of the *Contractible* class:

```
instance Contractible AMEN where
  step A [v1, v2, v3, v4] = return (n :@: f :@: (m :@: f :@: x))
    where
      [m, n, f, x] = map fromVal [v1, v2, v3, v4]
  step M [v1, v2, v3]     = return (n :@: (m :@: f))
    where
      [m, n, f] = map fromVal [v1, v2, v3]
  step E [v1, v2]         = return (n :@: m)
    where
      [m, n] = map fromVal [v1, v2]
  step N [v1, v2]         = return x
    where
      [f, x] = map fromVal [v1, v2]
  step _ _                = Nothing
```

If you squint a bit, you should be able to recognize the original definition of the AMEN combinators from Figure 1. We can use this *step* function to (try to) contract a redex:

```
contract :: (Contractible b) ⇒ Val b → Val b → Maybe (Term b)
contract (Val x args) arg = step x (args ++ [arg])
```

To contract a redex, we add the new argument to the current value. Calling the *step* function then produces a new term, if the redex can now reduce.

Evaluation

```
eval :: Contractible a ⇒ Term a → Val a
eval t = go (decompose t)
  where
  go :: Contractible a ⇒ Decomposition a → Val a
  go (IsVal v)              = v
  go (IsRedex (f, v) ctx) =
    case (f `contract` v) of
      Just t'   → go (load t' ctx)
      Nothing → go (unload (addArg f v) ctx)
addArg :: Val a → Val a → Val a
addArg (Val x args) arg = Val x (args ++ [arg])
```

Fig. 2. The *eval* function

We can now define *eval* as a tail-recursive function in Figure 2. The *eval* function decomposes its argument term. If this yields a value, evaluation is complete. If we find a redex, we can try to contract it. If the contraction is successful and reduction takes place, continue evaluation with the new term and the current evaluation context. If the contraction did not (yet) reduce, we add the new argument to our stuck value and continue evaluation, in the hope that this will eventually yield further arguments for this stuck value.

It is easy to adapt this evaluator to produce an evaluation trace, printing every intermediate reduction step during execution. By doing so, and adding some extra lhs2TEX pragmas, we are able to produce the proofs in the appendix. The question remains, however, how we found the terms for I, C, W, B, K and S.

Searching for combinators

To find an AMEN term automatically that implements these combinators, we will specify their behaviour and use SmallCheck (Runciman et al., 2008) to search for a term that exhibits this intended behaviour.

To specify the desired behaviour of a combinator, we start by extending our *Term* language with variables. To do so, we require the following two definitions:

```
data Var = Var String
```

```
infixr 6 :+:
data (a :+: b) = Inl a | Inr b
class (:<:) sub sup
  where inj :: sub → sup
instance (:<:) a a
  where inj = id
instance (:<:) a (a :+: b)
  where inj = Inl
instance ((:<:) a c) ⇒ (:<:) a (b :+: c)
  where inj = Inr . inj
instance (Contractible a, Contractible b) ⇒ Contractible (a :+: b) where
  step (Inl x) args = step x args
  step (Inr y) args = step y args
```

Fig. 3. Automated injections

```
instance Contractible Var where
  step (Var x) args = Nothing
```

The *Contractible* instance for variables merely states that variables never reduce.

To grow our language, we take the coproduct of *Var* and *AMEN* as the possible values stored in the leaves of *Term* data type. Using the technology from Swierstra (2008), summarized here in Figure 3 we can write a smart constructor to create variables.

```
var :: (Var :<: a) ⇒ String → Term a
var x = Const (inj (Var x))
```

Now finally, we can state the property that a term behaves precisely as the combinator K should:

```
isK :: Term (AMEN :+: Var) → Bool
isK t = eval (t :@: var "x" :@: var "y") ≡ Val (inj (Var "x")) []
```

Now we can call SmallCheck and ask it to exhaustively search up to a certain depth, trying to find a term t that satisfies this property. We could formulate the test that this property holds as:

```
exists isK
```

Unfortunately, SmallCheck does not report the witness that satisfies such a property. Instead we negate the statement and search for a counterexample as follows:

```
*Main> smallCheck 5 (forAll (not . isK))
Failed test no. 118.
there exists M E (M N) such that
  condition is false
```

To run this test, we need to write a few lines of boilerplate to help SmallCheck generate terms. This is entirely standard using the combinators provided by the SmallCheck library. The only interesting case is that for variables. As we only want SmallCheck to generate closed terms using combinators, the generator for variables always fails.

In a similar fashion, we can find the definitions of I, C, and B. All these searches return a result almost immediately. The other combinators are harder to find. While we could increase the depth of our search, this does not help much: the search space quickly becomes too large.

Instead, we exploit the fact that we know how to implement C using our original AMEN combinators. It is therefore safe to add it as a primitive combinator to our combinator language:

```
data C = C
instance Contractible C where
  step C [v1, v2, v3] = return (f :@: y :@: x)
    where
      [f, x, y] = map fromVal [v1, v2, v3]
  step C _ = Nothing
```

To permit the usage of C in our search, all we need to do is change the *type signature* of our predicate over terms. For instance, the predicate characterizing the W combinator becomes:

```
isW :: Term (AMEN :+: Var :+: C) → Bool
isW t = eval (t :@: (var "f") :@: (var "x"))
  ≡ Val (inj (Var "f")) [Val (inj (Var "x")) [], Val (inj (Var "x")) []]
```

With this modification, SmallCheck finds the terms for all the six combinators on page 319 in less than thirty seconds.

Further fun

No paper for Doaitse would be complete without some mention of parsing. When writing this paper, I was never really happy with some of the SmallCheck properties. They quickly devolve into large values and terms, full of parentheses, string literals, and lists—the *isW* property above is a good example. Surely we can do better! Doaitse is certainly not one to shy away from experimental GHC extensions, which we will use for this last diversion.

Let's start by defining a parser for values using Doaitse's `uu-parsinglib`. As values can contain arbitrary combinators, we define a type class to represent types that have some associated parser:

```
type Parser a = P (Str Char String LineColPos) a
class Parsable a where
  parser :: Parser a
```

Using this type class, we can parse any value containing *Parsable* combinators.

```
pVal :: Parsable a ⇒ Parser (Val a)
pVal = foldl1 addArg <$> pList1Sep pSpaces pValAtom
  where
  pValAtom = (λx → Val x []) <$> parser
           <|> pParens pVal
```

This parser recognizes a series of atomic values, separated by whitespace. An atomic value is either a primitive combinator, variable, or a compound value surrounded by parentheses. For example, to parse various combinators or variables we define straightforward parsers.

```
instance Parsable AMEN where
  parser = A <$ pSym 'A'
         <|> M <$ pSym 'M'
         <|> E <$ pSym 'E'
         <|> N <$ pSym 'N'
instance Parsable C where
  parser = C <$ pSym 'C'
instance Parsable Var where
  parser = (λx xs → Var (x : xs)) <$> pLower <*> pMany pLetter
```

We can also define a parser for sum of parsible types:

```
instance (Parsable a, Parsable b) ⇒ Parsable (a :+: b) where
  parser = Inl <$> parser
         <|> Inr <$> parser
```

Of course, just defining these parsers is not very useful just yet, as we have not yet defined how to run them.

Using GHC's recent extension for quasiquoting (Mainland, 2007), however, we can add special syntax for values. To do so, we need to define a record of type *QuasiQuoter*. As we will only use this quasiquoter to quote expressions, as opposed to types, patterns, and declarations, we only fill in one field, *quoteExp*. The *quoteExp* field requires a function of type *String* → *Q Exp* which parses any string to a Template Haskell abstract syntax tree. By running our value parser, and quoting the result using the Template Haskell *lift* function, we can assemble the desired quasiquoter:

```
val :: QuasiQuoter
val = let p = pVal :: Parser (Val (AMEN :+: Var :+: C))
      in QuasiQuoter { quoteExp = lift . run p }
```

The *run* function discards any leading or trailing whitespace and runs its argument parser on the input string. The Template Haskell *lift* function turns a Haskell value into the corresponding abstract syntax tree. The complete code, once again, requires a few trivial instances of the Template Haskell *Lift* class. These instances are easy to write by hand.

Using this QuasiQuoter, makes it a bit easier to write complicated SmallCheck properties using non-trivial values.[1] For example, to specify the behaviour of the S combinator, we can now write:

```
isS :: Term (AMEN :+: Var :+: C) → Bool
isS t = eval (t :@: var "x" :@: var "y" :@: var "z")
      ≡ [val | x z (y z) |]
```

Of course, we can define a quasiquoter for our *Term* type as well. To be useful, however, we would need to extend our language with *anti-quotation*, that is, the ability to refer to the Haskell variable *t* in such a quoted expression. But perhaps that is a story for another time.

[1] To use such quasi-quotation, this code needs to be in a separate file due to GHC's Template Haskell stage restriction.

Closure

Thank you Doaitse, for all your work. Your uninhibited enthusiasm for functional programming has certainly proved to be contagious. I fondly remember your hospitality when I stayed in Houten during my first week at university as a young student. Whenever I would stop by your office, you would always recommend an interesting paper to read. At the end of my third year in Utrecht, you suggested I visit Oege de Moor to write my BSc thesis. This turned out to be my first encounter with dependent types. At the time, I could never have imagined teaching a seminar in Utrecht on the topic ten years later.

You will be missed. AMEN.

References

Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics, Studies in Logic and the Foundations of Mathematics, vol. 103. Elsevier (1981)

Böhm, C.: Un modèle arithmétique des termes de la logique combinatoire. In: Robinet, B. (ed.) Proc. Sixième Ecole de Printemps d'Informatique Theorique – Lambda Calcul et Semantique Formelle des Langages de Programmation. pp. 97–108 (1979)

Burge, W.: Recursive Programming Techniques. Addison-Wesley Publishing Company (1975)

Curry, H., Feys, R.: Combinatory Logic, vol. 1. North-Holland Publishing Company (1958)

Danvy, O.: From reduction-based to reduction-free normalization. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) Proceedings of the 6th International School on Advanced Functional Programming. pp. 66–164. No. 5382 in LNCS, Springer-Verlag (2008)

Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoretical computer science 103(2), 235–271 (1992)

Fokker, J.: The systematic construction of a one-combinator basis for lambda-terms. Formal Aspects of Computing 4, 776–780 (1992)

Huet, G.: The zipper. Journal of Functional Programming 7(5), 549–554 (1997)

Mainland, G.: Why it's nice to be quoted: Quasiquoting for Haskell. In: Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop. pp. 73–82. ACM, New York, NY, USA (2007)

McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming 14(1) (2004)

Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In: Proceedings of the first ACM SIGPLAN symposium on Haskell. pp. 37–48. Haskell '08 (2008)

Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism, Studies in Logic and the Foundations of Mathematics, vol. 149. Elsevier Science Inc. (2006)

Stenlund, S.: Combinators, λ-Terms and Proof Theory, Synthese Library, vol. 42. Reidel, Dordrecht (1972)

Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423–436 (2008)

Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 307–313 (1987)

A Proofs

A.1 Correctness of the definition of I

$$
\begin{array}{ll}
& \mathsf{I}\ x \\
= & \quad \{\text{by definition of } \mathsf{I}\} \\
& \mathsf{N}\ \mathsf{M}\ x \\
= & \quad \{\text{by definition of } \mathsf{N}\} \\
& x
\end{array}
$$

A.2 Correctness of the definition of C

$$
\begin{array}{ll}
& \mathsf{C}\ f\ x\ y \\
= & \quad \{\text{by definition of } \mathsf{C}\} \\
& \mathsf{M}\ \mathsf{M}\ (\mathsf{M}\ \mathsf{E})\ f\ x\ y \\
= & \quad \{\text{by definition of } \mathsf{M}\} \\
& \mathsf{M}\ \mathsf{E}\ (\mathsf{M}\ f)\ y\ x \\
= & \quad \{\text{by definition of } \mathsf{M}\} \\
& \mathsf{M}\ f\ (\mathsf{E}\ x)\ y \\
= & \quad \{\text{by definition of } \mathsf{M}\} \\
& \mathsf{E}\ x\ (f\ y) \\
= & \quad \{\text{by definition of } \mathsf{E}\} \\
& f\ y\ x
\end{array}
$$

A.3 Correctness of the definition of *W*

$$
\begin{array}{ll}
& W\ f\ x \\
= & \quad \{\text{by definition of } W\} \\
& \mathsf{C}\ (\mathsf{E}\ \mathsf{E}\ (\mathsf{A}\ \mathsf{E}))\ f\ x \\
= & \quad \{\text{by definition of } \mathsf{E}\} \\
& \mathsf{C}\ (\mathsf{A}\ \mathsf{E}\ \mathsf{E}\ x\ f) \\
= & \quad \{\text{by definition of } \mathsf{C}\} \\
& \mathsf{A}\ \mathsf{E}\ \mathsf{E}\ x\ f \\
= & \quad \{\text{by definition of } \mathsf{A}\} \\
& \mathsf{E}\ x\ (\mathsf{E}\ x\ f) \\
= & \quad \{\text{by definition of } \mathsf{E}\} \\
& \mathsf{E}\ x\ (f\ x) \\
= & \quad \{\text{by definition of } \mathsf{E}\} \\
& f\ x\ x
\end{array}
$$

A.4 Correctness of the definition of *B*

$$
\begin{array}{ll}
& B\ f\ g\ x \\
= & \quad \{\text{by definition of } B\}
\end{array}
$$

$\mathsf{M}\ \mathsf{E}\ (\mathsf{M}\ \mathsf{M})\ f\ g\ x$
$=$ {by definition of M}
$\mathsf{M}\ \mathsf{M}\ (\mathsf{E}\ f)\ x\ g$
$=$ {by definition of M}
$\mathsf{E}\ f\ (\mathsf{M}\ g)\ x$
$=$ {by definition of E}
$\mathsf{M}\ g\ f\ x$
$=$ {by definition of M}
$f\ (g\ x)$

A.5 Correctness of the definition of K

$\mathsf{K}\ x\ y$
$=$ {by definition of K}
$\mathsf{M}\ \mathsf{E}\ (\mathsf{M}\ \mathsf{N})\ x\ y$
$=$ {by definition of M}
$\mathsf{M}\ \mathsf{N}\ (\mathsf{E}\ x)\ y$
$=$ {by definition of M}
$\mathsf{E}\ x\ (\mathsf{N}\ y)$
$=$ {by definition of E}
$\mathsf{N}\ y\ x$
$=$ {by definition of N}
x

A.6 Correctness of the definition of *S*

$S\ x\ y\ z$
$=$ {by definition of S}
$\mathsf{M}\ (\mathsf{A}\ \mathsf{E})\ \mathsf{C}\ x\ y\ z$
$=$ {by definition of M}
$\mathsf{C}\ (\mathsf{A}\ \mathsf{E}\ x)\ z\ y$
$=$ {by definition of C}
$\mathsf{A}\ \mathsf{E}\ x\ z\ y$
$=$ {by definition of A}
$x\ z\ (\mathsf{E}\ z\ y)$
$=$ {by definition of E}
$x\ z\ (y\ z)$