

A PROGRAMMING ROAD TO LOGIC, MATHS, LANGUAGE, AND PHILOSOPHY



*A Tribute to Jan van Eijck
on the Occasion of His Retirement*

A Programming Road to Logic, Maths, Language, and Philosophy

*A Tribute to Jan van Eijck on the Occasion of
His Retirement*

Table of Contents

Preface.....	7
<i>Krzysztof Apt</i>	
For Jan: About the Pythagorean Theorem	9
<i>Matteo Capelletti</i>	
My Story with Jan, in Haskell	12
<i>Kees Doets</i>	
Unavoidability of Induction	21
<i>Malvin Gattinger</i>	
From Zero to Logic in Haskell	26
<i>Fengkui Ju</i>	
Normative Notions by Colored Actions	30
<i>Paul Klint</i>	
An Exercise in Exercises	36
<i>Barteld Kooi and Rineke Verbrugge</i>	
From Principles to Practical Pigeon Protocols	49
<i>Bert Lisser</i>	
Alain Badiou plays the game of Set	54
<i>Ştefan Minică</i>	
Interactive Reasoning	55
<i>Reinhard Muskens</i>	
16T^AP : A Toy Tableaux Theorem Prover for 16-Valued Trilattice Logics	57
<i>Rick Nouwen</i>	
Nothing to Add	78
<i>Rohit Parikh</i>	
A Tribute to Jan van Eijck	87
<i>Marc Pauly</i>	
Programming Real Social Software: Matching Students to Supervisors using Perl	96
<i>R. Ramanujam</i>	
A Conversation on Money	104
<i>Hans van Ditmarsch, Ji Ruan, and Yanjing Wang</i>	
True Lies and True Love	107
<i>François Schwarzentruber</i>	
Hintikka's World	124
<i>Floor Sietsma</i>	
A Letter to Jan	130
<i>Martin Stokhof</i>	
A Tale of Two Jans	132

<i>Elias Thijssse</i>	
Personal Note	133
<i>Christina Unger</i>	
Hup! Hup!	135
<i>Johan van Benthem</i>	
Working with Jan in Four Movements	137
<i>Tijs van der Storm</i>	
The Value of Alternative Semantics	143
<i>Stijn van Dongen</i>	
Rambling with Minkowski	163
<i>Heleen Verleur</i>	
Route 65	187
<i>Jurgen Vinju</i>	
The Syntax of Truth: A Grammar-based Approximation of Satisfiability	188
<i>Albert Visser</i>	
The Story of Urgh	200

Preface

Instead of an academic Festschrift, this is a true *liber amoricum* — a collection of contributions from friends. On the content level, these contributions demonstrate the influence Jan had on each of us in a diversity of fields at the intersection of logic, philosophy, computer science, and linguistics, as well as the inspiration we draw from it. On a personal level, they also express our gratitude for the past and our dear wishes for the future.

The book lives at <http://jve2017.herokuapp.com> in a tree-saving and dust-avoiding format that allows for code snippets and demos to be run. As many of the contributions included in the volume will illustrate, the choice of the online format is not so much an editorial convenience but rather captures the spirit of the Haskell way that Jan shaped and shared during his career. Alongside executable code, the collection comprises rigorous formalizations, informal text, argumentative dialogs, insights, personal notes, poems, and pictures. And as a special contribution it includes music that Heleen Verleur has composed for this occasion (here included as non-executable code; sorry to everyone who has not been live at the workshop to enjoy it at runtime).

We hope that the sum of all these contributions captures most of the relevant aspects of Jan's work and personality, and we are sure we speak for everyone when we say: It has been a great honor and an even greater pleasure to be a part of it.

Ștefan Minică
Christina Unger
Yanjing Wang
on behalf of all contributors

June 2, 2017
Amsterdam

For Jan: About the Pythagorean Theorem

Krzysztof R. Apt

When my children were at a secondary school I organised for a couple of years a 2-3 days long internship for a group of enthusiast children who wanted to learn more about mathematics and computer science. The main problem was to find adequate speakers. Jan never failed me and each time I asked him he gladly agreed to give a lecture followed by a short session during which he discussed solutions in Haskell to programming problems he posed. One time I followed his lecture out of curiosity. It started with the Pythagorean Theorem.

Jan presented a visual proof given in 12th century by an Indian mathematician, *Bhāskara*. It is based on the following two drawings:

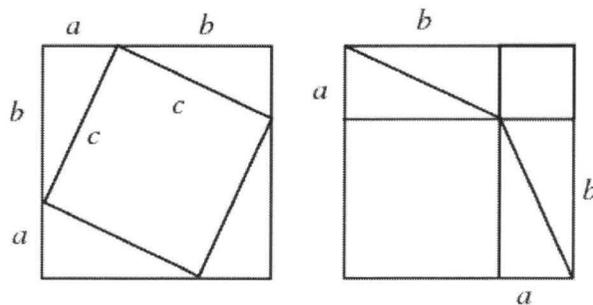


Figure 1: Bhāskara's proof of the Pythagorean Theorem

This is certainly a good start for a lecture for pupils. But recently I found another proof that might be even more appropriate for a school presentation because of a colourful story surrounding it. This proof was published by an American President, *James Abram Garfield*. Garfield was in office only for 199 days—he died in 1881 after being shot by an assassin. Garfield's proof is based on a drawing of a trapezoid given in the following figure.

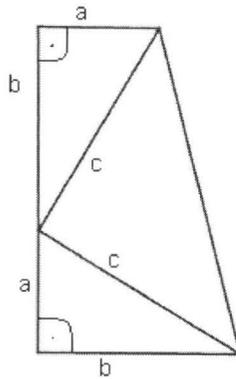


Figure 2: Garfield's proof of the Pythagorean Theorem

The area of the trapezoid equals the height times the average of the bases, so $(a + b) \frac{a+b}{2}$, i.e., $\frac{1}{2}(a + b)^2$. But it also equals the sum of the areas of the three triangles, that is, $\frac{1}{2}ab + \frac{1}{2}c^2 + \frac{1}{2}ab$, i.e., $ab + \frac{1}{2}c^2$. So $\frac{1}{2}(a + b)^2 = ab + \frac{1}{2}c^2$, that is, $(a + b)^2 = 2ab + c^2$, from which $a^2 + b^2 = c^2$ follows. Garfield concluded his article with this remark (see Dunham [2], page 99): "We think it something on which the members of both houses can unite without distinction of party."

To tease the pupils one could ask them what is the use of Pythagoras' theorem. The typical answer is that it helps one to produce a straight angle. Namely, take a rope, with equally spaced 11 knots on it (so that 12 equal units are formed) and use it to form a triangle with the side lengths of 3, 4, and 5. Since $3^2 + 4^2 = 5^2$, this triangle is right-angled. This is apparently how Egyptians used this knowledge to produce straight angles.

But a perceptive reader will notice that we cheated here. Namely, we used the *reverse* of the Pythagorean Theorem: if $a^2 + b^2 = c^2$, then the triangle with the side lengths a , b and c is right-angled. This implication also holds and is established in Euclid's *Elements* together with the proof of the original theorem.

One can go even further. Edsger Wybe Dijkstra [1] proved the following strengthening of the Pythagorean Theorem that covers both implications and four others, and makes no mentioning of right angles.

Theorem Consider a triangle with the side lengths a , b and c and the angles α , β and γ , lying opposite a , b and c . Then the signs of the expressions $a^2 + b^2 - c^2$ and $\alpha + \beta - \gamma$ are the same.

For example, for the triangle with the side lengths 3, 4 and 6 we have $3^2 + 4^2 < 6^2$, so $\alpha + \beta < \gamma$. Hence $180^\circ = \alpha + \beta + \gamma < 2\gamma$, so $90^\circ < \gamma$, which means that the triangle is obtuse.

References

[1] Edsger Wybe Dijkstra: On the theorem of Pythagoras. EWD-975, 1986. Available from <http://www.cs.utexas.edu/~EWD/ewd09xx/EWD975.PDF>.

[2] William Dunham: *Mathematical Universe*. John Wiley & Sons, 1994.

My story with Jan, in Haskell

Matteo Capelletti

This occasion has brought back to my memory many episodes from my life as a student in Utrecht under the supervision of Jan. I would like to use this space to recall some of them, and to share with you all some code and ideas that I developed through the years, both as a student and later just for fun, that in a way or another owe something to Jan.

Introduction

Jan has always been a source of inspiration for me both as a teacher and as a man. I remember him coming to the *Utrecht Institute of Linguistics* on Friday. I knew that at some point of the day he would drop by my office to see what I was up to. He would sit next to me, listen to my explanation of the experiments I was attempting and tell me that, well, this is very interesting, but it is difficult, and you need to prove your claims, you need to show that what you say actually works. I was playing a bit with Prolog at the time, but he recommended to learn Haskell, that is 'better'. That was a great advice that changed my way of approaching research, of exploring ideas and of thinking.

In this contribution I am going to present through code examples, some results which I find somewhat inspired by Jan. I report only parts of the code, and sometimes with some simplification to make it fit to the informal character of this contribution. I would be happy to share the code and also the literate version of it with whoever may be interested in seeing more.

The Catalan number

I read interesting contributions by Jan on many subjects. I especially liked the code accompanying most of them. When he started helping me on my linguistic research on efficient parsing for Categorical Grammars, it was not clear to me what his first assignment meant: "Write a function that returns all possible balanced brackets of length n ". In other words, a grammar generating the language:

$$S \rightarrow (S)S \mid \epsilon$$

Second question: "How many such strings are possible for any n ?"

It took me a while to find a nice way to do it. I spent an afternoon on this, but I wanted to be able to show something to Jan in our next meeting. Here is a revision of my solution, and the number was the Catalan number.

```
brack :: Int -> [String]
brack 0 = [""]
brack (n+1) =
  [ "(" ++ l ++ ")" ++ r | i <- [0..n],
                           l <- brack i,
                           r <- brack (n-i) ]
```

I banged my head for some days on this code. I had seen some Prolog, but my programming skills were low and functional programming was a rather different paradigm. I was so proud of this code when it finally worked, and I think Jan was happy too that I didn't give up and that we could move on for instance with a memoized version of this.

At the same time, with my other supervisor Michael Moortgat, we were wondering in what way generating all brackets is useful in parsing, but that's another thing. Actually, from this first coding experience I learned some very important things. At technical level, *recursion* and *list comprehension* which proved to be amazingly powerful tools. I remember the words of Jan in one of his classes on programming: "It looks like magic, but it's the power of recursion". And I often had the impression, in my meetings with Jan and in my studies in the Netherlands, to be initiated to some sort of *magia naturalis* while learning to use words that create and change things... At personal level, I learned not to give up when faced with something new and maybe initially difficult.

Later in my career, I used the above approach for language generation in some professional context. But the list comprehension approach is a simple, elegant and powerful one, that I used in several contexts, whenever a non-deterministic behavior had to be implemented. It has been found so simple and elegant that also other programming languages have adopted it from Haskell.

Beyond coding

After getting acquainted (but actually I should better say *addicted*) to Haskell and ready to disseminate my writings with code snippets, during one of his Friday visits Jan advised me not to use code in my abstracts! I was pretty confused and a bit frustrated, I shall say.

The recommendation was to use code in developing ideas, but to use logical notation for dissemination purposes. I never succeed much in the latter, but I tried. Here I present an example.

During my studies on Type Logical Grammar, I came across a contraction procedure that could be used to prove the theorems of the Non-Associative Lambek calculus (Lambek 1961). I present this in a sort of more logical way, but I think the similarity with the style of the previous code with list comprehension is clear. List comprehension is perfect to model *non-determinism* and generate *multiple solutions*.

The idea, which I present here shortly is that a sequent $a \rightarrow c$ is provable in NL if and only if the intersection of the reduction set r computed for the antecedent and of the expansion set e computed for the succedent is non-empty:

$$\vdash_{NL} a \rightarrow c \quad \text{iff} \quad r(a) \cap e(c) \neq \emptyset.$$

The contraction procedure e and r work as follow:

- 1) $e(a) = \{a\}$, if a is an atom
- 2) $e(a \otimes b) = \{ a' \otimes b' \mid a' \in e(a) \ \& \ b' \in e(b) \}$
- 3) $e(a/b) = \text{let mon be } \{ a'/b' \mid a' \in e(a) \ \& \ b' \in r(b) \} \text{ in}$
 - (a) mon
 - (b) $\cup \{ c \mid (c \otimes b')/b' \in \text{mon} \}$
 - (c) $\cup \{ c \mid a'/(c \setminus a') \in \text{mon} \}$
- 1') $r(a) = \{a\}$, if a is an atom
- 2') $r(a \otimes b) = \text{let mon be } \{ a' \otimes b' \mid a' \in r(a) \ \& \ b' \in r(b) \} \text{ in}$
 - (a) mon
 - (b) $\cup \{ c \mid c/b' \otimes b' \in \text{mon} \}$
 - (c) $\cup \{ c \mid a' \otimes a' \setminus c \in \text{mon} \}$
- 3') $r(a/b) = \{ a'/b' \mid a' \in r(a) \ \& \ b' \in e(b) \}$

Suppose for instance that we want to prove a sequent

$$(s/(n \setminus s)) \setminus s \rightarrow (s/(n \setminus s)) \setminus s$$

then we compute the reduction r of the antecedent:

$$r((s/(n \setminus s)) \setminus s) = \{(s/(n \setminus s)) \setminus s, n \setminus s\}$$

and the expansion e of the succedent:

$$e((s/(n \setminus s)) \setminus s) = \{(s/(n \setminus s)) \setminus s, n \setminus s\}$$

Because the intersection of the two sets is non-empty, we know that the sequent is provable in NL. A claim that I never succeeded in proving is that the cardinality of the (multiset) intersection is the number of proofs of the sequent, so that the procedure not only answers the provability problem, but also tells us how many distinct proofs a sequent has.

Beyond syntax

There are many other works by Jan on syntax and parsing that I studied with interest. I remember his Haskell versions of the CYK and Earley algorithms, his version of the monadic parser combinators, the proof net contraction algorithm. Not only the topics he proposed to me in our Friday sessions were interesting and challenging, but by looking at his code I could find clever solutions that could be reused in other contexts.

My dissertation was about syntax and theorem proving, but I was also fascinated by semantics and after completing my studies, I had the freedom to dedicate myself to that too, again starting from some reading of Jan's work.

I wanted to build an Haskell interpreter for predicate calculus. I ended up building an Haskell interpreter for Prolog, which would take as input programs like the following definition of `map` :

```
mapping =
  [
    map f [] [] :- [],
    map f (y : ys) (z : zs) :- [f y z, map x ys zs]
  ]
```

Here the program `mapping` is a list of clauses, each with a left-hand side term and a list of right-hand side terms.

The code includes several components among which a unification algorithm, some routines to refresh variables in terms and to normalize terms (adapted from Jan's code for lambda term normalization), ways of instatiating a query term to a program, which I am not reporting here. More interesting, and inspired by some of Jan's code for *exhausting* alternative possible combinations, the implementation of the depth-first exhaustion procedure which I report here (with some simplification):

```
eval :: OR -> Prog -> [Subs]
eval [] prog = []
eval ((f,[]):as) prog =
  f:eval as prog
eval ((f,q:qs):as) prog =
  eval (os++as) prog
  where
    os = [ (f'.f,rs ++ qs) | (f',rs) <- scan (f q) prog ]
```

As we said above, a program is a list of pairs made of a left-hand side term and a list of right-hand side terms:

$$[l_1 : -[r_1, \dots, r_n], \dots, l_k : -[r_1, \dots, r_m]]$$

The `eval` function is called initially on the pair `id` function and list of query terms `qs`, `[(id, [qs])]`. The `qs` are *AND* conditions that all need to be satisfied. This means that only when `qs` is empty we can return `f` as a substitution that provides one solution.¹ However there may be more than one solution. So the first argument of `eval`, of type *OR*, is of the form

$$[(f_1, [q_1, \dots, q_n]), \dots, (f_l, [q_1, \dots, q_m])]$$

and accumulates the right hand side of clauses that are instantiated. The list *OR* will consist of alternative branchings or alternative solutions to the initial problem. The `eval` function above generates all possible solutions.

The `scan` function, which we omit here, tries to match the current query term `q` under the current variable instantiation `f` with the left-hand side of a clause in a program `prog`. So it tries to unify `f q` and any `l`-term in a program.

Observe, about `(f',rs)` that

1. `scan**` returns all unifiable right-hand sides in the input program,
2. these are piled on top of the calling rhs `qs` as new *AND* conditions with new substitution function.
3. different possible clause instantiations give rise to new *OR* branchings.

I had some fun testing the procedure on some typical Prolog programs. For instance, we can define the following programs:²

```
addition =
  [
    add zero x x :- [],
    add (succ x) y (succ z) :- [add x y z]
  ]

multiplication =
  addition ++
  [
    mul zero x zero :- [],
    mul (succ y) x z :- [mul y x w, add x w z]
  ]

exponentiation =
  multiplication ++
  [
    exp zero x (succ zero) :- [],
    exp (succ x) y z :- [exp x y w, mul y w z]
  ]
```

Which unsurprisingly give us:

```
*Run> eval [add x y (num 5)] addition
[[ (x, 0), (y, 5) ], [ (x, 1), (y, 4) ], [ (x, 2), (y, 3) ], [ (x, 3), (y, 2) ], [ (x, 4), (y, 1) ], [ (x, 5), (y, 0) ]]
*Run> eval [mul (num 3) (num 5) x] multiplication
[[ (x, 15) ]]
*Run> eval [mul x (num 3) (num 15)] multiplication
[[ (x, 5) ]]^CInterrupted.
```

More interestingly, the full program is also capable of handling higher order term resolution as in the following example.

```
*Run> eval [map x y [3,4,5,6,7]] (mapping ++ addition)
[[ (x, (add 0)), (y, [3,4,5,6,7]) ],
 [ (x, (add 1)), (y, [2,3,4,5,6]) ],
 [ (x, (add 2)), (y, [1,2,3,4,5]) ],
 [ (x, (add 3)), (y, [0,1,2,3,4]) ]]
```

Beyond semantics

I knew towards the end of my studies in Utrecht that Jan was approaching new areas of investigation, that had to do with knowledge, belief, communication. It seemed all very interesting, but I had to write my thesis. A couple of years after my Dutch period, I met Jan at ESSLLI in Bordeaux, where he was teaching those things. Again, I found his lectures and explanations through code fascinating, and when I went back home, I decided to implement a MasterMind code breaker.

In the game of Mastermind at every guess the player is returned the number of colors that are in the right position and the number of color that are present but not in the right position. This information can be used to build a sort of propositional logic formulas that will act as a filter on the possible states of the world. Formulas are for instance `At (x, i)`, meaning that color `x` is at position `i`, or `In x` meaning that color `x` should be present. Then formulas built from connectives. The `eval` function below takes a sequence of colors (a world) `xs` and a formula, and checks if the formula is true in that world

```
eval :: String -> F -> Bool
eval xs (At (x,i)) = xs!!i==x
eval xs (In x) = x `elem` xs
eval xs (Not x) = not (eval xs x)
eval xs (And ys) = all (eval xs) ys
eval xs (Or ys) = any (eval xs) ys
```

Given a set of possible colors and a predefined length of the sequence to guess, the set of possible worlds is defined.

Subsequent guesses are used as filters. Here is an example run of the code with possible colors `[a, b, c, d, e, f]` and sequence length 4:

```

*Main> test "aacc"
"Code: aacc"
"Guess: abcd"
"Worlds:"
1295
"Outcome: (2,0)"
"Val: v^[a0,b1,-c2,-d3,-c,-d],^[a0,c2,-b1,-d3,-b,-d],^[a0,d3,-b1,-c2,-b,-c],^[b1,c2,-a0,-d3,-a,-d],^[b1,d3,-a0,-c2,-a,-c],^[c2,d3,-a0,-b1,-a,-b]]"
"Guess: abef"
"Worlds:"
95
"Outcome: (1,0)"
"Val: v^[a0,-b1,-e2,-f3,-b,-e,-f],^[b1,-a0,-e2,-f3,-a,-e,-f],^[e2,-a0,-b1,-f3,-a,-b,-f],^[f3,-a0,-b1,-e2,-a,-b,-e]]"
"Guess: aacc"
"Worlds:"
15
"Success after 2"

```

In the above, `Outcome: (2,0)` states that 2 colors that are in the right position and 0 colors are present but not in the right position. This is encoded in the formula after `val1`: which uses prenex notation with `v` for *or*, `^` for *And* and `-` for *not*. For instance:

```
v^[b0,-a],^[e2,f]]
```

Read: `b` is at `0` and `a` is not in the code, or `e` is not at `2` and `f` is in the code

The initial guess is always "abcd". The set of possible worlds is initially 1,295 ($6^4 - 1$). The first guess gives two colors in the right position, and is translated in the formula in `val1`. This formula is used then as a filter on the set of possible worlds and the program randomly guesses another color from the remaining set. In two iterations the code is broken.

Conclusion

After my studies, I didn't find a job as a Haskell programmer. Actually when asked "what programming language do you know?" I had a hard time explaining what Haskell was and that yes, I can program. However, when I met some of those who can program in the private sector, I often thought that they may benefit from learning some Haskell. I myself have used it sometimes also at work, but more than it, it's been useful to learn to see problems in their various components, in gaining an analytical view of the problem, and learning to design solutions that not only do work, but are also efficient and elegant. Under the guidance of Jan, learning Haskell has been an exciting, lively, always interesting adventure.

Notes

1. For instance mapping `f` on the free variables in `qs` returns the variable instantiations that satisfy the query. ↩
2. Observe that in the code, `succ` does not have the standard Haskell meaning, but is a term constructor for integers. ↩

Unavoidability of Induction

Kees Doets

For Jan, at the occasion of his farewell

1

This began August 17th, 2014.

In an episode of the Dutch TV series "Zomergasten", Ionica Smeets (professor scientific communication, Leiden University) presented a BBC production with Marcus du Sautoy on Euclid's theorem about the number of primes.

The famous argument: whenever you have a finite list of (prime) numbers p_0, \dots, p_{n-1} , none of them will divide the number $m = 1 + \prod_{i < n} p_i$ (the point of the construction being that all divisions have remainder 1) and, hence, no (prime) divisor of m occurs on your list.

Which inevitably entails the conclusion that the number of primes cannot be finite.

2

The exposition was meant for a general TV audience: it should be both watertight and simple. Simple it was, but (admittedly, sometime later) I observed a hidden use of induction. This becomes more conspicuous when asking the question: why should m have any prime divisor at all?

The answer is: search for the least number > 1 that divides it, by consecutively trying 2, 3, 4, ... (In the worst case, this search goes all the way up to m itself if, by coincidence, this number happens to be prime.) Finally, a least divisor obviously cannot fail to be prime.

So we used the fact that a certain non-empty set of natural numbers has a least element. And most of us know (cf. part 7): this is just an instance of induction.

The intended audience will probably be unfamiliar with induction. (A first year math student at my first encounter, it took some effort before I got familiarized just a bit.) But what about this *Least number principle*? A least number in a clear-cut non-empty set of natural numbers — who of the many, watching TV on August 17th 2014, will have doubted its existence? And if such persons do not exist, is it within reason to call this cheating a little?

3

I wrote to Ionica. She dismissed my criticism by referring to the theorem on prime decomposition. OK, this well-known fact is taught at primary school. Nevertheless, in the given situation, this felt as overkill.

4

Perhaps I should have but I did not consider writing du Sautoy. Instead, I grabbed for van Eijck and Visser [1]. The first theorem in their book concerns the irrationality of $\sqrt{2}$. And —bingo!—the second one is Euclid's. The proof slightly deviating, but, nevertheless, there it was: least divisors exist.

I wrote to Jan. He agreed there was a minor point here to make explicit.

5

Two years passed. The retirement of Jan was approaching and I mused about Doets and van Eijck [2] for some handhold. Was I surprised when I noticed Euclid not as second, but even as first theorem — be it on p. 102. However, the existence of least divisors was somewhat carelessly mentioned already on p. 4, thereby firmly closing full circle on yours truly.

6

Some serious answers came from an unexpected source. At some stage, I had told Krzysztof Apt about the apparent need for induction in proving Euclid's theorem. On one of his famous bicycle trips with fellow-mathematicians, Krzysztof mentioned this to Zofia Adamowicz. She informed us that things were more intricate when looked at in the setting of formalized arithmetic.

Shepherdson († 2015)¹ showed long ago that induction for quantifier-free formulas doesn't suffice to prove the Euclid theorem. However, it isn't the existence of the least divisor which is the bottleneck here (although the induction uses a bounded formula). It is the existence of the number $\prod_{i < n} p_i$, or, at least, of large numbers with many divisors such as factorials. In the usual formalisation the only operations are those for successor, addition and multiplication. Others (exponentiation, factorial,...) have to be defined and it requires proof to show they are total. This is where more serious induction is required. Finally, Woods proved Euclid using induction for bounded formulas and the totality of $x^{\log(x)}$, but his proof is very different from the usual one.

7

For completeness' sake, here follow the relevant forms of induction.

The standard formulation:

(Induction) *If A is a set of natural numbers such that (i) $0 \in A$ and (ii) $\forall n \in A (n + 1 \in A)$, then A contains every natural number.*

Existence of the least divisor follows immediately from either of the following two equivalents.

(Least number principle) *Every non-empty set of natural numbers has a least element.*

(Termination) *There doesn't exist an infinite descending sequence of natural numbers $n_0 > n_1 > n_2 > \dots$.*

Most of the six implications between these principles are easily proved. Equivalence of the last two is remarkable, as the first one is a second-order statement whereas the second one employs infinitary language. (The Axiom of Choice is involved, but that is a different story.)

Only one of the implications requires a little trick:

(Induction) implies (Least number principle): Assume that P is a set of natural numbers without least element. Assuming Induction, we will show it to be empty.

Define (this is the trick) $Q = \{n \mid \forall m < n(m \notin P)\}$.

Note that Q and P are disjoint (a common element would be a least one of P).

Thus, for P to be empty, it suffices to show that every number is in Q . Here is where we employ induction:

First, $0 \in Q$ holds vacuously. Next, assume that $n \in Q$. In order that $n + 1 \in Q$, choose any $m < n + 1$; we have to show $m \notin P$.

Since $m < n + 1$, we have either $m = n$ or $m < n$. In the first case, the required $m \notin P$ follows because of $n \notin P$, in the second case this follows because of $n \in Q$.

8

Krzysztof Apt sent me the paper [3] that lists a wealth of 147 references on Euclid's theorem. However, induction isn't the issue here. (Nor, I suspect, is it in any of these references.) It is the remarkable and unexplained fact that nearly all of them are historically incorrect in stating that Euclid's is among the earliest proofs by contradiction (that derive a contradiction from *the negation of what has to be proved*; in this particular case: that only finitely many primes exist).

Krzysztof also observed that, if N is any natural number, the least number > 1 that divides $N! + 1$ will be a prime greater than N (which, accidentally, is $\leq N! + 1$). Obviously, this immediately shows there are infinitely many primes.

Notes

¹. <http://www-groups.dcs.st-and.ac.uk/history/Biographies/Shepherdson.html> ↵

References

- [1] Jan van Eijck, Albert Visser, *Inzien en Bewijzen*. Amsterdam University Press 2005.

[2] Kees Doets, Jan van Eijck, *The Haskell Road to Logic, Maths and Programming*. King's College 2004.

[3] Michael Hardy, Catherine Woodgold, Prime Simplicity. *The Mathematical Intelligencer* **31** (4) 2009, pp. 44–52.

From Zero to Logic in Haskell

Malvin Gattinger

My first contact with Jan and Haskell coincide. At the beginning of my second year as a Master of Logic student I took his course "Functional Specification of Algorithms".

My previous programming experience had been completely outside universities. Moreover, it was mainly in PHP which you might call the opposite of Haskell. Not just imperative and stateful as hell, but with so many weird parts that there are [dedicated forums](#) to make fun of them. Maybe this is why I usually did not think about connections between my interests in mathematics and computers.

Jan easily changed this and showed me a way to merge logic and programming. The final blow to my imperative upbringing happened towards the end of the course. Jan gave a short introduction to DEMO, the epistemic model checker. Since then my favorite example of how well Haskell accommodates logic is the comparison between mathematical definitions as we write them in a paper and their implementations. For example, consider the syntax of Public Announcement Logic (PAL):

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid K_i\phi \mid [\phi]\phi$$

This can be easily translated to a new `data` type:

```
type Prop = Char
type Agent = String
data Form = Top | P Prop | Neg Form | Con Form Form | K Agent Form | Ann
          Form Form
```

The symmetry continues when we interpret the language. Here is the standard truth definition for PAL, saying when formulas are true in a pointed model:

$$\begin{aligned}
\mathcal{M}, w \models \top &\Leftrightarrow \text{always} \\
\mathcal{M}, w \models p &\Leftrightarrow p \in V(w) \\
\mathcal{M}, w \models \neg\phi &\Leftrightarrow \text{not } \mathcal{M}, w \models \phi \\
\mathcal{M}, w \models \phi \wedge \psi &\Leftrightarrow \mathcal{M}, w \models \phi \text{ and } \mathcal{M}, w \models \psi \\
\mathcal{M}, w \models K_i \phi &\Leftrightarrow \forall v \sim_i w : \mathcal{M}, v \models \phi \\
\mathcal{M}, w \models [\phi]\psi &\Leftrightarrow \mathcal{M}, w \models \phi \Rightarrow \mathcal{M}^\phi, w \models \psi
\end{aligned}$$

How do we write this in Haskell? First we need a definition of models.

```

type World = Int
data Model = Mo {worlds :: [World], val :: World -> [Prop], rel :: Agent
-> World -> [World]}

```

Now the semantics given by \models above can be written as a function from pointed models and formulas to booleans. The helper function `!` implements the update from \mathcal{M} to \mathcal{M}^ϕ .

```

eval :: (Model,World) -> Form -> Bool
eval (_,_) Top           = True
eval (m,w) (P p)        = p `elem` (val m w)
eval (m,w) (Neg phi)    = not (eval (m,w) phi)
eval (m,w) (Con phi psi) = eval (m,w) phi && eval (m,w) psi
eval (m,w) (K i phi)    = and [eval (m,v) phi | v <- rel m i w]
eval (m,w) (Ann phi psi) = eval (m,w) phi <= eval (m ! phi,w) psi

(!) :: Model -> Form -> Model
(!) m phi = m
  { worlds = filter (\w -> eval (m,w) phi) (worlds m)
  , rel = \i w -> filter (\w -> eval (m,w) phi) (rel m i w) }

```

This minimalistic toy variant of DEMO can be used as follows:

```

myModel :: Model
myModel = Mo [0,1] myval myrel where
  myval 0 = "pq"
  myval 1 = "p"
  myrel "Anne" 0 = [0,1]
  myrel "Anne" 1 = [0,1]
  myrel "Bob" 0 = [0]
  myrel "Bob" 1 = [1]

λ> eval (myModel,0) (K "Bob" (P 'q'))
True
λ> eval (myModel,0) (K "Anne" (P 'q'))
False
λ> eval (myModel,0) (Ann (P 'q') $ K "Anne" (P 'q'))
True

```

Why would you want to use anything else to implement Logics and model checkers?

```

eval :: (Model,World) -> Form -> Bool
eval (M,w) ⊨ ⊤           ⇔ always
eval (M,w) ⊨ p           ⇔ p ∈ V(w)
eval (M,w) ⊨ ¬ϕ          ⇔ not (M,w ⊨ ϕ)
eval (M,w) ⊨ ϕ ∧ ψ       ⇔ M,w ⊨ ϕ and M,w ⊨ ψ
eval (M,w) ⊨ Kϕ          ⇔ ∀v ~i w : M,v ⊨ ϕ
eval (M,w) ⊨ [!ϕ]ψ       ⇔ M,w ⊨ ϕ w ⇒ Mϕ,w ⊨ ψ

```

Given this background, I often look at new definitions of semantics for a new logic and immediately wonder what they would look like in code. Can we easily translate all logics and their semantics to Haskell? Of course not. The language is more restrictive than mathematical notation, but this can be seen as a feature. When I started to implement the Logic of Agent Types and Questions from Liu & Wang 2013 one obstacle was this definition (adapted from page 138):

$$\mathcal{M}, w \models_{\mu} [!_a]\phi \Leftrightarrow \text{for all } \psi : \mathcal{M}, w \models_{\mu} [!_a\psi]\phi$$

The intended meaning of $[!_a]\phi$ is "No matter how agent a answers the current question μ , ϕ will be true afterwards." On the right side of the definition we quantify over all formulas to represent all possible answers. But of course we can not run through infinitely many ψ in an implementation that should ever be run (and finish). But in this case there is an easy way out: The logic only formalizes binary questions μ , so the only relevant answers are equivalents of μ and $\neg\mu$. Thus we do not actually care about all formulas, only those two,

and the logic can still be implemented easily.

One of the things I learned from Jan is that in situations like this we can realize that implementation is not a one-way street: we might as well go back and change the definition that we wanted to implement (and this is actually what (Liu & Wang 2013) already do implicitly on page 138). Haskell thus prevents us from defining something in a non-computable or non-constructive way if there is no real reason to do so.

Normative Notions by Colored Actions

Fengkui Ju

Jan is a very nice professor. I would like to thank him for all the kind help to me and my family. I wish Jan all the happy things. I also hope that he could visit Beijing again some day with his family.

Jan and I like a semantic setting: labeled and colored transition systems with preference among colors. I design a language to talk about this semantic setting. In it, a few normative notions can be defined. The semantics is from our previous joint work.

1 Language

Let k be a natural number and $C = \{c_0, \dots, c_k\}$ a set of $k + 1$ colors. Let \preceq be a reflexive and transitive order on C . $c_i \preceq c_j$ indicates that c_j is at least as good as c_i . Let \prec be the strict version of \preceq . $c_i \prec c_j$ denotes that c_j is better than c_i . Conceptually, one has preference among colors and \preceq is its preference.

Let Π_0 be a finite set of atomic actions and a range over it. Let Φ_0 be a countable set of atomic propositions and p range over it. Let i range over the set $\{0, \dots, k\}$. Define mutually recursively a set $\Pi_{\chi L}$ of actions and a set $\Phi_{\chi L}$ of propositions as follows:

$$\begin{aligned} \alpha ::= a \quad | \quad \mathbf{id} \quad | \quad (\alpha; \alpha) \quad | \quad (\alpha \cup \alpha) \quad | \quad \alpha^* \quad | \quad \tilde{\alpha} \quad | \quad \phi? \\ \phi ::= p \quad | \quad \top \quad | \quad c_i \quad | \quad \neg\phi \quad | \quad (\phi \wedge \phi) \quad | \quad \mathbf{X}\phi \quad | \quad (\phi \mathbf{U}\psi) \quad | \quad \mathbf{A}_\alpha\phi \end{aligned}$$

Doing \mathbf{id} is doing *nothing*. Doing $\tilde{\alpha}$ is *refraining* from α . The reading of the featured formulas is as follows:

1. c_i : the next transition has the color c_i .
2. $\mathbf{X}\phi$: ϕ will be the case in the *next* moment.
3. $(\phi \mathbf{U}\psi)$: ϕ will be the case *until* ψ .
4. $\mathbf{A}_\alpha\phi$: no matter how the agent will perform α , ϕ is the case *now*.

Note that ϕ in $\mathbf{A}_\alpha \phi$ might contain temporal operators and ϕ being true now might mean something else being true in the future.

Here are some derivative expressions:

1. $\mathbf{1} := a_0 \cup \dots \cup a_n \cup \mathbf{id}$ where $\Pi_0 = \{a_0, \dots, a_n\}$: doing it means doing a *basic* action.
2. $\mathbf{F}\phi := (\top \mathbf{U}\phi)$: ϕ will be the case.
3. $\mathbf{G}\phi := \neg \mathbf{F}\neg\phi$: ϕ will *always* be the case.
4. $\mathbf{D}\phi := \mathbf{F}(\neg \mathbf{X}\top \wedge \phi)$: ϕ will be the case at the *end*.
5. $\mathbf{E}_\alpha \phi := \neg \mathbf{A}_\alpha \neg\phi$: the agent has a way to perform α s.t. ϕ is the case now.
6. $[\alpha]\phi := \mathbf{A}_\alpha \mathbf{D}\phi$: no matter how the agent will perform α , ϕ will be the case after α is done. This is the classical *box* modality.
7. $\langle \alpha \rangle \phi := \mathbf{E}_\alpha \mathbf{D}\phi$: the agent has a way to perform α s.t. ϕ will be the case after α is done. This is the classical *diamond* modality.
8. $\mathbf{C}\phi := \mathbf{E}_{\mathbf{1}^*} \phi$: the agent has the *ability* to make ϕ true *now*.

ϕ in $\mathbf{C}\phi$ might also contain temporal operators and making ϕ true now might mean making something else true in the future. In reality, when we say that someone has the ability to make something true, we usually mean that he has the ability to make it true in the future.

2 Models

Let \mathcal{C} and \preceq be specified as before. $\mathfrak{M} = (W, \{R_a \mid a \in \Pi_0\}, R_{\mathbf{id}}, \mathcal{C}, \preceq, \sigma, V)$ is a model if

1. W is a nonempty set of states
2. $R_a \subseteq W \times W$ for any $a \in \Pi_0$
3. $R_{\mathbf{id}} = \{(x, x) \mid x \in W\}$
4. σ is a function from R to \mathcal{C} where $R = \bigcup \{R_a \mid a \in \Pi_0\} \cup R_{\mathbf{id}}$

5. V is a function from Φ_0 to 2^W

For any transition (w, u) , $\sigma(w, u)$ is called the color of it. We say that (w, v) is at least as good as (w, u) if $\sigma(w, u) \preceq \sigma(w, v)$, and (w, v) is better than (w, u) if $\sigma(w, u) \prec \sigma(w, v)$. Note that the preference order concerning colors is the same in all models.

Let w be a state. For any u s.t. $(w, u) \in R$, (w, u) is a *best* transition from w if there is no v s.t. $(w, v) \in R$ and $\sigma(w, v) \succ \sigma(w, u)$, and (w, u) is a *worst* transition from w if there is no v s.t. $(w, v) \in R$ and $\sigma(w, v) \prec \sigma(w, u)$. Best/worst transitions are *relative* notions.

A *finite* sequence $w_0 \dots w_n$ of states is called a *path* if $w_0 R \dots R w_n$. Specially, w is a path. A path $w_0 \dots w_n$ is *legal* if for any $i < n$, (w_i, w_{i+1}) is a best transition, and *evil* if for any $i < n$, (w_i, w_{i+1}) is a worst transition. Trivially, w is a both legal and evil path.

(continued on next page)

Normative Notions by Colored Actions

Fengkui Ju

(continued from previous page)

3 Normative Notions

Recall that $C = \{c_0, \dots, c_k\}$ is a set of $k + 1$ colors. For any $i \leq k$, let $\Delta_{c_i}^> = \{c_j \mid c_j \succ c_i\}$ and $\Delta_{c_i}^< = \{c_j \mid c_j \prec c_i\}$. In an intuitive sense, $\bigvee \Delta_{c_0}^>$, the disjunction of the propositional constants in $\Delta_{c_0}^>$, says that the color of the next transition is better than the color c_0 . $\neg \bigvee \Delta_{c_i}^>$ says that the color of the next transition is not better than c_i . $\neg \bigvee \Delta_{c_i}^<$ indicates that the color of the next transition is not worse than c_i . With the path quantifier \mathbf{A} and the special action $\mathbf{1}$, we can express best/worst transitions:

1. $\mathfrak{b} := (c_0 \wedge \mathbf{A}\mathbf{1} \neg \bigvee \Delta_{c_0}^>) \vee \dots \vee (c_k \wedge \mathbf{A}\mathbf{1} \neg \bigvee \Delta_{c_k}^<)$: the next transition is a best transition.
2. $\mathfrak{w} := (c_0 \wedge \mathbf{A}\mathbf{1} \neg \bigvee \Delta_{c_0}^<) \vee \dots \vee (c_k \wedge \mathbf{A}\mathbf{1} \neg \bigvee \Delta_{c_k}^>)$: the next transition is a worst transition.

The following three constraints offer the language the power to express best/worst transitions:

1. there are only finitely many atomic actions;
2. there are only finitely many colors;
3. the preference order among colors is the same in all models.

$\mathbf{X}\top \rightarrow \mathfrak{b}$ says that the next transition is a best one if it exists and $\mathbf{X}\top \rightarrow \mathfrak{w}$ that the next transition is a worst one if it exists. Fix an action α and a proposition ϕ . Some meaningful things about α and ϕ can be expressed by use of the ingredients from the sets $\{\mathbf{A}, \mathbf{E}\}$, $\{\mathbf{G}, \mathbf{F}\}$, $\{\mathbf{X}\top \rightarrow \mathfrak{b}, \mathbf{X}\top \rightarrow \mathfrak{w}\}$ and $\{\neg, \wedge, \rightarrow\}$.

One of them is $\mathbf{E}_\alpha(\mathbf{G}(\mathbf{X}\top \rightarrow \mathbf{b}) \wedge \phi)$. It says that there is a legal path of α s.t. ϕ is true at it. This *in some sense* means that the agent is *permitted* to do α to make ϕ true. It can also be read as that the agent can *legally* do α to realize ϕ . Define $\mathcal{P}_\alpha\phi$ as $\mathbf{E}_\alpha(\mathbf{G}(\mathbf{X}\top \rightarrow \mathbf{b}) \wedge \phi)$. Quite a few normative notions can be defined in terms of $\mathcal{P}_\alpha\phi$:

1. $\mathcal{P}\alpha := \mathcal{P}_\alpha\top$: the agent is permitted to do α .
2. $\mathcal{P}\phi := \mathcal{P}_1\phi$: the agent is permitted to make ϕ true.
3. $\mathcal{F}_\alpha\phi := \neg\mathcal{P}_\alpha\phi$: the agent is *forbidden* to do α to make ϕ true.
4. $\mathcal{F}\alpha := \mathcal{F}_\alpha\top$: the agent is forbidden to do α .
5. $\mathcal{F}\phi := \mathcal{F}_1\phi$: the agent is forbidden to make ϕ true.
6. $\mathcal{O}_\alpha\phi := \neg\mathcal{P}_1\neg\phi \wedge \neg\mathcal{P}_{\bar{\alpha}}\top$: the agent is *obligated* to do α to make ϕ true.
7. $\mathcal{O}\alpha := \mathcal{O}_\alpha\top$: the agent is obligated to do α .
8. $\mathcal{O}\phi := \mathcal{O}_1\phi$: the agent is obligated to make ϕ true.

In an intuitive sense, what follows is the case:

1. *one is permitted to do α to make ϕ true* implies *he is permitted to do α and he is permitted to make ϕ true*, but not vice versa.
2. *one is forbidden to do α to make ϕ true* does not imply *he is forbidden to do α and does not imply he is forbidden to make ϕ true* either.
3. *one is obligated to do α to make ϕ true* implies *he is obligated to do α and he is obligated to make ϕ true*, and vice versa.

As the triple negation of $\mathcal{P}_\alpha\phi$, the formula $\neg\mathcal{P}_{\bar{\alpha}}\neg\phi$ is read as this: the agent is forbidden to refrain from α to make ϕ false. It does not imply *he is forbidden to refrain from α* . It does not imply *he is forbidden to make ϕ false*. Therefore, $\neg\mathcal{P}_{\bar{\alpha}}\neg\phi$ does not mean that the agent is obligated to do α to make ϕ true.

Implementation

```
-- This is my first Haskell code. Given a regular action, it can generate
the set of computation sequences of it. I got a lot of fun from his Haskell
course.
```

```
module For_Jan where

import Data.List

data Action = Ato Integer
  | Com Action Action
  | Cho Action Action
  | Sta Action
  deriving Eq

a,b,c,d :: Action

a = Ato 0
b = Ato 1
c = Ato 2
d = Ato 3

instance Show Action where
  show (Ato 0) = "a";
  show (Ato 1) = "b";
  show (Ato 2) = "c";
  show (Ato 3) = "d";
  show (Ato n) = 'a' : show n

mer :: [[Action]] -> [[Action]] -> [[Action]]
mer x y = [s ++ t | s <- x, t <- y]

ite :: [[Action]] -> [[Action]]
ite x = union x (mer x (ite x))

cs :: Action -> [[Action]]
cs (Ato n) = [(Ato n)]
cs (Com x y) = mer (cs x) (cs y)
cs (Cho x y) = union (cs x) (cs y)
cs (Sta x) = union [[]] (ite (cs x))
```

An Exercise in Exercises

Paul Klint

Abstract

It is well known that a student learns best when he or she directly applies the knowledge that has been presented by teacher or textbook. Interactive, computer-based, exercises have the potential of even better delivering on this promise since they can be offered, checked and repeated anytime at the convenience of the student. However, designing and implementing interactive exercises is difficult and it is not simple to make them both educational and entertaining for the student. I describe an experiment—in the context of the Rascal meta-programming language—in creating a domain-specific language for authoring interactive exercises.

This short note is dedicated to Jan van Eijck on the occasion of his retirement. Since Jan is a teacher at heart, he knows that exercises are essential for education and he may appreciate this effort to achieve simpler authoring of interactive exercises.

Teaser: even random testing plays a role here!

Jan thanks for our many, pleasant, interactions over the years.

Requirements

As every software engineer knows, starting with requirements is a solid way to embark on a project like this. So let's begin to write them down:

	Description
R1	The effort to describe an exercise should be small.
R2	Exercises should be reused as much as possible. Possibly by randomly generating exercises from a given template.
R3	Once a student has filled in an answer to an exercise, it can be automatically checked.
R4	When a student enters a good answer, it is rewarded with positive feedback.
R5	When a student enters a wrong answer, the error is explained and the student is encouraged to make a new attempt.
R6	Two categories of questions should be supported: knowledge questions and source code questions.
R7	To make exercises more entertaining, different styles of questions should be supported.

Design

How to design a system that satisfies these requirements? Our approach will be to create a small domain-specific language (let's call it EDL for *Exercise Description Language*) that supports generation of random exercises (R1, R2) in various styles (R6, R7) that can be checked automatically (R3). No particular attention will be given here to (R4, R5) but they are handled by the implementation of EDL. I will present the following question categories:

- *Multiple-choice question*: the student has to select one (or more than one) of the choices that are presented.
- *Fill-in-the-hole question*: an incomplete source code fragment is presented and the student must complete it.
- *Move code question*: place given source code fragments in the right order.
- *Click-all-cases question*: a text is presented and the student must click on all occurrences of a text with a certain property.
- *Reorder question*: a number of sentence fragments must be reordered to form correct sentences.

Multiple-choice questions

The multiple-choice question is the prototypical interactive exercise and presents a number of choices to the student. Each question starts with the keyword `question` and an introductory text. Each question ends with the keyword `end`. Our description of this multiple-choice questions consists of a number of choices, where each choice (indicated by the keyword `choice`) states whether it is correct or not (`y` or `n`), the actual text of the choice, and an additional explanatory text; `|||` is used as separator.

```
question Which means of transportation is faster:
  choice n ||| Apache Helicopter ||| The speed of an Apache is 293 km/hou
r
  choice y ||| High-speed train ||| The speed of a high-speed train is 5
70km/hour
  choice n ||| Ferrari F430 ||| The speed of a Ferrari is 315 km/hour
  choice n ||| Hovercraft ||| The speed of a Hovercraft is 137 km/hou
r
end
```

This will be presented to the student as follows:

Question 1

Which means of transportation is faster:

- Apache Helicopter
- High-speed train
- Ferrari F430
- Hovercraft

Check It

Fill-in-the-hole questions

A fill-in-the-hole question presents a text to the student and asks to fill-in missing text. This can range from the answer to a simple arithmetic question (e.g., $2 * 3 == ?$) to filling-in statements or declarations in a source code fragment. The general pattern is either:

- A single equality. Here there are two subcases:
 - An equality in which a single hole occurs on the right-hand side, e.g., $2 + 3 == ?$.
 - An equality in which a single hole occurs that is nested in the left-hand side or right-hand side of the equality, e.g. $2 + ? == 5$.
- A code fragment that contains one or more holes followed by a number of tests. After filling in the holes, all tests should pass.

Fill-in-the-hole: single equality

The general idea is not to write a *separate* exercise for each possible case, but to write an exercise *template* from which all cases can be generated. Let's consider an exercise for rehearsing multiplication on numbers: instead of defining exercises for a large combination of arguments and describing the outcome for each case (e.g., $1 * 1 == ?$, $2 * 3 == ?$, and so on, where $?$ marks the answer to be filled in by the student) we write a single template

```
$gen(int, A) * $gen(int, B).
```

EDL can be best understood as a macro language to generate text where directives starting with $\$$ indicate generation-time expansion/computation actions. The abovetemplate can be read as follows:

1. $\$gen(int, A)$: Generate a random integer and bind it to A .
2. $\$gen(int, B)$: Generate another random integer and bind it to B .
3. We are therefore looking for a value that is equal to $A*B$.
4. Create an exercise consisting of a test function containing the following equality test:
 - i. the value of A ,
 - ii. a multiplication operator ($*$),
 - iii. the value of B ,
 - iv. an equals operator ($==$),
 - v. a text entry field for the answer (or $?$ in this text).

Although this could generate the above exercise example, it will also generate exercises that are less suited for a beginner such as $1730304988 * 982060868 == ?$. This illustrates that we need a slightly more sophisticated type system to specify an interval of

integers (e.g., `int[2,5]` to denote an integer in the interval $[2,5]$) or the size of lists or sets (e.g. not only `list[int]` for an arbitrary length list of integers but also `list[int, 2, 5]` to denote lists of integer of length in the interval $[2,5]$).

Without further ado, we can now show the complete specification of a multiplication exercise:

```
question Replace the text box by the result of the multiplication and make
the test true:
  expr multiplication $gen(int[2,5],A) * $gen(int[2,5],B)
end
```

The keyword `expr` indicates that we are looking for the value of the given expression and `multiplication` is the name of the test that will be generated.

Here is a screen shot showing how this question will be presented to the student:

Question 2

Replace the text box by the result of the multiplication and make the test true:

```
module Question2

test bool multiplication() =
  4 * 6 ==  ;
```

Each question is presented as a self-contained Rascal module (possibly containing imports and declarations of auxiliary functions or datatypes) that can also be executed outside the tutoring environment. Also observe that even this simple exercise is framed as a test function, which is in Rascal a Boolean function prefixed with the keyword `test`.¹

There are cases where even more control over the generated values in an exercises is needed. A case in point is an exercise for set intersection (in Rascal denoted by `&`). A naive approach would be to write: `$gen(set[int],A) & $gen(set[int],B)`, but due to the random generation of the set values `A` and `B`, there is a high change that these sets are disjoint and that the answer of the exercise is in most cases the empty set. A more sophisticated approach is as follows:

```
$eval($gen(set[int]) + $gen(set[int],B)) & $eval($gen(set[int]) + $use(B)
).
```

This introduces two new features: `$eval` evaluates its argument and yields its value, while `$use` inserts the value of a generated variable. We can read the above recipe as follows. First consider the part left of the intersection operator `&` :

1. `$gen(set[int])` : no variable is included, therefore generate an (anonymous) set value;
2. `$gen(set[int],B)` : generate set value `B` ;
3. `$eval($gen(set[int]) + $gen(set[int],B))` : compute the union of `A` and `B` by evaluating the text consisting of the first set value, the symbol `+` (the set union operator) and the set value `B` .
4. The result is a set value, call it `L` .

For the right operand of `&` a similar story holds:

1. `$eval($gen(set[int])` : generate another (anonymous) set value;
2. `$use(B)` : reuse the previously generated set value `B` .
3. `$eval($gen(set[int]) + $use(B))` : evaluate the text consisting of the generated set value, the symbol `+` and the set value `B` .
4. The result is set value call it `R` .

The exercise as a whole is now `L & R` , where we know (by construction) that `L` and `R` have the set value `B` in common thus guaranteeing a "more interesting" answer to the exercise.

Here is the complete description of the set intersection question:

```
question Replace the text box by the result of the intersection and make
the test true:
  expr setIntersection $eval($gen(set[int]) + $gen(set[int],B)) &
                        $eval($gen(set[int]) + $use(B))
end
```

And here is a screen shot of the end result:

Question 3

Replace the text box by the result of the intersection and make the test true:

```
module Question3

test bool setIntersection() =
  {-84,57,-87,74,-44,77,56} & {-85,31,-71,92,-87,74,-44,77,56} ==  ;
```

Fill-in-the-hole: source code

The single equality question is a degenerated case of a source code question. A simple example is the case where a function name has to be inserted in a given code fragment:

```
question Replace the text box by a function name and make the test true:
  prep import List;
  expr listFunction $answer(size)($gen(list[int][1,10]))
end
```

Here `$answer(size)` defines the required answer (needed for checking the answer) but is replaced by a text box when presented to the student:

Question 4

Replace the text box by a function name and make the test true:

```
module Question4
import List;

test bool listFunction() =
   ([80, -53, -92, -19, 58, -46, -25, 7, 8, -12]) == 10;
```

The general case (not illustrated here) is arbitrary source code that contains a number of holes to be filled in. The source code should contain one or more test functions² to assert the correctness of the answers provided by the student.

Move code questions

The idea of a move code question is to present a number of source code fragments and to ask the student to place them in the correct order with correct indentation. We illustrate this with a simple function to print a table of squares:

```
question Create a function to print squares by placing all code fragments
  in the grey box in the right order with the right indentation:
movable
void squares(int n){
-----
    println("Squares from 1 to " + n);
-----
    for(int i <- [1 .. n + 1])
-----
        println(i + " squared = " + (i * i));
-----
}
end
```

The keyword `movable` defines this as a move code question and the fragments are separated by lines of dashes (when the dashes are missing the code is automatically split in 2-line fragments). The result look as follows:

Question 5

Create a function to print squares by placing all code fragments in the grey box in the right order with the right indentation:

```
for(int i <- [1 .. n + 1])
}
println(i + " squared = " + (i * i));
println("Squares from 1 to " + n);
void squares(int n){
```

Check It

To make this kind of question more challenging, one or more "decoy" fragments can be defined that should not be used by the student:

question Create a function to print squares by placing all code fragments in the grey box in the right order with the right indentation (and avoid decoys!):

```
movable
void squares(int n){
-----
    println("Squares from 1 to " + n);
-----
    for(int i <- [1 .. n + 1])
-----
        println(i + " squared = " + (i * i));
-----
}
decoy
i = 0;
-----
i += 1;
-----
println(i + " squared = " + (i + i));
end
```

With the following result:

Question 6

Create a function to print squares by placing all code fragments in the grey box in the right order with the right indentation (and avoid decoys!):

```
void squares(int n){
i += 1;
}
println(i + " squared = " + (i * i));
println("Squares from 1 to " + n);
for(int i <- [1 .. n + 1])
i = 0;
println(i + " squared = " + (i + i));
```

Check It

Click-all-cases questions

A click-all-cases question presents a text to the student and the exercise is to click on all text fragment with a certain property, such as "is an identifier", "is a type", "is a bug" and the like. Each clickable area is marked with `$click(...)` in the question description:

```
question Click on all identifiers in this code fragment:
clickable
$click(x) = 1;
if($click(x)){
    $click(y) = $click(x) + 2;
}
end
```

This is how the question will look like:

Question 7

Click on all identifiers in this code fragment:

```
x = 1;
if(x){
    y = x + 2;
}
```

Check It

Reorder questions

A reorder question presents two columns of sentence fragments and the student is asked to reorder the fragments until they all form true statements. In the question description the true statements (starting with the keyword `fact`) are specified and they are presented to the student in random order.

```
question Reorder the following items and make all statements true:
fact [1,2,3] ||| is a list[int]
fact {1,2,3} ||| is a set[int]
fact 123 ||| is an int
fact "abc" ||| is a str
end
```

It is presented as follows:

Question 8

Reorder the following items and make all statements true:

<input type="text" value="123"/>	<input type="text" value="is a list[int]"/>
<input type="text" value="[1,2,3]"/>	<input type="text" value="is a str"/>
<input type="text" value="{1,2,3}"/>	<input type="text" value="is an int"/>
<input abc\""="" type="text" value="\"/>	<input type="text" value="is a set[int]"/>
<input type="button" value="Check It"/>	

Implementation

EDL has been implemented as part of Rascal's Tutor, an interactive documentation and teaching system. An EDL description is implemented using Rascal, HTML and Javascript: the EDL description is translated by a Rascal module to HTML+Javascript and exercise checking is done by callbacks to another component in Rascal. The version as described here is a technology preview and is not yet available in the standard Rascal distribution at the time of writing (but this may very well be the case at the time of publication).

Conclusions

The ideas presented here represent work-in-progress and no formal evaluation has been done to assess the effect of this style of exercises on learning. We can, however, go back to our original requirements and evaluate how well we have done:

	Description	Assessment
R1	The effort to describe an exercise should be small.	Achieved. EDL has been designed to provide a dense but readable question description.
R2	Exercises should be reused as much as possible. Possibly by randomly generating exercises from a given template.	Achieved. Random value generation is a built-in feature of EDL.
R3	Once a student has filled in an answer to an exercise, it can be automatically checked.	Achieved. All question categories have been designed with automatic checking in mind. Open questions, for instance, have not been included.
R4	When a student enters a good answer, it is rewarded with positive feedback.	Achieved but not described. Occasionally, a Zen proverb is presented as reward. ("No snowflake ever falls in the wrong place.")
R5	When a student enters a wrong answer, the error is explained and the student is encouraged to make a new attempt.	Achieved. A diagnosis of the wrong answer is given. Optionally, the exercise description can provide extra feedback. Occasionally, a Zen proverb is presented as encouragement. ("After climbing the hill, the view will be excellent.")
R6	Two categories of questions should be supported: knowledge questions and source code questions.	Achieved. This is achieved by the various question categories.
R7	To make exercises more entertaining, different styles of questions should be supported.	Achieved. This is already achieved by the various question categories but more categories can be considered to increase variation.

There are several systems around that offer interactive programming exercises. In most cases, the student is provided with a general editing pane to enter his/her answer. In my experience this is a major obstacle for beginning students since they are not aware of the syntax and typing rules of the language they are learning (obviously!). The question categories proposed here assume minimal knowledge and present canned source code fragments that only have to be completed or rearranged by the student thus avoiding many beginner's pitfalls and traps.

The ideas as presented here will be included in future releases of the Rascal language.

This paper is accompanied by a small screen cast that gives an impression how to answer the example questions discussed here.

Notes

¹. Although we will not illustrate this here, the full power of random testing is available while writing and running exercises: a test may have formal parameters and these will get random values assigned for each execution. By default, a random test is executed 500 times. ↩

². Even full random tests as observed before. ↩

From principles to practical pigeon protocols

Barteld Kooi and Rineke Verbrugge

Dramatis personae:

Jan van Eijck

Hans van Ditmarsch

Barteld Kooi

Rineke Verbrugge

Yanjing Wang

Setting: neomedieval Amsterdam. After the calls of Jan van Eijck to reduce carbon emissions and energy consumption, the world has gone back to using only sustainable energy and technology [7, 3, 6]. Air travel is only allowed with hot air balloons. Only sailing ships are allowed on the oceans. The Internet has collapsed. Scientists are struggling to maintain their international networks. Many logicians have flocked to Amsterdam to be able to continue collaborating.

Google still provides the fastest communication possible, because it has been converted into the largest carrier pigeon company in the world. The range of Google pigeons is limited and they often lose their way or are shot down with bows and arrows for consumption.

In the city center, Barteld Kooi and Yanjing Wang are working on a new logic with which PCP (Pigeon Communication Protocols) can be verified (cf. [8, 4]).

Yanjing: This is fascinating stuff. Who would have thought that such simple technology could lead to such deep logical insights. I wonder whether anyone has worked on this before? If only we could search the library quickly.

Barteld: Indeed! Last time I spoke to Jan and Rineke, they were heading to CWI where they are trying to get a computer to run. Perhaps they succeeded and can actually consult the library catalogue.

Yanjing: How would they get a computer to run? You need electricity for that and the only way to generate it is by hooking up a bicycle to a generator. Only very few people are prepared to bike fast and long enough to be able to use a computer.

Barteld: What I heard, is that they sent a pigeon to Nancy to invite Hans to make an important contribution to this project. He may have arrived. Let's send a pigeon to Jan and Rineke and ask whether Hans has arrived.

Yanjing: It's worth a try. I'll send them a pigeon.

Barteld: Good. But I propose that we use a protocol for sending pigeon messages, how about something like this—and let's send this protocol in our pigeon message too, together with our question about Hans:

Sender

```

i:=0;
while true do
  begin read  $x_i$ ;
    send  $x_i$ 
    until  $K_S K_R(x_i)$ ;
    send ‘‘ $K_S K_R(x_i)$ ’’
    until  $K_S K_R K_S K_R(x_i)$ ;
    i:=i+1
  end

```

Receiver

```
when  $K_R(x_0)$  set  $i := 0$ ;  
while true do  
  begin write( $x_i$ );  
    send ‘‘ $K_R(x_i)$ ’’  
    until  $K_R K_S K_R(x_i)$ ;  
    send ‘‘ $K_R K_S K_R(x_i)$ ’’  
    until  $K_R(x_{i+1})$ ;  
     $i := i + 1$ ;  
  end
```

Yanjing: Yes. That should do the trick, because the Google pigeon service comes with a guarantee of fairness and no other errors than deletions, and those only happen when pigeons get lost or shot.

Barteld: So, there it goes, our first pigeon message to Jan and Rineke. Godspeed!

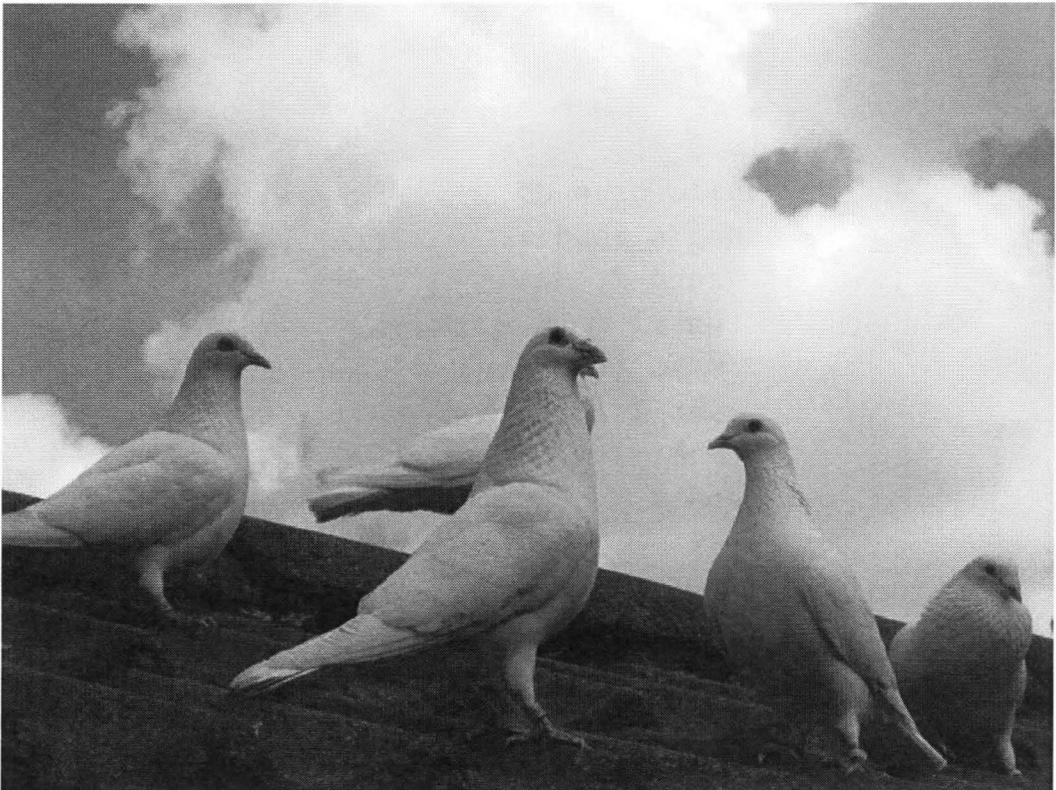


Figure 1: White carrier pigeons in The Netherlands

An hour later, at CWI:

Rineke: Look, Jan, a pigeon has just flown into your window. It has a message for us. Barteld and Yanjing ask us whether Hans has arrived yet, and they also propose an "original" pigeon communication protocol for us to use, which they dub PCP... Mmm, to me it looks suspiciously like good old protocol A by Halpern and Zuck [2], which I teach my students every year.

Jan: Barteld and Yanjing should know. These thirty- and forty-somethings have become so used to looking up everything on the Internet instead of remembering the literature, unlike us fifty- and sixty-somethings. I'm happy for my daughters that their generation is developing good memory and other skills again.

Rineke: A downside is that this PCP communication is really inefficient: it would take four messages just to get this 4-way handshake thing going.

Jan: Yes, and it's common knowledge that this protocol will never lead to common knowledge [5]. At each point in time, only $E^n \varphi$ for some $n \in \mathbf{N}, n \geq 1$ is achieved, where φ is "at least one message was delivered".

Rineke: I have a suggestion for our pigeon posts. For an infinitary epistemic proof system [1], we have $\{E^n \varphi \mid n \in \mathbf{N}, n \geq 1\} \vdash C\varphi$. Now we only need the pigeons to fly twice as fast in every new round as in the previous one; the first round takes one hour, the second round half an hour, and so on, ad infinitum. Summing the series, after two hours all the $\{E^n \phi \mid n \in \mathbf{N}, n \geq 1\}$ are achieved, whereby $C\phi$ holds.

Jan: If you want to try to make the pigeons do that, be my guest. But it doesn't sound sustainable to me.

Three quarters of an hour later, Hans arrives at the CWI on his bicycle, all the way from Nancy.

Hans: I was almost hit by a pigeon, flying about 300 miles an hour.

Rineke (looking guilty): Really?

Jan: I have a much better idea. let's all bike to the Vondelpark and meet Yanjing and Barteld there. Live. Hans, I hope you aren't too tired to bike some more?

Hans: Of course not! But first a cup of black coffee, please.

An hour later, all five researchers are sitting around in a circle in the Vondelpark. Some pigeons are sitting in their midst, looking exhausted.

Jan: Dear friends, now this is clearly the best way to create common knowledge, sitting in a circle, looking one another deep into the eyes... Who's in for some meditation practice?

References

- [1] Gerard Renardel de Lavalette, Barteld Kooi, and Rineke Verbrugge. A strongly complete proof system for propositional dynamic logic. In *AiML2002: Advances in Modal Logic*, pages 377–393, 2002.
- [2] Joseph Y Halpern and Lenore D Zuck. A little knowledge goes a long way: Knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, 1992.
- [3] David JC MacKay. *Sustainable Energy—Without the Hot Air*. UIT Cambridge, England, 2009.
- [4] Hans Van Ditmarsch, Sujata Ghosh, Rineke Verbrugge, and Yanjing Wang. Hidden protocols: Modifying our expectations in an evolving world. *Artificial Intelligence*, 208:18–40, 2014.
- [5] Hans van Ditmarsch, Jan van Eijck, and Rineke Verbrugge. Common knowledge and common belief. In *Discourses on Social Software*, volume 5 of *Texts in Logic and Games*, pages 99–122. Amsterdam University Press, Amsterdam, 2009.
- [6] Jan van Eijck. Over het kerst-essay van Rob Wijnberg, 2016. See <http://vaneijck.org/posts/2016-12-24-wijnberg.html>.
- [7] Jan van Eijck, Rohit Parikh, Marc Pauly, and Rineke Verbrugge. Social software and the ills of society. In *Discourses on Social Software*, volume 5 of *Texts in Logic and Games*, pages 219–226. Amsterdam University Press, Amsterdam, 2009.
- [8] Yanjing Wang, Lakshmanan Kuppusamy, and Jan van Eijck. Verifying epistemic protocols under common knowledge. In *Proceedings of the 12th Conference on Theoretical Aspects of Rationality and Knowledge*, pages 257–266. ACM, 2009.

Alain Badiou plays the game of Set

Bert Lisser

Push the button *Start Truth Procedure*. Drag some *symbols* into the *intuition* so that the intuition becomes a *truth*. Badiou calls such an action a *truth procedure*. After the intuition has become a truth the intuition lights up, you receive match points, and the accepted *set* is displayed in the history panel. After clicking on the intuition this will be empty again and a new *truth procedure* can be started. The purpose of this game is to obtain the highest possible score. The button *restart* starts a new game and *reset* resets the existing game.

In this case the set of symbols which transforms an intuition into a truth are the symbols placed on the three cards which form a set. (see *Game of Set*)).

Alain Badiou is a French contemporary philosopher. His key concept is *event*. Events arise in a *situation* (art, love, politics, or science). An *event* happens often unexpectedly and is in first instance unexplainable. An *event* must be first recognized by a *subject*. This *subject* have the choice of ignoring this *event* or accepting this *event*. In the later case a *truth procedure* will be started in the hope this *event* becomes a *truth*. For the definition of *truth* he turns to mathematics, specially set theory.

For simplicity the term *event* is here replaced by *intuition*, *situation* is replaced by *mind*, *factors* is replaced by *symbols*.

Dear Jan, success with your future work in philosophy and mathematics.

Interactive Reasoning (*paper version*)

Ștefan Minică

Abstract

The paper will present `DEMO_S5` interactive code together with its personal/historical context and code for visualizing interactive logical reasoning tasks in games.

From Prisoners and Lightbulbs to Epistemic Probability

My first memories of Jan go back to ...

```
RUNTIME ERROR: "MEMORY OVERFLOW"  
your program has run out of memory ...  
stack trace below:  
  undefined is not a function at (line 1, column 37)  
  
'This is an automatically generated message  
For further details please contact the maintainer of this package'
```

Sorry about that, ... Obviously, I should have used a `pure function`. Let me start over ...

My first memories of Jan go back to the years spent as a Ph.D. student ...

```
TIMEOUT ERROR: "THIS REQUEST HAS TIMED OUT"  
(moved permanently 301, redirecting ...)  
your program has run out of time ...  
stack trace below:  
  too much time has passed since the last request at (line 11, col 112)  
  
--This is an automatically generated message.  
--For further details please contact the maintainer of this package
```

Sorry again ... I might have used an `async call` ... Let me refresh and try again ...

My first memories of Jan go back to the years spent as a Ph.D. student at University of Amsterdam starting from 2007. ...

```
FORMAT ERROR: "YOUR CODE USES THE WRONG FORMAT"  
(unrecognized syntax ...)  
your program has encountered an unparseable token ...  
stack trace below:  
  You should never use code in a paper format  
  implementation is not meant for reading (line 24, column 45)  
  
--This is an automatically generated message  
--For further details please contact the maintainer of this package
```

Oh, I must have forgotten to compile the source. One more time ...

hits SHIFT + CTRL + C

My first memories of Jan go back to the years spent as a Ph.D. student at University of Amsterdam starting from 2007. I was just starting the work on my thesis on dynamic epistemic logics of questions (DELQ) ...

```
UNRECOVERABLE ERROR: "YOUR PROGRAM CAN NEVER BE RECOVERED ANYMORE"  
your program has moved to a virtual location ...  
stack trace below:  
  This is NOT an error ...  
  It seems that a deliberate choice has occurred at (line 324, column 7).  
  Your code will probably run on the on-line version of the Festschrift  
  
--This is an automatically generated message  
--For further details please contact the maintainer of this package
```

I don't understand !? ... This must be a paper bug !? ...

I really hope the on-line version works ...

$16T^A P$: A Toy Tableaux Theorem Prover for 16-Valued Trilattice Logics

Reinhard Muskens

Introduction

Jan van Eijck and I go back a very long time. We have known each other since the 1970s, when we were both philosophy students in Groningen. The Groningen Philosophy Department was in a large house at Kraneweg at that time and I must have met Jan there in 1974 when I arrived, or a bit later—the mists of time do not reveal a first meeting. I don't think we ever followed the same courses in those early years—Jan had started earlier than I had and was in another phase of the program altogether—but we certainly had a lot of contact after Johan van Benthem had become a 'lector' at the institute and had started livening things up. Jan and I participated in reading groups organised by Johan and there was also a lot of political activity within the institute that we took part in. Jan was admired by all for his ability to write. He could explain things as no-one could. And he also wrote some biting satire under the pseudonym 'Hein Egel'. Sweet memories.

Contacts became less regular after I had moved to Amsterdam late in '79, but increased in frequency again ten years later when Jan accepted a position at CWI and I was still living in Amsterdam (but working in Tilburg as Jan's successor there...). More reading group meetings, now organised by Jan. And more sweet memories. I also remember a trip to Saarbrücken Jan and I made in his car. It was in the late 1990s and we were scheduled to talk about dynamic semantics there. For some reason I remember Jan's car as a 2cv, which is a near impossibility, but a low-hanging sun in the almost deserted Eupen-Malmédy area seems real.

Enough of the reminiscing already! Let us ask ourselves a question: Who were the people who created the intellectual atmosphere in the Netherlands in which people like Jan and I—people who combine interests in logic and computer science with an interest in philosophy and the humanities—could thrive? There are many who can be named, and our common teacher Johan van Benthem played a central role. But if we go back a bit further

in time another name also stands out—Evert Willem Beth. Beth stood at the cradle of the *Centrale Interfaculteit*, based on the idea that philosophy should be a kind of central hub between other disciplines. This took philosophy out of its all too narrow context of humanities only. Beth likewise created the Amsterdam *Instituut voor Grondslagenonderzoek en Filosofie der Exacte Wetenschappen*, ILLC's precursor (even in a formal sense). The Centrale Interfaculteit is no more, but ILLC is alive and kicking (or certainly seems to be from my short distance) and Beth's idea that philosophy can be a bridge between science and the humanities, to the advantage of both, may be under some threat but is far from dead.

Beth brings me to the real topic of the contribution I want to make to this festive Festschrift, which combines two interests of Jan's—coding (not in Haskell, alas) and Beth Tableaux. I have written a little toy theorem prover $16T^A P$ for 16-valued trilattice logics (Shramko & Wansing [9]). The theorem prover itself is at the following url:
<http://swish.swi-prolog.org/p/sixteentap.pl>.

$16T^A P$ is based on a calculus for 16-valued trilattice logics that is described in Muskens & Wintein [5]. I wrote it just for the heck of it, about two years ago, and have now brushed it up so that it will actually display trees and generate counterexamples to sequents if there are any.

In order to explain what the prover does and why I must say a few things about 16-valued trilattice logics and about our calculus, but I shall be quite informal, as all these things have been explained in detail and with great formal precision elsewhere. For more information on the logics one could do worse than to start with Wansing [11] or the original Shramko & Wansing [9]. An important precursor in a constructivist setting is Shramko et al. [8], while Shramko & Wansing [10] offers a full monograph.

In order to understand trilattice logics we must start with the well-known four-valued Belnap-Dunn logic (Belnap [2,3], Dunn [4]). This logic is based on the values $\mathbf{T} = \{1\}$ (true and not false), $\mathbf{F} = \{0\}$ (false and not true), $\mathbf{N} = \emptyset$ (neither true nor false), and $\mathbf{B} = \{0, 1\}$ (both true and false) and can be viewed as a generalisation of classical logic—Belnap and Dunn move from $\{0, 1\}$ with its usual ordering to $\mathcal{P}(\{0, 1\})$ with—as [2,3] stress in particular—two lattice orders (which actually form a *bilattice*). Shramko and Wansing (and Shramko, Dunn, and Takenaka before them) in fact repeat this move, going from the set of truth-values $\mathcal{P}(\{0, 1\}) = \{\mathbf{T}, \mathbf{F}, \mathbf{N}, \mathbf{B}\}$ to its power set $16 = \mathcal{P}(\{\mathbf{T}, \mathbf{F}, \mathbf{N}, \mathbf{B}\})$, now with *three* lattices (as we shall see). While the four-

valued logic is meant to model the reasoning of a computer that is fed potentially incomplete or conflicting information, the 16-valued logic that results models the reasoning of a central server that is connected to a network of such computers. Each computer in the network is a Belnap-Dunn reasoner on information that is potentially incomplete or inconsistent and the central server must now reason with the 16 possible outcomes of their reasoning.

Let us define the three lattice orders with the help of four auxiliary orderings. For each $x \in \{\mathbf{T}, \mathbf{F}, \mathbf{N}, \mathbf{B}\}$, we define the relation \leq_x on $\mathbf{16}$ by letting, for $a, b \in \mathbf{16}$,

$$a \leq_x b := x \in a \implies x \in b$$

Using this, and writing \geq_x for the converse of \leq_x ($x \in \{\mathbf{T}, \mathbf{F}, \mathbf{N}, \mathbf{B}\}$) we define the *truth* ordering \leq_t , the *nonfalsity* ordering \leq_f and the *information* ordering \leq_i as follows.

$$\begin{aligned} \leq_t &:= \leq_{\mathbf{B}} \cap \geq_{\mathbf{F}} \cap \leq_{\mathbf{T}} \cap \geq_{\mathbf{N}} \\ \leq_{\neg f} &:= \geq_{\mathbf{B}} \cap \geq_{\mathbf{F}} \cap \leq_{\mathbf{T}} \cap \leq_{\mathbf{N}} \\ \leq_{\neg i} &:= \leq_{\mathbf{B}} \cap \leq_{\mathbf{F}} \cap \leq_{\mathbf{T}} \cap \leq_{\mathbf{N}} \end{aligned}$$

Figure 1 depicts $\mathbf{16}$ with the \leq_t order (left) and with the \leq_f order (right; in both cases one actually needs to take the reflexive transitive closure of what is indicated with the arrows), while Figure 2 gives the information order \leq_i .

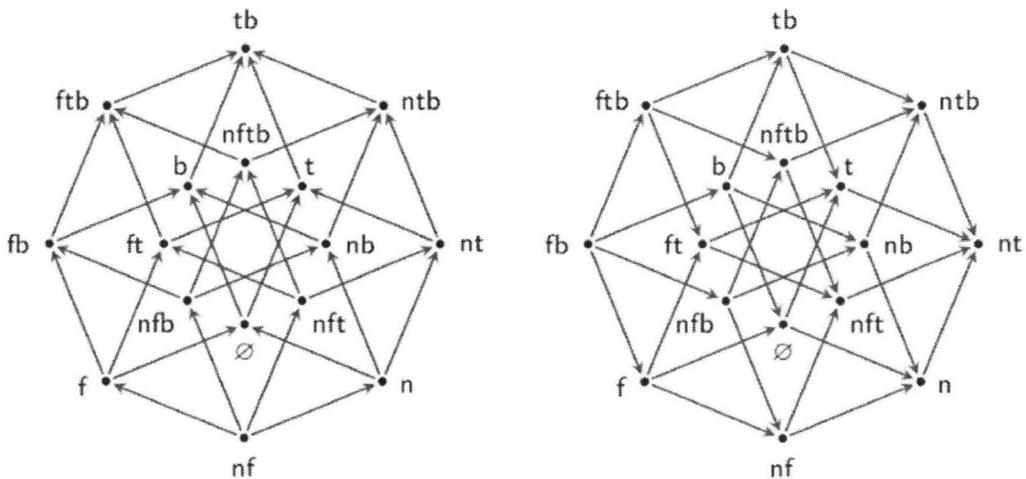


Figure 1 The trilattice SIXTEEN_3 with the truth order \leq_t (left) and with the nonfalsity order \leq_f (right). Vertices are accompanied by $\mathcal{L}_{t,fi}$ formulas denoting them. The top and bottom elements of \leq_t are \mathbf{tb} and \mathbf{nf} , while those of \leq_f are \mathbf{nt} and \mathbf{fb} .

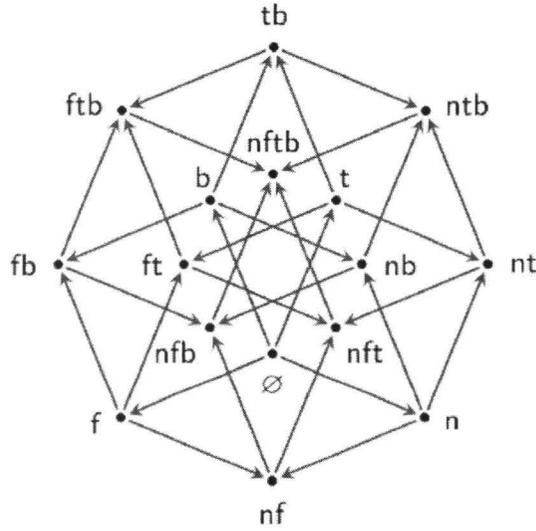


Figure 2 The trilattice $SIXTEEN_3$ with the information order \leq_i (top: **nftb**, bottom: \emptyset).

Each ordering \leq_k ($k \in \{t, f, i\}$) comes with obvious meet and join operations, denoted as \sqcap_k and \sqcup_k . (Clearly, \sqcap_i is \cap and \sqcup_i is \cup .) $SIXTEEN_3$ can now be defined as $\langle \mathbf{16}, \sqcap_t, \sqcup_t, \sqcap_f, \sqcup_f, \sqcap_i, \sqcup_i \rangle$.

It can be verified that, for any $\bullet, \circ \in \{\sqcap_t, \sqcup_t, \sqcap_f, \sqcup_f, \sqcap_i, \sqcup_i\}$, we have $a \circ (b \bullet c) = (a \circ b) \bullet (a \circ c)$ (see also Riviaccio [7]). This makes $SIXTEEN_3$ into a *distributive* trilattice, from which it can be shown to follow that the structure is *interlaced*, in the sense that all six lattice operations are monotone with respect to the three lattice orders.

It is also possible to add three unary operations $-_t$, $-_f$, and $-_i$ that can be taken to be the semantic correlates of negation connectives. We do this by letting, for any $a \in \mathbf{16}$:

$$\begin{array}{lll}
 \mathbf{T} \in -_t a \Leftrightarrow \mathbf{N} \in a; & \mathbf{T} \in -_f a \Leftrightarrow \mathbf{B} \in a; & \mathbf{T} \in -_i a \Leftrightarrow \mathbf{F} \notin a; \\
 \mathbf{B} \in -_t a \Leftrightarrow \mathbf{F} \in a; & \mathbf{B} \in -_f a \Leftrightarrow \mathbf{T} \in a; & \mathbf{B} \in -_i a \Leftrightarrow \mathbf{N} \notin a; \\
 \mathbf{F} \in -_t a \Leftrightarrow \mathbf{B} \in a; & \mathbf{F} \in -_f a \Leftrightarrow \mathbf{N} \in a; & \mathbf{F} \in -_i a \Leftrightarrow \mathbf{T} \notin a; \\
 \mathbf{N} \in -_t a \Leftrightarrow \mathbf{T} \in a; & \mathbf{N} \in -_f a \Leftrightarrow \mathbf{F} \in a; & \mathbf{N} \in -_i a \Leftrightarrow \mathbf{B} \notin a;
 \end{array}$$

Inspection will show that, for each pairwise distinct $k, \ell \in \{t, f, i\}$,

$$\begin{aligned}
&\text{if } a \leq_k b, \text{ then } \neg_k b \leq_k \neg_k a ; \\
&\text{if } a \leq_\ell b, \text{ then } \neg_k a \leq_\ell \neg_k b ; \\
&a = \neg_k \neg_k a .
\end{aligned}$$

(Again see Riviaccio [7].)

Having defined *SIXTEEN*₃ and the operations we need on it, we now turn to a propositional language that will be interpreted in this structure. The language \mathcal{L}_{tfi} is defined by the following BNF (where p comes from some countably infinite set of propositional constants).

$$\varphi ::= p \mid \sim_t \varphi \mid \sim_f \varphi \mid \sim_i \varphi \mid \varphi \wedge_t \varphi \mid \varphi \wedge_f \varphi \mid \varphi \wedge_i \varphi$$

This language now receives an entirely expected interpretation by letting a *valuation function* be a function V from the sentences of \mathcal{L}_{tfi} to **16** such that, for each $k \in \{t, f, i\}$, the following hold.

$$\begin{aligned}
V(\varphi \wedge_k \psi) &= V(\varphi) \sqcap_k V(\psi); \\
V(\sim_k \varphi) &= \neg_k V(\varphi).
\end{aligned}$$

If $\varphi \vee_k \psi$ is then defined as $\sim_k(\sim_k \varphi \wedge_k \sim_k \psi)$, one also gets

$$V(\varphi \vee_k \psi) = V(\varphi) \sqcup_k V(\psi).$$

Muskens & Wintein [5] show that \mathcal{L}_{tfi} is functionally complete. \mathcal{L}_{tfi} is more expressive than Shramko & Wansing's [9] original language \mathcal{L}_{tf} , which is based on $\{\wedge_t, \vee_t, \wedge_f, \vee_f, \sim_t, \sim_f\}$. It is also more expressive than the language $\mathcal{L}_{tf}^{\rightarrow_t}$ considered in Odintsov [6], in which an implication \rightarrow_t is added to \mathcal{L}_{tf} . But the calculus defined in [5], and hence our toy prover, works for all such sublanguages of the functionally complete language.

Let us define notions of entailment based on each of the three lattice orderings. Using \prod_k for greatest lower bound in the \leq_k order ($k \in \{t, f, i\}$) and \sqcup_k for least upper bound, let the relations \vDash_t , \vDash_f , and \vDash_i between sets of sentences Γ and Δ be defined in the following way.

$$\begin{array}{l}
\Gamma \models_t \Delta \iff \prod_{\gamma \in \Gamma} V(\gamma) \leq_t \prod_{\delta \in \Delta} V(\delta) \quad \text{for all valuations } V \\
\Gamma \models_f \Delta \iff \prod_{\gamma \in \Gamma} V(\gamma) \leq_f \prod_{\delta \in \Delta} V(\delta) \quad \text{for all valuations } V \\
\Gamma \models_i \Delta \iff \prod_{\gamma \in \Gamma} V(\gamma) \leq_i \prod_{\delta \in \Delta} V(\delta) \quad \text{for all valuations } V
\end{array}$$

We also let \models be the intersection $\models_t \cap \models_f$. Since \models_t is an entailment relation based on transmission of *truth* and \models_f is based on transmission of *nonfalsity*, this seems a natural notion.

$\frac{x : \varphi \wedge_t \psi}{x : \varphi, x : \psi} (\wedge_t^1)$ <p>where $x \in \{\bar{n}, \bar{f}, t, b\}$</p>	$\frac{x : \varphi \wedge_t \psi}{x : \varphi \mid x : \psi} (\wedge_t^2)$ <p>where $x \in \{n, f, \bar{t}, \bar{b}\}$</p>
$\frac{x : \varphi \wedge_f \psi}{x : \varphi, x : \psi} (\wedge_f^1)$ <p>where $x \in \{n, \bar{f}, t, \bar{b}\}$</p>	$\frac{x : \varphi \wedge_f \psi}{x : \varphi \mid x : \psi} (\wedge_f^2)$ <p>where $x \in \{\bar{n}, f, \bar{t}, b\}$</p>
$\frac{x : \varphi \wedge_i \psi}{x : \varphi, x : \psi} (\wedge_i^1)$ <p>where $x \in \{n, f, t, b\}$</p>	$\frac{x : \varphi \wedge_i \psi}{x : \varphi \mid x : \psi} (\wedge_i^2)$ <p>where $x \in \{\bar{n}, \bar{f}, \bar{t}, \bar{b}\}$</p>
$\frac{x : \sim_t \varphi}{y : \varphi} (\sim_t) \quad \text{where } \{x, y\} \in \{\{n, t\}, \{f, b\}, \{\bar{n}, \bar{t}\}, \{\bar{f}, \bar{b}\}\}$	
$\frac{x : \sim_f \varphi}{y : \varphi} (\sim_f) \quad \text{where } \{x, y\} \in \{\{n, f\}, \{t, b\}, \{\bar{n}, \bar{f}\}, \{\bar{t}, \bar{b}\}\}$	
$\frac{x : \sim_i \varphi}{y : \varphi} (\sim_i) \quad \text{where } \{x, y\} \in \{\{n, \bar{b}\}, \{f, \bar{t}\}, \{\bar{n}, b\}, \{\bar{f}, t\}\}$	

Table 1: Tableau expansion rules for PL16.

It is also expedient to have—auxiliary—entailment relations based on the relations \leq_x ($x \in \{\mathbf{T}, \mathbf{F}, \mathbf{N}, \mathbf{B}\}$) considered above. These we define as follows.

$\Gamma \models_x \Delta \iff \bigcap_{\gamma \in \Gamma} V(\gamma) \leq_x \bigcup_{\delta \in \Delta} V(\delta)$, for all valuations V

For each \models_x entailment relation defined in this way, write \models^x for its converse. It is not difficult to verify that the \models_t , \models_f , \models_i , and \models relations can be characterised in the following way.

$$\begin{aligned} \models_t &= \models_{\mathbf{B}} \cap \models^{\mathbf{F}} \cap \models_{\mathbf{T}} \cap \models^{\mathbf{N}} \\ \models_f &= \models^{\mathbf{B}} \cap \models^{\mathbf{F}} \cap \models_{\mathbf{T}} \cap \models_{\mathbf{N}} \\ \models_i &= \models_{\mathbf{B}} \cap \models_{\mathbf{F}} \cap \models_{\mathbf{T}} \cap \models_{\mathbf{N}} \\ \models &= \models_{\mathbf{B}} \cap \models^{\mathbf{B}} \cap \models^{\mathbf{F}} \cap \models_{\mathbf{T}} \cap \models_{\mathbf{N}} \cap \models^{\mathbf{N}} \end{aligned}$$

This means that a syntactic characterisation of the \models_x relations (where $x \in \{\mathbf{T}, \mathbf{F}, \mathbf{N}, \mathbf{B}\}$) will automatically provide us with a syntactic characterisation of \models_t , \models_f , \models_i , and \models .

This is where our tableau calculus—dubbed **PL16**—comes in. Entries in this calculus are *signed formulas* $x : \varphi$, where φ is an \mathcal{L}_{tfi} formula and x is one of the signs \mathbf{b} , \mathbf{f} , \mathbf{t} , \mathbf{n} , $\bar{\mathbf{b}}$, $\bar{\mathbf{f}}$, $\bar{\mathbf{t}}$, and $\bar{\mathbf{n}}$. The role of these signed sentences is purely formal, as far as the calculus is concerned, but they also have an intuitive meaning. $\mathbf{b} : \varphi$, for example, can be read as saying that \mathbf{B} is an element of the value of φ ; the meaning of $\bar{\mathbf{b}} : \varphi$ is that \mathbf{B} is not an element of the value of φ . The other signs are interpreted in an entirely similar fashion.

$$\frac{x : \varphi \vee_t \psi}{x : \varphi, x : \psi} (V_t^1)$$

where $x \in \{\mathbf{n}, \mathbf{f}, \bar{\mathbf{t}}, \bar{\mathbf{b}}\}$

$$\frac{x : \varphi \vee_f \psi}{x : \varphi, x : \psi} (V_f^1)$$

where $x \in \{\bar{\mathbf{n}}, \mathbf{f}, \bar{\mathbf{t}}, \mathbf{b}\}$

$$\frac{x : \varphi \vee_i \psi}{x : \varphi, x : \psi} (V_i^1)$$

where $x \in \{\bar{\mathbf{n}}, \bar{\mathbf{f}}, \bar{\mathbf{t}}, \bar{\mathbf{b}}\}$

$$\frac{x : \varphi \vee_t \psi}{x : \varphi \mid x : \psi} (V_t^2)$$

where $x \in \{\bar{\mathbf{n}}, \bar{\mathbf{f}}, \mathbf{t}, \mathbf{b}\}$

$$\frac{x : \varphi \vee_f \psi}{x : \varphi \mid x : \psi} (V_f^2)$$

where $x \in \{\mathbf{n}, \bar{\mathbf{f}}, \mathbf{t}, \bar{\mathbf{b}}\}$

$$\frac{x : \varphi \vee_i \psi}{x : \varphi \mid x : \psi} (V_i^2)$$

where $x \in \{\mathbf{n}, \mathbf{f}, \mathbf{t}, \mathbf{b}\}$

Table 2: Derived tableau expansion rules for disjunctions.

Table 2 gives the expansion rules of **PL16**, but only for the official \mathcal{L}_{tfi} connectives. Expansion rules for the three defined disjunctions are shown in Table [der]. Note that each of the conjunctions (disjunctions) comes with two rule schemas while each of the negations comes with one, and that each of these schemas is similar to a rule in a standard formulation of a signed tableau calculus for classical propositional logic. Only the side conditions are different.

The procedure to obtain tableaux from these expansion rules, given an initial set of signed sentences, is entirely standard. A tableau branch is *closed* if it contains signed sentences $x : \varphi$ and $\bar{x} : \varphi$ for $x \in \{\mathbf{n}, \mathbf{f}, \mathbf{t}, \mathbf{b}\}$. A branch that is not closed is called *open*. And a tableau is *closed* if all its branches are closed.

We can now characterise the four auxiliary entailment relations. Let Γ and Δ be finite sets of \mathcal{L}_{tfi} sentences. Then

$$\begin{aligned} \Gamma \models_{\mathbf{T}} \Delta &\iff \{\mathbf{t} : \varphi \mid \varphi \in \Gamma\} \cup \{\bar{\mathbf{t}} : \varphi \mid \varphi \in \Delta\} \text{ has a closed tableau;} \\ \Gamma \models_{\mathbf{F}} \Delta &\iff \{\mathbf{f} : \varphi \mid \varphi \in \Gamma\} \cup \{\bar{\mathbf{f}} : \varphi \mid \varphi \in \Delta\} \text{ has a closed tableau;} \\ \Gamma \models_{\mathbf{N}} \Delta &\iff \{\mathbf{n} : \varphi \mid \varphi \in \Gamma\} \cup \{\bar{\mathbf{n}} : \varphi \mid \varphi \in \Delta\} \text{ has a closed tableau;} \\ \Gamma \models_{\mathbf{B}} \Delta &\iff \{\mathbf{b} : \varphi \mid \varphi \in \Gamma\} \cup \{\bar{\mathbf{b}} : \varphi \mid \varphi \in \Delta\} \text{ has a closed tableau.} \end{aligned}$$

(continued on next page)

16^{TAP}: A Toy Tableaux Theorem Prover for 16-Valued Trilattice Logics

Reinhard Muskens

(continued from previous page)

This requires a proof, of course, but the proof is straightforward and given in [5]. Note that a *single* calculus is used to characterise each of these four entailment relations.

It can now be concluded that if we want to check whether any of the \models_t , \models_f , \models_i , or \models relations holds between given Γ and Δ we can do that by developing several tableaux. For example, if we want to verify $\Gamma \models \Delta$ we can do it by making tableaux for $\Gamma \models_{\mathbf{B}} \Delta$, $\Delta \models_{\mathbf{B}} \Gamma$, $\Delta \models_{\mathbf{F}} \Gamma$, $\Gamma \models_{\mathbf{T}} \Delta$, $\Gamma \models_{\mathbf{N}} \Delta$, and $\Delta \models_{\mathbf{N}} \Gamma$. If all these close then $\Gamma \models \Delta$ holds, otherwise there is a counterexample. The relations \models_t , \models_f , and \models_i each require four tableaux. (In general, not always in practice. It can be shown, for example, that if \wedge_i does not occur in any of the formulas, we need only two tableaux in order to check \models_t and \models_f relations.)

Working out all these tableaux by hand is no real fun of course and here is meant to come to the rescue. The prover is loosely based on the propositional part of Beckert & Posegga's [LeanTaP]^{LeanTaP}, but is considerably less lean. On the other hand, it was programmed to give some feedback to the user, so that it might conceivably give them some insight in what is and what is not valid in 16-valued trilattice logics (and why). In particular, it will print out any counterexample it may find and it will graphically display the tableaux it constructs (with closed branches marked \times and open ones marked \circ).

Any query about the \models_t , \models_f , \models_i , or \models relations will be translated into several subgoals that relate to the four auxiliary relations. These are then each satisfied by signing the formulas in question and developing a tableau.

The prover is programmed in SWISH (SWI-Prolog for SHaring), which implements a subset of SWI-Prolog, but comes with a nice interface and a tree renderer (svgtree—originally meant for rendering syntax trees). The following predicates are meant to be used in queries.

```
entailst(+Premises,+Conclusions,-T1,-T2,-T3,-T4)
```

This predicate implements \models_t (or rather its syntactic equivalent \vdash_t). Premises and Conclusions are lists of formulas, and T1, T2, T3, and T4 are the tableaux that are returned.

```
entailsf(+Premises,+Conclusions,-T1,-T2,-T3,-T4)
```

```
entailsi(+Premises,+Conclusions,-T1,-T2,-T3,-T4)
```

Similar to `entailst/6`, but now \models_f and \models_i are modeled.

```
entails(+Premises,+Conclusions,-T1,-T2,-T3,-T4,-T5,-T6)
```

The \models relation. Six trees are returned.

```
equiv(+Formula1,+Formula2,-T1,-T2,-T3,-T4,-T5,-T6,-T7,-T8)
```

The predicate tests whether the formulas `Formula1` and `Formula2` are logically equivalent by developing eight trees.

```
thmt(+Formula,-T1,-T2,-T3,-T4)
```

```
thmf(+Formula,-T1,-T2,-T3,-T4)
```

```
thmi(+Formula,-T1,-T2,-T3,-T4)
```

```
thm(+Formula,-T1,-T2,-T3,-T4,-T5,-T6)
```

`thmt` is a version of `entailst/6` with an empty set of premises while the three other predicates bear the same relation to `entailsf/6`, `entailsi/6`, and `entails/8`, respectively.

Formulas can be entered with binary connectives as infix operators, as usual, but I have not been able to persuade SWISH to also render their occurrences in trees in this way, so we get their internal Prolog representation there. An overview of the way connectives are represented as Prolog operators is given in Table 3.

$\wedge_t : \mathbf{at}$	$\wedge_f : \mathbf{af}$	$\wedge_i : \mathbf{ai}$
$\vee_t : \mathbf{ot}$	$\vee_f : \mathbf{of}$	$\vee_i : \mathbf{oi}$
$\sim_t : \mathbf{nt}$	$\sim_f : \mathbf{nf}$	$\sim_i : \mathbf{ni}$
$\rightarrow_t : \mathbf{it}$	$\rightarrow_f : \mathbf{if}$	$\rightarrow_i : \mathbf{ii}$
$\neg : \mathbf{neg}$		

Table 3: Connectives and the Prolog operators representing them.

It was a joy to write $16T^A P$. I hope that Jan and others find pleasure in playing with it.

References

- [1] B. Beckert and J. Posegga. leanTAP : Lean $T^A P$: Tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.
- [2] N. D. Belnap. How a Computer Should Think. In G. Ryle, editor, *Contemporary Aspects of Philosophy*, pages 30–56. Oriel Press, Stocksfield, 1976.
- [3] N. D. Belnap. A Useful Four-Valued logic. In J.M. Dunn and G. Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 8–37. Reidel, Dordrecht, 1977.
- [4] J.M. Dunn. Intuitive Semantics for First-Degree Entailments and 'Coupled Trees'. *Philosophical Studies*, 29:149–168, 1976.
- [5] R. Muskens and S. Wintein. Analytic Tableaux for all of *SIXTEEN*₃. *Journal of Philosophical Logic*, 44(5):473–487, 2015.
- [6] S. Odintsov. On Axiomatizing Shramko-Wansing's Logic. *Studia Logica*, 91:407–428, 2009.
- [7] U. Rivieccio. Representation of Interlaced Trilattices. *Journal of Applied Logic*, 11:174–189, 2013.
- [8] Y. Shramko, M. Dunn, and T. Takenaka. The Trilattice of Constructive Truth Values. *Journal of Logic and Computation*, 11(6):761–788, 2001.
- [9] Y. Shramko and H. Wansing. Some Useful 16-Valued Logics: How a Computer Network Should Think. *Journal of Philosophical Logic*, 34:121–153, 2005.

[10] Y. Shramko and H. Wansing. Truth and Falsehood: An Inquiry into Generalized Logical Values, volume 36 of *Trends in Logic*. Springer, 2011.

[11] H. Wansing. A Non-Inferentialist, Anti-Realistic Conception of Logical Truth and Falsity. *Topoi*, 31:93–100, 2012.

Static code

```
/*
```

```
16TaP---Tableaux for 16-valued Trilattice Logic  
Reinhard Muskens
```

```
16TaP is a simple propositional tableau prover/model generator for a  
functionally complete extension of the 16-valued Shramko-Wansing  
logic. The prover implements the tableau calculus  $PL_{\{16\}}$  defined in  
Reinhard Muskens and Stefan Wintein, Analytic Tableaux for all of  
SIXTEEN_3, Journal of Philosophical Logic 44 (5):473-487 (2015)  
(http://link.springer.com/article/10.1007/s10992-014-9337-3). It is  
loosely based on Beckert and Posegga's LeanTaP prover for predicate  
logic, but unlike that little marvel it is not very lean.
```

```
The calculus is based on signed formulas and there are eight signs,  
which we represent here as, n1, f1, t1, b1, n0, f0, t0, and  
b0. Intuitively, 'n1:F' means 'n is an element of the value of F' and  
'n0:F' stands for 'n is not an element of the value of F'. A similar  
interpretation can be given to the other signs.
```

```
There are several notions of entailment that are modelled by the  
calculus, and in fact the notions of entailment that are of most  
interest, for example those corresponding to one of the orderings in  
the trilattice SIXTEEN_3, can be obtained as the intersections of  
certain sets of auxiliary entailment relations that the calculus can  
characterize. Checking whether a sequence is correct often involves  
making several tableaux---four if an entailment relation based on one  
of the orderings in the trilattice is considered, and six when a  
natural notion of entailment based on two of these orderings is in  
focus.
```

```
The calculus is based on a functionally complete set of connectives,  
which we write here as {nt, nf, ni, at, af, ai, ot, of, oi}. The  
connectives whose names start with 'n' are negations; one for the  
truth lattice ('t'), one for the (non-)falsity lattice ('f'), and one
```

for the information lattice ('i'). The other connectives are conjunctions (names starting with 'a') and disjunctions ('o').

Unlike LeanTaP, the prover does not presuppose that formulas are in negation normal form. Counterexamples to an argument are given if they exist and graphical representations of the tableaux that are generated are given. If a tableau branch is closed, this is marked with an 'x'; open branches are marked with an 'o'.

Speed of execution or compactness of code have not been primary aims in writing this prover/generator. The basic aims of the program are demonstration of the calculus and providing a tool for further analysis of the logic.

As in Beckert and Posegga's LeanTap, the prover searches through the branches of a tableau in Prolog's left-right depth-first manner.

*/

```
:- use_rendering(svgtree).
```

/*

Operators

*/

% connectives we take to be primitives

```
:- op(400, fy, nt).    %negation in t lattice
:- op(400, fy, nf).    %negation in f lattice
:- op(400, fy, ni).    %negation in i lattice
```

```
:- op(500, xfy, at).   %conjunction in t lattice
:- op(500, xfy, af).   %conjunction in f lattice
:- op(500, xfy, ai).   %conjunction in i lattice
```

```
:- op(600, xfy, ot).   %disjunction in t lattice
:- op(600, xfy, of).   %disjunction in f lattice
:- op(600, xfy, oi).   %disjunction in i lattice
```

% connectives defined from the primitives

```
:- op(400, fy, neg).   %Odintsov's classical negation
:- op(650, xfy, it).   %Odintsov's t implication
:- op(650, xfy, if).   %Odintsov's f implication
:- op(650, xfy, ii).   %an analogous i implication
```

continued

```
/*  
Definitions of the defined connectives (roll your own).  
define(Definiendum,Definiens) will allow Definiendum to  
be rewritten as Definiens.  
*/
```

```
define(neg F,nt nf ni F).  
define(F1 it F2,neg F1 ot F2).  
define(F1 if F2,neg F1 of F2).  
define(F1 ii F2,neg F1 oi F2).
```

```
/*  
The calculus is based on eight signs, which we represent here as  
't1', 'f1', 'n1', 'b1', 't0', 'f0', 'n0',  
and 'b0'. The predicate opposite/2 defines an obvious notion  
of incompatibility. Branches close if they contain signed formulas  
S1:F and S2:F, with S1 and S2 opposite signs.  
*/
```

```
opposite(t1,t0).  
opposite(t0,t1).  
opposite(f1,f0).  
opposite(f0,f1).  
opposite(n1,n0).  

```

```
/*  
Next we give the predicate tableau/4, which is the heart of the  
program. tableau(Signedformulas,Atoms,S,Trees) develops a tableau from  
a tableau branch containing the signed formulas in Signedformulas  
and the signed atoms in Atoms. If a model is found it will be described  
(in the form of a list of signed atoms). If no model is found the set  
of formulas in Signedformulas + Atoms is inconsistent. The S argument  
contains a sign, while the Trees argument is used to build a graphical  
representation of the tableau. It is a list of (one or two) terms that  
each represent a subtree of the tableau directly dominated by the root.
```

There will be rules for primitive connectives, for defined connectives,
for closure, and for writing out sets of signed atoms.

Rules of the first kind take the first element of the Signedformulas list
and decompose it according to the tableau expansion rules. Bottom
formulas of a rule are added as roots to subtrees of the term

```
under construction.
```

```
*/
```

```
/*
```

```
Negation rules
```

```
(Side conditions follow.)
```

```
*/
```

```
tableau([Sign:nt F|Rest],Atoms,S,[Tree]) :-
    ntcond(Sign,Sign1), !,
    tableau([Sign1:F|Rest],Atoms,S,Subtrees),
    addroot(Sign1:F,Subtrees,Tree).
tableau([Sign:nf F|Rest],Atoms,S,[Tree]) :-
    nfcond(Sign,Sign1), !,
    tableau([Sign1:F|Rest],Atoms,S,Subtrees),
    addroot(Sign1:F,Subtrees,Tree).
tableau([Sign:ni F|Rest],Atoms,S,[Tree]) :-
    nicond(Sign,Sign1), !,
    tableau([Sign1:F|Rest],Atoms,S,Subtrees),
    addroot(Sign1:F,Subtrees,Tree).
```

```
/*
```

```
Conjunction rules
```

```
*/
```

```
tableau([Sign:(F1 at F2)|Rest],Atoms,S,[Tree]) :-
    member(Sign,[n0,f0,t1,b1]), !,
    tableau([Sign:F1,Sign:F2|Rest],Atoms,S,Subtrees),
    addroot(Sign:F2,Subtrees,Tree1),
    addroot(Sign:F1,[Tree1],Tree).
tableau([Sign:(F1 at F2)|Rest],Atoms,S,[Tree1,Tree2]) :-
    member(Sign,[n1,f1,t0,b0]), !,
    tableau([Sign:F1|Rest],Atoms,S,Subtrees1),
    tableau([Sign:F2|Rest],Atoms,S,Subtrees2),
    addroot(Sign:F1,Subtrees1,Tree1),
    addroot(Sign:F2,Subtrees2,Tree2).

tableau([Sign:(F1 af F2)|Rest],Atoms,S,[Tree]) :-
    member(Sign,[n1,f0,t1,b0]), !,
    tableau([Sign:F1,Sign:F2|Rest],Atoms,S,Subtrees),
    addroot(Sign:F2,Subtrees,Tree1),
    addroot(Sign:F1,[Tree1],Tree).
tableau([Sign:(F1 af F2)|Rest],Atoms,S,[Tree1,Tree2]) :-
    member(Sign,[n0,f1,t0,b1]), !,
    tableau([Sign:F1|Rest],Atoms,S,Subtrees1),
    tableau([Sign:F2|Rest],Atoms,S,Subtrees2),
```

```

    addroot(Sign:F1,Subtrees1,Tree1),
    addroot(Sign:F2,Subtrees2,Tree2).

tableau([Sign:(F1 ai F2)|Rest],Atoms,S,[Tree]) :-
    member(Sign,[n1,f1,t1,b1]), !,
    tableau([Sign:F1,Sign:F2|Rest],Atoms,S,Subtrees),
    addroot(Sign:F2,Subtrees,Tree1),
    addroot(Sign:F1,[Tree1],Tree).
tableau([Sign:(F1 ai F2)|Rest],Atoms,S,[Tree1,Tree2]) :-
    member(Sign,[n0,f0,t0,b0]), !,
    tableau([Sign:F1|Rest],Atoms,S,Subtrees1),
    tableau([Sign:F2|Rest],Atoms,S,Subtrees2),
    addroot(Sign:F1,Subtrees1,Tree1),
    addroot(Sign:F2,Subtrees2,Tree2).

/*
Disjunction rules
*/
tableau([Sign:(F1 ot F2)|Rest],Atoms,S,[Tree]) :-
    member(Sign,[n1,f1,t0,b0]), !,
    tableau([Sign:F1,Sign:F2|Rest],Atoms,S,Subtrees),
    addroot(Sign:F2,Subtrees,Tree1),
    addroot(Sign:F1,[Tree1],Tree).
tableau([Sign:(F1 ot F2)|Rest],Atoms,S,[Tree1,Tree2]) :-
    member(Sign,[n0,f0,t1,b1]), !,
    tableau([Sign:F1|Rest],Atoms,S,Subtrees1),
    tableau([Sign:F2|Rest],Atoms,S,Subtrees2),
    addroot(Sign:F1,Subtrees1,Tree1),
    addroot(Sign:F2,Subtrees2,Tree2).

tableau([Sign:(F1 of F2)|Rest],Atoms,S,[Tree]) :-
    member(Sign,[n0,f1,t0,b1]), !,
    tableau([Sign:F1,Sign:F2|Rest],Atoms,S,Subtrees),
    addroot(Sign:F2,Subtrees,Tree1),
    addroot(Sign:F1,[Tree1],Tree).
tableau([Sign:(F1 of F2)|Rest],Atoms,S,[Tree1,Tree2]) :-
    member(Sign,[n1,f0,t1,b0]), !,
    tableau([Sign:F1|Rest],Atoms,S,Subtrees1),
    tableau([Sign:F2|Rest],Atoms,S,Subtrees2),
    addroot(Sign:F1,Subtrees1,Tree1),
    addroot(Sign:F2,Subtrees2,Tree2).

tableau([Sign:(F1 oi F2)|Rest],Atoms,S,[Tree]) :-
    member(Sign,[n0,f0,t0,b0]), !,
    tableau([Sign:F1,Sign:F2|Rest],Atoms,S,Subtrees),
    addroot(Sign:F2,Subtrees,Tree1),

```

```

    addroot(Sign:F1, [Tree1], Tree).
tableau([Sign:(F1 oi F2)|Rest], Atoms, S, [Tree1, Tree2]) :-
    member(Sign, [n1, f1, t1, b1]), !,
    tableau([Sign:F1|Rest], Atoms, S, Subtrees1),
    tableau([Sign:F2|Rest], Atoms, S, Subtrees2),
    addroot(Sign:F1, Subtrees1, Tree1),
    addroot(Sign:F2, Subtrees2, Tree2).

```

```

/*
Rule for defined connectives
*/

```

```

tableau([Sign:F|Rest], Atoms, S, [Tree]) :-
    define(F, F1), !,
    tableau([Sign:F1|Rest], Atoms, S, Subtrees),
    addroot(Sign:F1, Subtrees, Tree).

```

```

/*
Closure rule
At this point any first element of the Signedformulas list is
assured to be a signed atom. If its opposite is on the Atoms
list we can close the branch.
*/

```

```

tableau([Sign:Atom|_], Atoms, _, [X]) :-
    opposite(Sign, Sign1),
    member(Sign1:Atom, Atoms), !.

```

```

/*
Otherwise, we add the signed atom to the list of atoms already
found and continue.
*/

```

```

tableau([Sign:Atom|Rest], Atoms, S, Trees) :- !,
    tableau(Rest, [Sign:Atom|Atoms], S, Trees).

```

```

/*
If all signed formulas have been reduced to signed atoms and there
is no conflicting pair of signed atoms, we have a model, which is
a counterexample if we are testing for validity.
*/

```

continued

```
tableau([],Atoms,S,[o]) :- !,
    write('counterexample:'), nl,
    write(Atoms), nl, write('no transmission of '),
    write(S), nl, nl.

/*
Side conditions on the negation rules
*/

ntcond(n1,t1).
ntcond(t1,n1).
ntcond(f1,b1).
ntcond(b1,f1).
ntcond(n0,t0).
ntcond(t0,n0).
ntcond(f0,b0).
ntcond(b0,f0).

nfcond(n1,f1).
nfcond(f1,n1).
nfcond(t1,b1).
nfcond(b1,t1).
nfcond(n0,f0).
nfcond(f0,n0).
nfcond(t0,b0).
nfcond(b0,t0).

nicond(n1,b0).
nicond(b0,n1).
nicond(f1,t0).
nicond(t0,f1).
nicond(n0,b1).
nicond(b1,n0).
nicond(f0,t1).
nicond(t1,f0).

/*
Entailment predicates, equivalence, and theorems.
*/

entailsaux(Sign, Premises, Conclusions,Tree) :-
    sign(Premises,Conclusions,Sign,SignedFormulas), !,
    tableau(SignedFormulas,[],Sign,Trees),
    initialbranch(SignedFormulas,Trees,[Tree]).
```

```

entailst(Premises, Conclusions, Tree1, Tree2, Tree3, Tree4) :-
    entailsaux(t1, Premises, Conclusions, Tree1),
    entailsaux(b1, Premises, Conclusions, Tree2),
    entailsaux(f0, Premises, Conclusions, Tree3),
    entailsaux(n0, Premises, Conclusions, Tree4).

entailsf(Premises, Conclusions, Tree1, Tree2, Tree3, Tree4) :-
    entailsaux(t1, Premises, Conclusions, Tree1),
    entailsaux(b0, Premises, Conclusions, Tree2),
    entailsaux(f0, Premises, Conclusions, Tree3),
    entailsaux(n1, Premises, Conclusions, Tree4).

entailsi(Premises, Conclusions, Tree1, Tree2, Tree3, Tree4) :-
    entailsaux(t1, Premises, Conclusions, Tree1),
    entailsaux(b1, Premises, Conclusions, Tree2),
    entailsaux(f1, Premises, Conclusions, Tree3),
    entailsaux(n1, Premises, Conclusions, Tree4).

entails(Premises, Conclusions, Tree1, Tree2, Tree3, Tree4, Tree5, Tree6) :-
    entailsaux(t1, Premises, Conclusions, Tree1),
    entailsaux(b1, Premises, Conclusions, Tree2),
    entailsaux(b0, Premises, Conclusions, Tree3),
    entailsaux(f0, Premises, Conclusions, Tree4),
    entailsaux(n1, Premises, Conclusions, Tree5),
    entailsaux(n0, Premises, Conclusions, Tree6).

equiv(Formula1, Formula2, Tree1, Tree2, Tree3, Tree4, Tree5, Tree6, Tree7, Tree8)
:-
    entailsaux(t1, [Formula1], [Formula2], Tree1),
    entailsaux(b1, [Formula1], [Formula2], Tree2),
    entailsaux(f1, [Formula1], [Formula2], Tree3),
    entailsaux(n1, [Formula1], [Formula2], Tree4),
    entailsaux(t0, [Formula1], [Formula2], Tree5),
    entailsaux(b0, [Formula1], [Formula2], Tree6),
    entailsaux(f0, [Formula1], [Formula2], Tree7),
    entailsaux(n0, [Formula1], [Formula2], Tree8).

thmt(Formula, Tree1, Tree2, Tree3, Tree4) :-
    entailst([], [Formula], Tree1, Tree2, Tree3, Tree4).
thmf(Formula, Tree1, Tree2, Tree3, Tree4) :-
    entailsf([], [Formula], Tree1, Tree2, Tree3, Tree4).
thmi(Formula, Tree1, Tree2, Tree3, Tree4) :-
    entailsi([], [Formula], Tree1, Tree2, Tree3, Tree4).
thm(Formula, Tree1, Tree2, Tree3, Tree4, Tree5, Tree6) :-
    entails([], [Formula], Tree1, Tree2, Tree3, Tree4, Tree5, Tree6).

```

```

/*
Some predicates to do with the term representation of the
tree. addroot/3 takes a collection of trees (terms) and
adds a root (functor), so that the result is a single tree.
initialbranch/3 uses this to provide a collection of trees
with a common initial branch.
*/

```

```

addroot(Signedformula, Trees, Tree) :-
    term_to_atom(Signedformula, Quoted),
    Tree =.. [Quoted|Trees].

```

```

initialbranch([], Trees, Trees).

```

```

initialbranch([Signedformula|Rest], Trees, [Tree]) :-
    initialbranch(Rest, Trees, Trees1),
    addroot(Signedformula, Trees1, Tree).

```

```

/*
sign(Premises, Conclusions, Sign, SignedFormulas) signs
the formulas in Premises with Sign and those in Conclusions
with its opposite. The signed formulas are collected in
SignedFormulas.
*/

```

```

sign([], [], _, []).
sign([], [F|Rest], Sign, [Sign2:F|SignedRest]) :-
    opposite(Sign, Sign2), sign([], Rest, Sign, SignedRest).
sign([F|Rest], Conclusions, Sign, [Sign:F|SignedRest]) :-
    sign(Rest, Conclusions, Sign, SignedRest).

```

```

/** <examples>

```

```

Some valid, some not.

```

```

?- entails([p at q], [p ot q], Tree1, Tree2, Tree3, Tree4, Tree5, Tree6).
?- entailst([p at q], [p ot q], Tree1, Tree2, Tree3, Tree4).
?- entails([p at q], [p], Tree1, Tree2, Tree3, Tree4, Tree5, Tree6).
?- entailst([p at q], [p], Tree1, Tree2, Tree3, Tree4).
?- entails([p at q], [p of q], Tree1, Tree2, Tree3, Tree4, Tree5, Tree6).
?- entailst([p, p it q], [q], Tree1, Tree2, Tree3, Tree4).
?- equiv(p at (q of r oi s), (p at q) of (p at r) oi (p at s), Tree1, Tree2,
Tree3, Tree4, Tree5, Tree6, Tree7, Tree8).
?- equiv(p at q af r ai s, s at r af q ai p, Tree1, Tree2, Tree3, Tree4, Tree5,
Tree6, Tree7, Tree8).

```

```

?- equiv(neg (p at q af r ai s), neg p ot neg q of neg r oi neg s, Tree1, Tree2, Tree3, Tree4, Tree5, Tree6, Tree7, Tree8).
?- equiv(nf (p at (q af (r ai s))), nf p at (nf q of (nf r ai nf s)), Tree1, Tree2, Tree3, Tree4, Tree5, Tree6, Tree7, Tree8).
?- thmt((p it (q it r)) it ((p it q) it (p it r)), Tree1, Tree2, Tree3, Tree4).
?- thm((p it (q it r)) it ((p it q) it (p it r)), Tree1, Tree2, Tree3, Tree4, Tree5, Tree6).
*/

```

For further details see: Analytic Tableaux for all of SIXTEEN₃

Nothing to add

Rick Nouwen

Context

Jan van Eijck taught me to be precise. As with models in any discipline, semantic analyses only make sense if we have a detailed understanding of how they work and what their predictions are. Jan often insisted on the value of implementation for this purpose: it allows the exploration and rigorous testing of an analysis, and it creates a fuller appreciation of the complexity of a proposed system. He was ultimately unsuccessful in convincing me that my dissertation should contain a Haskell implementation of the system I was proposing, but we did collaborate on a joint paper in which we presented a semantic analysis of plural discourse anaphora alongside Haskell code implementing that analysis (van Eijck and Nouwen 2002). In this brief note, I want to reconsider one of the design choices we made in that paper.

No doubt the most important line of code in our analysis is the line that determines the view taken on what exactly a context is.

```
type Context = ...
```

It is clear what a context is *for*, that it has a double function in interpretation. On the one hand it provides the necessary background that makes interpretation possible. On the other hand, it is being manipulated by interpretation: by interpreting natural language, we are *changing* the context. To determine what a context *is*, we then need to ascertain what it is that interpretation potentially depends and impacts on.

Doing this, a dual view on context emerges. When you read or listen, you don't just gain potential new beliefs, you also create a memory of the topics and things that are being discussed. In other words, by interpreting natural language we keep track of (i) the topics of conversation and (ii) the information that is being shared on those topics. This way of looking at *interpretation* is the central philosophy behind *dynamic semantics* (Kamp 1981,

Groenendijk and Stokhof 1991, van Eijck 2001, Nouwen et al. 2016), a family of theories of how ordinary semantics can be extended to take into account the basic dynamics of information processing.

In line with this view, Jan van Eijck's Incremental Dynamics (van Eijck 2001) takes a *context* to be a stack of entities (the list of entities that are discussed, if you want) and interpretation is a function from such stacks to sets of such stacks (those sets that comply with the given information). Different positions in a stack correspond to different *discourse referents*, the entities under discussion. Introducing a new referent in the discourse amounts to simply adding an entity to the context.¹

$$\llbracket \exists \rrbracket(c) = \{c \wedge d \mid d \in D\}$$

There are no ordinary variables, only indices, corresponding to locations in the context: $c[n]$ returns the entity stored on the n th position in c . Predication tests are defined as follows:²

$$\llbracket P(n) \rrbracket(c) = \begin{cases} \{c\} & c[n] \in I(P) \\ \emptyset & \text{otherwise} \end{cases}$$

Dynamic actions are combined using $\llbracket \varphi; \psi \rrbracket(c) = \{\llbracket \psi \rrbracket c' \mid c' \in \llbracket \varphi \rrbracket(c)\}$. And, so, the (in)famous *A man came in. He sighed.* gets the following analysis in Incremental Dynamics, where i is the length of the input context.

$$\exists; \text{man}(i); \text{sighed}(i)$$

One major advantage of this view on context is that it is absolutely explicit about what the anaphoric potential of the context is: the context *is* the set of accessible discourse referents. As we will see next, this has an interesting consequence once we add quantifiers and plurals to the system.

Context and Quantification

In van Eijck and Nouwen (2002), generalised quantifiers work quite similarly to \exists in that they push entities to the stack. This is different from the traditional take on quantifiers in the dynamic semantic literature, most notably different from Kamp (1981), Kamp and

Reyle (1993) and Groenendijk and Stokhof (1991). In line with van den Berg (1996), we took quantifiers to simply push the so-called reference set, $A \cap B$ for a statement $Q(A)(B)$, to the context. For instance, *Most students passed the test*, if true, will push the set of all students that passed the test to the input context. The semantic mechanism behind this need not concern us here. Here, I want to focus on the question what the addition of plural discourse referents means for our notion of context.

Since van den Berg (1996), it is standardly assumed that plural referents are not just stored in the context as complex individuals, but that they are rather *distributed* over that context (see also: Nouwen 2007, Brasoveanu 2008). This is because quantifiers do not just add potentially plural referents to the context, they also store dependencies between referents. For instance,

1. Every farmer owns exactly one donkey. And most of them beat it.

Here, the first sentence introduces the set of (all) donkey-beating farmers, and the set of all donkeys owned by a farmer. The second sentence shows that we do not just have access to these pluralities, we also have access to which farmer goes with which donkey. This is because the second sentence can only be interpreted as saying that for most farmers, the farmer in question beats the donkey *he or she* owns. If the context is a stack just containing a set of farmers and a set of donkeys, then we have not kept track of this dependency. Instead, we need to say that the context is a set of stacks, each containing one farmer and one donkey:

$$\{f_1d_1, f_2d_2, f_3d_3, \dots\}$$

The set of entities occurring on the first position in these stack is the relevant set of farmers. The set of entities in second position is the set of donkeys. The dependency is captured by the way these sets are distributed over the set of stacks.

Using my rather rusty Haskell skills, here is what this setup could look like:

```

import Data.List

data Entity = A | B | C | D | E | F | G | H | I | J | K | L | M
    deriving (Eq, Bounded, Ord, Enum, Show)

type StringEnt = [Entity]    -- strings of entities
type Context = [StringEnt]  -- set of strings of entities
type Coll = [Entity]        -- a collection is a set of entities

lookupIdx :: StringEnt -> Int -> Entity
lookupIdx [] i = error "undefined context element"
lookupIdx (x:xs) 0 = x
lookupIdx (x:xs) i = lookupIdx xs (i-1)

lookupIdxCtxt :: Context -> Int -> Coll
lookupIdxCtxt [] i = []
lookupIdxCtxt (x:xs) i = [lookupIdx x i] ++ lookupIdxCtxt xs i

distrib :: Context -> Entity -> Context
distrib [] e = []
distrib (x:xs) e = [x ++ [e]] ++ distrib xs e

push :: Context -> Coll -> Context
push x [] = x
push x [e] = distrib x e
push x (e:es) = distrib x e ++ push x es

project :: Context -> Int -> Coll
project x i = nub (lookupIdxCtxt x i)

cardi :: Context -> Int -> Int
cardi x i = length ( project x i )

```

To illustrate how this works, here is an example of some operations on an input context $\{AB, AC\}$. This is a context with two discourse referents, one of which is simply A and the other is the plural individual with B and C as its parts (usually written as $B \sqcup C$). The first line shows how this context was created.

```

*Main> push (push [[]] [A]) [B,C]
[[A,B],[A,C]]
*Main> push [[A,B],[A,C]] [D,E,F]
[[A,B,D],[A,C,D],[A,B,E],[A,C,E],[A,B,F],[A,C,F]]
*Main> project (push [[A,B],[A,C]] [D,E,F]) 0
[A]
*Main> project (push [[A,B],[A,C]] [D,E,F]) 1
[B,C]
*Main> project (push [[A,B],[A,C]] [D,E,F]) 2
[D,E,F]
*Main> cardi (push [[A,B],[A,C]] [D,E,F]) 1
2

```

As setup like this will now allow us to provide a general recipe for the dynamic semantics of quantifiers. In general, quantifiers are operators that take two sets and perform a cardinality test on these two sets. If successful, the intersection of the two sets, the reference set, will then be pushed to the stack. Here I give a schematic interpretation of *most* as an example.³

$$[[\text{most}]] = \lambda X.\lambda Y.\lambda c. \begin{cases} \text{push } c \ X \cap Y & |X \cap Y| > |X \setminus Y| \\ \emptyset & \text{otherwise} \end{cases}$$

Nothing to add

An issue arises as soon as downward entailing quantifiers come into the picture. A quantifier is downward entailing if for any A , B and B' such that $B \subseteq B'$ it is the case that $Q(A)(B') \Rightarrow Q(A)(B)$. The issue concerns the fact that $Q(A)(\emptyset)$ is true for any downward entailing Q and any A . Van Eijck and Nouwen remark:

"An[...] unsolved problem is the treatment of downward entailing quantifiers. There are essentially two issues. First of all, since we have chosen to make quantifiers dynamic by existentially introducing a new index, we presuppose the existence of a set of individuals satisfying the restrictor and scope of the quantifier relation. Of course, in case the determiner is downward monotone in its right argument, this relation is satisfied even if no such individual exists." (van Eijck and Nouwen, 2002, p.22)

In my dissertation (Nouwen 2003), I offered a way out of this dilemma. In case the reference set is empty, then incrementing is vacuous and consequently there will be nothing to refer to. Pronouns simply won't have the chance to refer back.

This idea already happens to be in place in the short Haskell fragment above (which is different from our 2002 paper):

```
*Main> push [[A,B],[A,D],[A,E]] []
[[A,B],[A,D],[A,E]]
```

That is, pushing the empty set onto the stack simply returns the input stack. No new position is created and, consequently, no new anaphoric potential has been added. This is fully in line with the philosophy of contexts in *Incremental Dynamics*: contexts are fully explicit about what the anaphoric potential of the context is. Whether or not downward entailing quantifiers introduce a discourse referent depends on the model. Only in models in which the intersection between restrictor and scope is non-empty will there be a new index. Using a pronoun to refer back to the quantifier presupposes that there is such an index and, as such, it presupposes that we are in a model in which that intersection is not empty.

This is an elegant solution to the problem, but one which I am now no longer fully convinced of. The reason is that this solution depends on a standard assumption about collections of entities that is often made in the literature, but which I now think is wrong. It is often assumed that there are two kinds of entities, namely atoms and pluralities, and that the latter are built the former. More precisely, the full set of entities is built by taking the powerset of the set of atomic entities and then removing the empty set. More commonly, the set of entities is actually thought to be only structurally similar to this: plural individuals are not sets but sums and the set of entities is a complete join semi-lattice with the atomic entities as smallest elements (Landman 1989). Explicitly, the assumption is that the structure is a semi-lattice and not a lattice: there is no unique bottom element, that is a part of any other element in the lattice. This is the detail that I now question.

There are (as far as I know) two arguments in the literature that natural language semantics needs to take a bottom element, i.e. an element with cardinality 0, into account. Bylinina and Nouwen (2017) provide a complex argument that the fact that most languages have a word for "0" (for instance, *zero*), entails that there has to be an full-fledged entity with 0

cardinality. The other is Landman (2011), which is more relevant to our discussion of dynamics. Landman notices that some plural definite descriptions appear to refer, even though the referent is empty:

1. Of the ten students in my class I would say that the students that studied for the test got a good grade (but nobody did).

The parenthetical continuation does not appear contradictory, and so the description *the students that studied for the test* must have referred to the empty individual. Note, especially the contrast with (3), where the singular morphology forces the definite description to refer to an atom, and hence to something that is not empty.

1. Of the ten students in my class I would say that the student that studied for the test got a good grade (#but nobody did).

On a conceptual level, this observation already goes against my assumption above that empty sets are referentially void. More generally, I believe it is not impossible that Landman's observation extends to pronouns.

1. Of the ten students in my class I would say that the students that studied for the test got a good grade and that they are likely to do well in the next course. Unfortunately, none of my students studied for the test.
2. Of the ten students in my class I would say that the student that studied for the test got a good grade and that he is likely to do well in the next course. #Unfortunately, none of my students studied for the test.

I will leave a detailed discussion of these examples for a future occasion. Potentially, they indicate that there can be empty discourse referents. In the terms of Incremental Dynamics this would mean that the operation of *adding nothing to the stack* does not simply return the input stack, but rather returns a stack with an empty value pushed on top of it. In fact, had we chosen not to implement the dependency phenomenon illustrated by (1) but instead taken a simpler view on context as a stack of collections, then we would have arrived at that behaviour straightforwardly. Take the following simpler code:

```
push2 :: Context -> Coll -> Context
push2 x e = x ++ [e]
```

and how it functions:

```
*Main> push2 [[A],[B,C]] []
[[A],[B,C],[ ]]
```

Here, a context is not a collection of stacks, but a stack of collections. Each position in that stack is a potentially plural and potentially empty entity (represented by a list). In such a setup, pronouns point to positions in the stack and come with a non-emptiness implicature, one that is cancelled in (3).

Of course, the simple `push2` operation above leaves no room to account for examples involving quantificational dependencies like (1). But I have no reason to believe that the absence of an empty element follows from adopting a view of context in which pluralities are distributed over stacks. It just so happened that my crude way of implementing this idea resulted in this behaviour. I leave it as an easy exercise for the reader to come up with a Haskell program that at the same time implements the idea of contexts with distributed pluralities and the idea that pluralities are potentially empty.

Notes

¹. For those familiar with the literature: what in Dynamic Predicate Logic amounts to a *random reset* is a *random push* in Incremental Dynamics. ↵

². Here and in what follows I am ignoring undefinedness cases when n exceeds the length of \mathbf{c} . ↵

³. This is a simplification. The arguments of *most* need to be interpreted in context too, for they may for instance have dynamic effects themselves. In fact, it is this dynamic interpretation of the X and Y argument that ultimately results in the storage of quantificational dependencies. See van den Berg (1996), Nouwen (2003) and Brasoveanu (2008) for discussion. ↵

References

Brasoveanu, Adrian. 2008. Donkey Pluralities: Plural Information States Versus Non-Atomic Individuals. *Linguistics and Philosophy* 31 (2): 129–209.

Bylinina, Lisa, and Rick Nouwen. 2017. *The Semantics of 'Zero'*.

- Groenendijk, J.A.G., and M.J.B. Stokhof. 1991. Dynamic Predicate Logic. *Linguistics and Philosophy* 14: 39–100.
- Kamp, H. 1981. A Theory of Truth and Semantic Representation. In *Formal Methods in the Study of Language*, edited by J. A. G. Groenendijk, T. M. V. Janssen, and M. J. B. Stokhof. Amsterdam: Mathematical Centre.
- Kamp, H., and U. Reyle. 1993. *From Discourse to Logic*. Dordrecht: D. Reidel.
- Landman, Fred. 1989. Groups. Part I, II. *Linguistics and Philosophy* 12: 559-605;723-744.
- Landman, Fred. 2010. Boolean Pragmatics. In *This Is Not a Festschrift*, edited by Jaap van der Does and Catarina Dulith Novaes.
- Nouwen, Rick. 2003. *Plural Pronominal Anaphora in Context*. Netherlands Graduate School of Linguistics Dissertations 84. Utrecht: LOT.
- Nouwen, Rick. 2007. On Dependent Pronouns and Dynamic Semantics. *Journal of Philosophical Logic* 36 (2): 123–54.
- Nouwen, Rick, Adrian Brasoveanu, Jan van Eijck, and Albert Visser. 2016. Dynamic Semantics. *Stanford Encyclopedia of Philosophy*.
- van den Berg, M.H. 1996. *Some Aspects of the Internal Structure of Discourse: The Dynamics of Nominal Anaphora*. PhD thesis, ILLC, Universiteit van Amsterdam.
- van Eijck, Jan. 2001. Incremental Dynamics. *Journal of Logic Language and Information* 10 (3): 319–51.
- van Eijck, Jan, and Rick Nouwen. 2002. Quantification and Reference in Incremental Processing. Unpublished Manuscript, CWI, ILLC and UiL-OTS. Presented at the (Preferably) Non-Lexical Semantics Conference.

A tribute to Jan van Eijck

Rohit Parikh

I have known Jan for eleven years, at least since the NIAS project in 2006. He is a very soft and polite person who has shared several interests with me. Among them is of course Epistemic reasoning in which both he and I have worked, and Social Software where we collaborated on a paper—more a dialogue than an essay.

Another interest we share is language, but in spite of strong interest on both our parts, we have not actually worked together. I hope this will change because his interest and expertise in language is truly impressive.

But we still have a lot in common which was the reason why I suggested to Sergei Artemov that Jan be invited to give the keynote talk for my 80th birthday conference.

Let me now proceed to some remarks that I would like to make and hope that Jan will approve.

Introduction

Epistemic logic started as a child of modal logic and **Kripke structures** were the usual tools for a long time. But of two classical books, one by **Hintikka** and one by **David Lewis**, only the first was primarily in the modal direction. Lewis' work was more game theoretic and it is widely believed that he was influenced by the Nobelist **Thomas Schelling**. Lewis died some years ago and Schelling only recently, but their influence has been strong.

The difference in these two approaches is between the one person case and the many person case. Descartes was interested in what **he** knew and we, **part time solipsists**, are still his children in a way. It is hard to escape being a Cartesian. Much work in epistemic reasoning is Cartesian in spirit.

But Lewis' notion of common knowledge (anticipated by **Robert Nozick**, see the paper by Cubitt and Sugden) is fundamentally a many person or social notion. Even a psychologist like Pinker is aware of **common knowledge** as is Herb Clark. And since common

knowledge has been important in distributed computing since the classic paper by Halpern and Moses, it is inevitable that computer scientists will come to play a large part.

This insight inevitably leads us to **game theory**, based on the notion of the super rational **homo economicus**. But as **Kahneman and Tversky** pointed out, the actual homo is not super rational. He (she) is quite often irrational, and even when rational, rational in a different way from what the classical theory of **von Neumann and Morgenstern**, and **Savage** presumes.

<http://www.nobelprize.org/mediaplayer/index.php?id=531>

We reveal our beliefs in action, and even answering **yes** or **no** is an **action**. As we know, an utterance may or may not supply the truth value of a proposition, but even when it does, it says **something about the speaker** (and what the speaker thinks about the listener) as well as about the world. The speaker wants to bring about a change in the mental state of the listener and the desired change may be something different from, or more than merely a change in her state of beliefs about the world.

A related issue is that we need devices for converting a game theoretic problem into a decision theoretic one. When an agent is playing a game with another, she thinks what the other might do. And having decided, she has converted the game theoretic problem into a decision theoretic one, "since he is going to do x, what shall I do?" Thus suppose there are two restaurants A and B and Bob might want to go to the same restaurant as Ann. Then if he knows that Ann is going to B, his best decision will be to go to B himself. If she is going to A then his best decision will be to go to A. If she usually goes to A, then also his best bet is to go to A. So once he has resolved his doubts about Ann's action, he has a decision theoretic problem.

If he wants to go to the same restaurant as Ann he too should go to A. If he wants to go to a different one, then once he decides that she is going to A, he should go to B.

So once he has an idea about her beliefs, her preferences and hence her probable action, his game theoretic problem becomes a decision theoretic one. But knowing others' preferences and beliefs is not always easy. We often know their ordinal utilities but not their cardinal ones and Savage's method of eliciting responses will not work in real life.

Here is an example.

Ann lives in San Francisco and Bob is visiting her from out of town.

After giving him dinner, Ann asks, "Would you like some ice cream?"

"Do you have chocolate?" says Bob.

"I am sorry I do not but I do have vanilla and strawberry" says Ann.

"Vanilla, then" says Bob.

Now Ann knows that for Bob the utilities have the order chocolate > vanilla > strawberry.

But she does not know the cardinal utilities. So she proceeds.

"Oh, I am sorry, I do have chocolate. Would you like vanilla or a 50-50 chance of chocolate and strawberry?"

Bob is puzzled but says, "Vanilla."

"And how about vanilla versus a 70% chance of chocolate and 30% chance of strawberry?"

What is the matter with Ann? says Bob to himself and then says aloud,

"Actually the doctor told me to avoid ice cream. How about just coffee?"

Ann now knows that that she will not find out more without being rude.

So we live in a world where we know **something** about the beliefs and preferences of others but do not and cannot know more precisely. Constraints of time and politeness prevent us from finding out everything. And yet we get along by and large.

It is clear that if we are to have a theory of people which is realistic and yet allows us to say something more than bare common sense then we need to move away from the overly precise models of classical epistemic logic and *Homo Economicus*.

Logic and Society

One of the strange things about us Homo Sapiens is that evidence reveals that we are more clever than wise. We have settled the entire globe, developed the internet and sent expeditions to the moon and to Mars.

Given this, how is that we have not achieved the condition that most or all humans are happy and well taken care of? We look at war movies with dismay and pain but we should also be **amazed** that humans are doing such things to other humans.

Here is joke about two aliens in a space ship discussing humans.

One says, "They have even developed missiles!"

The other says, "Then they must be very advanced."

The first says, "Not really—they have aimed them at themselves!"

What we humans need now is the ability to be amazed that we are so poor at taking care of ourselves. According to legend, Newton was amazed that the apple fell on his head rather than away from his head. And that the moon revolved around the earth even though there was no rope tying the moon to the earth. This amazement eventually led to his theory of gravitation, amended by Einstein in the early XXth century.

What was the big deal that the Michelson-Morley experiment did not reveal the "speed at which the earth travels through the ether?" That information is not going to help us to pay our taxes. And yet, being amazed at our inability to detect that speed led to the theory of Relativity.

So we need to be amazed that we are so poor at taking care of our species and other species whom, for whatever reason, we seem to love. We should not say, "It has always been so." Many things have always been so and are not any longer. A hundred years ago we did not have airplanes, telephones, let alone the internet.

Also, the major wars in 1914-18 and 1939-45 did not lead to any further major wars in Europe and the furthest we have come from war in Western Europe is Brexit! So progress is possible.

A usual explanation for why things are bad is "some bad guy" (it is usually a guy). That guy can be Hitler or Trump or whoever. But we do not ask where these bad guys get so much power. And we do not ask whether putting Trump in the same boat with Hitler might not be a result of our need to look for "bad guys".

It was not Trump who invaded Iraq or who bombed that hospital in Kunduz.

Sages like Jesus or Dalai Lama look to **personal transformation**. If we become compassionate and loving then there will be no bad guys and the world will be better—perhaps even perfect. (See the book by Thich Nhat Hanh.)

I am skeptical. The tendency to cooperate is built into us humans. See the little essay by Tomasello.

<http://www.nytimes.com/2008/05/25/magazine/25wwln-essay-t.html>

But aggression and excessive ambition are also built into us. The millionairess, for whatever reason, does not want to share any excess wealth. She wants to become a billionaire and "break the glass barrier"!

And people keep having more and more children, creating ever greater demands on the planet. Ditto for the illusion that the more energy we consume, the better off we are. Certainly individual human failings are behind our problems.

But I do not want to make a moral lesson. I disagree with the Dalai Lama much as I admire him. I want to look for techniques which will allow us **imperfect humans** to create a good society. So I look for systems where even selfish people will act so as to make everyone happy. Adam Smith had this idea long ago but we all agree that his idea does not work invariably. So we need systems which harness self-interest into the good of society.

Economic Design is in a way a contribution in this direction. The Vickrey auction is an example of an auction where being honest with one's bid is a winning strategy. In the Vickrey auction, the highest bidder wins but only pays the bid of the second highest bidder. Then it turns out that being honest about one's own preferences is a dominant strategy.

Perhaps we can set up **social software** which encourages people to act in a rational way which helps other people as well.

Where does logic come in?

Ultimately, economists are **practical logicians**. They are not concerned as much with truth as with what works. So they are more in the realm of practical reason. But practical reason and theoretical reason are not unconnected. We have game theoretic accounts of truth in

first order logic. And backward induction is a logical argument telling us how a certain game will be played.

So logicians can help by coming up with systems which encourage people to act in ways which are good not only for them but also for other human beings.

Criminal law has this function. "Harm others and society will harm you!" But politicians who enact laws are not logicians. They tend to come up with "remedies" which do not work.

The US incarcerates at a rate 4 to 7 times higher than other western nations such as the United Kingdom, France, Italy, and Germany and up to 32 times higher than nations with the lowest rates such as Nepal, Nigeria, and India.

https://www.nccdglobal.org/sites/default/files/publication_pdf/factsheet-us-incarceration.pdf

Cleverer devices are needed to encourage people to help themselves by helping others. And the right way is an analysis using tools from social psychology, Economic Design and the logic of programs to ask what will work with actual people.

Once we have such a solution, then the problem will arise of getting the politicians to go along. But that part of the project is, luckily, far from **our** work.

Language

One reason why we have misunderstandings is that language is too poor to express the complexities of the world. Suppose that the "natural language" for the world consists of ten unary predicates. But the average person can only deal with five at one moment. Then there will be a many one homomorphism which will conflate 32 different states of the world into a single description. And moreover, not everyone will use the same homomorphism. Thus words like "feminist", "fascist", "progressive" mean different things to different people.

Is Bernie Sanders a communist? Is Trump a fascist? Different people with the same facts will come up with different judgments.

A related problem is that of multiple identities. For instance Mohammad Ali Jinnah, the founder of Pakistan was a Muslim, whereas Gandhi was a Hindu. But they were both Gujaratis and both lawyers. So are they opponents or were they fellow travelers?

Amartya Sen goes into this issue of the conflict between different identities. And it can well be true that professional identities which usually have less emotions attached to them, can overcome national or religious identities which are more invested with passion. Thus at the CUNY Graduate Center where I teach there is nothing unusual about someone from Israel being the doctoral supervisor of someone from Iran.

Perhaps if we remember our multiple identities, then some of the anguish of the world can be resolved. In the US currently, it is political identities which have become most passionate and a white Democrat would have no problem with his daughter marrying a black man, but would be very upset at her marrying a Republican. So let me end with a joke from my undergraduate days.

Girls from a Catholic school are graduating and one by one, each goes to the Mother Superior to receive her diploma. The Mother Superior asks each of them what she is going to do now that she has graduated.

One girl says, "Mother, I am going to be a prostitute."

The Mother faints and is revived with cold water and smelling salts.

"What did you say my dear?" asks the Mother Superior.

"Mother, I am going to be a prostitute," is the reply.

"Thank God," says the Mother. "I thought you said a Protestant."

References

Clark, Herbert H., and Catherine R. Marshall. Definite reference and mutual knowledge. In Joshi, Webber and Sag, *Elements of discourse understanding*. (1981).

Cubitt, Robin P., and Robert Sugden. Common Knowledge, Salience and Convention: A Reconstruction of David Lewis' Game Theory. *Economics and Philosophy* 19.02 (2003): 175-210.

- Van Ditmarsch, Hans, et al. On the logic of lying. *Games, actions and social software*. Springer Berlin Heidelberg, 2012. 41-72.
- Eijck, Jan. *Discourses on social software*. Ed. Rineke Verbrugge. Amsterdam University Press, 2009.
- Hanh, Thich Nhat. *Peace is every step: The path of mindfulness in everyday life*. Bantam, 1991.
- Halpern, Joseph Y., and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)* 37.3 (1990): 549-587.
- Hardin, Garrett. The Tragedy of the Commons*. *Journal of Natural Resources Policy Research* 1.3 (2009): 243-253.
- Hayek, Friedrich August. The use of knowledge in society. *The American economic review* (1945): 519-530.
- Hintikka, Jaakko. *Knowledge and belief*. (1962).
- Hintikka, Jaakko, and Gabriel Sandu. *Game-Theoretical Semantics*, Chapter 6. (1997).
- Kahneman, Daniel, and Amos Tversky. Prospect theory: An analysis of decision under risk. *Econometrica: Journal of the econometric society* (1979): 263-291.
- Lewis, David. *Convention: A philosophical study*. John Wiley & Sons, 2008.
- Parikh, Rohit. D-structures and their semantics. Appeared in a volume dedicated to Johan van Benthem, University of Amsterdam (1999).
- Parikh, Rohit. Social software. *Synthese* 132.3 (2002): 187-211.
- Parikh, Rohit, Çağil Taşdemir, and Andreas Witzel. The power of knowledge in games. *International Game Theory Review* 15.04 (2013): 1340030.
- Parikh, Rohit, and Jouko Väänänen. Finite information logic. *Annals of Pure and Applied Logic* 134.1 (2005): 83-93.
- Pinker, Steven, <https://www.youtube.com/watch?v=3-son3EJTrU>
- Savage, Leonard J. *The foundations of statistics*. Courier Corporation, 1972.
- Schelling, Thomas C. *Micromotives and macrobehavior*. WW Norton & Company, 2006.

Sen, Amartya. *Identity and violence: The illusion of destiny*. Penguin Books India, 2007.

Van Benthem, Johan. Games in Dynamic-Epistemic Logic. *Bulletin of Economic Research* 53.4 (2001): 219-248.

Programming Real Social Software: Matching Students to Supervisors using Perl

Marc Pauly

Introduction

Thinking back of my time as a PhD student with Jan van Eijck, programming is not one of the things that immediately comes to my mind. My PhD thesis was about the connection between logic, game theory and social choice theory, and coding was not part of the work I did for my thesis. But when I was thinking back a bit further, I remembered that Jan taught a course I took during my Master of Logic program in 1996 or 1997 that dealt with the semantic foundations of programming. I do not remember the title of the course, but the book we used for the course was called "Semantics with Applications: A Formal Introduction" (Nielson & Nielson, 1992). I am still very fond of this book: In a very clear way, it covers operational semantics, denotational semantics and axiomatic program verification for a simple language called the *while-language*. The book also contains an implementation in the functional language Miranda, a predecessor of the later Haskell language that Jan worked with. As I think back, I realize that it was also thanks to this course that I was able to extend some ideas about program semantics and verification to a programming language for games (Pauly, 2005).

Ever since my move from computer science to philosophy, I am not surrounded any more by people who can (or at least at some point were able to) program. Still, it turns out that this ability comes in handy even for somebody working at a philosophy department. I would like to illustrate this with a short code snippet we use to solve a specific two-sided matching problem.

The Problem: Matching Students to Supervisors

At the philosophy department of the university of Groningen, between 50 and 100 students per year write their bachelor theses under the supervision of some staff member. In the past, students would approach a teacher and ask him or her to become their supervisor. Two problems led us to change this. On the one hand, this way of doing things led to certain staff members to attract many students while other staff members supervised almost nobody. This unequal distribution was not too problematic at a time when student numbers were low, but with more than 50 students per year especially the popular supervisors felt that we needed a new system. Given my own interest in social choice theory and game theory, I suggested to use the well-known Gale-Shapley algorithm for two-sided matching (Gale & Shapley, 1962). In this algorithm, each student submits a preference list of possible supervisors, saying who their 1st choice, 2nd choice, etc. for supervisor is. Similarly, each supervisor submits a list of possible students who they could imagine supervising, also ranked in order of preference. Also, each supervisor is associated with a given capacity of students to supervise, usually somewhere between 0 and 5. Given these preference lists of the two groups, the Gale-Shapley algorithm works as follows: First, every student applies to their first choice supervisor. If a supervisor has no more applicants than his or her supervision capacity, nothing happens. If, on the other hand, a supervisor has more applicants than capacity, the supervisor rejects those applicants which rank lowest on his or her preference list. In the second round, those students who have been rejected apply to their second choice supervisor, and again the supervisors reject depending on their capacity those applicants they least prefer. This process continues until all students are assigned. The process is guaranteed to terminate as long as the supervisor capacities when added together are at least as high as the number of students who need a supervisor. A well-known property of this algorithm is that it will result in a student-supervisor matching that is stable: Whenever a student A is matched to a supervisor B , there will never be another student A' matched to supervisor B' such that A prefers B' to B and B' also prefers A to A' .

So much for the theory. As always, the devil is in the detail. In the concrete situation of the philosophy department, there is one important extra complication to the basic setup: Not all supervisors speak Dutch, and many students want to write their thesis in Dutch. So my task was to first think of an extension of the algorithm that can handle this more complicated situation, before actually coding it.

As for the input, we now need to ask for additional pieces of information. We ask the students what language they want to write their thesis in. The options are *Dutch*, *English*, or *Any* (meaning: Dutch or English are both fine). Similarly for the supervisors, we ask

them what language they can supervise in, and here the options in practice are only *English* (only) or *Any*.

As for the algorithm, note that we first need to make sure that there are enough supervisors who speak Dutch to cater to the students who want to write in Dutch. In the past years, this fortunately has not been a problem, although with a continuing inflow of academics who do not speak Dutch, this problem may arise in the future. For now, we will assume that this problem does not arise.

How to modify the original Gale-Shapley algorithm? My first attempt was to do a language check during each round of the matching algorithm, making sure that each time we tentatively match a student with a supervisor, we make sure that the languages do not conflict. However, this attempt did not work. The problem can be illustrated with a simple example: Suppose that supervisor Jan speaks Dutch and English and prefers Marc to Sanne. Marc has Jan as his most preferred supervisor and does not care about the language he writes his thesis in, but Sanne wants to write her thesis in Dutch. Now we can imagine a stage in the matching algorithm where Marc is matched to Jan and Sanne is yet unmatched. From the perspective of Jan and Marc, everything is fine, and the languages match. However, if Sanne cannot find a supervisor who speaks Dutch it may turn out that we have to unmatch Marc and Jan in order to match Sanne to a supervisor (Jan) who is able to speak Dutch, even though Jan prefers Marc to Sanne. Clearly, in terms of preferences, this is a bad move, but due to hard language constraint, the move seems unavoidable.

Given this difficulty, I decided to match supervisors and students in two phases: First, only the supervisors who only speak English are considered. Their slots are filled first, using the Gale-Shapley algorithm. Second, the students who remain because they have not been matched to the English-only supervisors are matched to the remaining supervisors. This algorithm is not ideal for a number of reasons, but given the language constraint I do not think that we can do much better. The most important property we lose is the property of stability: As the example above with Jan and Marc already illustrates, the new algorithm can give rise to situations where an ideal couple (a student and a supervisor who have each other as their first choice) does not end up being matched to each other.

The Code

The main program first retrieves student and supervisor data from files. Since students and supervisors will have submitted only partial preference lists (e.g., a student may have only mentioned her top 5 supervisors), we first complete everyone's preference list by randomly ordering the candidates that have not been explicitly mentioned below these that have been mentioned explicitly. Note that this is what introduces an element of nondeterminism into the algorithm (in fact, the only nondeterminism). Next, we first match those supervisors who only speak English to those students who are willing to write their thesis in English using the Gale-Shapley algorithm. Afterwards, the unmatched students and supervisors are matched in a second round of Gale-Shapley matching. The Perl implementation of the Gale-Shapley matching algorithm is given below.

```
# Arguments passed: teachers, teacher-preferences, students, student preferences
# Returns reference to match hash

sub gs_algo (\@\%\@\%) {
    print "\nExecuting Gale-Shapley algorithm...\n";

    my $round=0;           # keeps track of the round of the algorithm
    my %student_apply;    # for each student the current choice he is
                           # applying to
    my %assigned;         # has value 1 for a student in case student
                           # is currently
                           # assigned to a supervisor
    my $some_unassigned = 1; # is true whenever unassigned students remain;
    my $teachers_ref = shift; # first reference argument passed to subroutine
    my @teachers = @{$teachers_ref}; # dereferencing;
    my $teacher_prefs_ref = shift;
    my %teacher_prefs = %{$teacher_prefs_ref};
    my $students_ref = shift;
    my @students = @{$students_ref};
    my $student_prefs_ref = shift;
    my %student_prefs = %{$student_prefs_ref};
    my %match;

    for my $name (@teachers) {
        @{$match{$name}}=(); # initialize matching to the empty matching
    }

    for my $name (@students) {
```

```

        $student_apply{$name} = 0; # this initializes all applications to
first choice
        $assigned{$name} = 0;      # and all students are initially
    }

    while ($some_unassigned) {
        $round++;
        print "\nRound $round:\n";

        # all unassigned students apply to their next choice, if there is
one
        for my $stu (keys %assigned) {
            if ($assigned{$stu} == 0) {
                my @prefs = @{$student_prefs{$stu}};
                my choice = prefs[$student_apply{$stu}];
                if (defined $choice) {
                    push@{match{$choice}}, stu;
                    $assigned{$stu}= 1;
                    $student_apply{$stu}++;
                }
                else # the student has run out of options to apply to and
drops out
                    {$assigned{$stu}= 1}
            }
        }
        $some_unassigned = 0;

        print "After application by students we have:\n";
        print_matching(@teachers,%match);

        # teachers reject students beyond their capacity, according to th
eir preferences
        for my $tea (@teachers) {
            my cur_capacity = teacher_caps{$tea};
            my @new_match = (); # new preference list of supervisor initi
alized

            for my $stu (@{$teacher_prefs{$tea}}) {
                my $is_present = 0 ;
                for my $j (@{$match{$tea}}) # check if student was matche
d to teacher
                {
                    if (stu eq j) {$is_present = 1};
                }
                if ($is_present) {
                    if ($cur_capacity > 0) {

```

```

        $cur_capacity--;
        push@new_match, $stu;
    }
    else
    {
        $assigned{$stu} = 0;
        $some_unassigned = 1;
    }
}
}
@{$match{$tea}}= @new_match;
}

print "After rejections by teachers we have:\n";
print_matching(@teachers,%match);
}

return %match;
}

```

Due to the nondeterminism, the matching produced will differ depending on how the initial preference profiles of students and supervisors were completed. For this reason, I usually run the program 1000 times and see which matching is best in terms of minimizing the number of total mismatches, i.e., the number of student-supervisor pairs (s, S) where s did not mention S in her initial preference list and S did not mention s in her initial preference list either. Those pairs will be listed explicitly by the algorithm, and together with a colleague I look at these cases to see whether these cases present real problems. In some cases, the problem will have arisen because a student applied late and did not submit any preference list and hence also the supervisors did not take this student into account when submitting their preferences. Hence, the application of the algorithm comes down to a combination of deterministic algorithm, nondeterministic preference completion, and human interpretation and selection of the results produced by the algorithm.

Conclusions

There are a number of lessons I learned from working on this problem. First, when applied to actual situations, algorithms often need to be modified and complicated to take account of all the relevant features of the situation. Second, such a complication can (and I suspect often will) lead to the loss of nice theoretical properties which the original algorithm had.

Third, practice has the tendency to lead to an ever increasing series of complications of initially simple algorithms, which leads not only to a loss of nice theoretical properties but also to a loss of transparency, the understandability of the algorithm decreases. Having said that, my impression is that my colleagues do consider the current situation of algorithmic matching an improvement over the initial situation where this process was not centralized. So while *algorithmic justice* leaves many things to be desired, there are situations where it can improve the world we live in. A hopeful message to end with in a Festschrift devoted to coding. Thank you, Jan!

Further reading

Jan van Eijck has edited a volume on games, actions and social software (van Eijck & Verbrugge, 2012), which gives more food for thought concerning some of the issues that came up in this contribution. In this volume, there is also an article by Rohit Parikh and myself (p.3-13) that introduces the notion of social software. Also, this article discusses another practical two-sided matching problem, the problem of the Stanford Housing Draw where students need to be matched to dormitories.

Another practical example of a social software problem is designing the competition schedule of sports tournaments. At the 2012 London Olympics, strategic play led to the disqualification of badminton teams. In order to meet a supposedly easier opponent in the next round of a round-robin tournament, these teams both wanted to lose rather than win their match. While this led to hilarious badminton, the public felt cheated. This raises the question whether it might not be possible to design a tournament schedule that avoids all possibilities for strategic manipulation. Besides the paper containing the technical result (Pauly, 2014), there is also a more accessible exposition of the problem and the result (Pauly, 2015). As it turns out, the technical result involves a computer-assisted proof: At some point in the proof, a C-program is used to verify that of the 65,536 possible functions to consider, none has a particular combination of properties. The program can be accessed at [this link](#).

References

van Eijck, J. & R. Verbrugge, eds. (2012). *Games, Actions and Social Software: Multidisciplinary Aspects*. Heidelberg, Germany: Springer.

Gale, D. & L. Shapley (1962). College admissions and the stability of marriage. *American Mathematical Monthly* 69:9-15.

Nielson, H.R. & F. Nielson (1992). *Semantics with Applications: A Formal Introduction*. Chichester, UK: Wiley.

Pauly, M. (2005). Programming and Verifying Subgame-Perfect Mechanisms, *Journal of Logic and Computation* 15(3): 295-316.

Pauly, M. (2014). Can strategizing in round-robin subtournaments be avoided? *Social Choice and Welfare*, 43(1):29-46.

Pauly, M. (2015). Winning Isn't Everything: How Sports Competition Rules Can Make You Want to Lose, *The Mathematical Intelligencer*, 37(3):66-71.

A conversation on money

R Ramanujam

December 2016. A logician, a philosopher and a computer scientist are travelling by car from Chennai to Tiruvannamalai in southern India. All of them are European and terrified by the Indian highway. The only way the travellers can keep sane is to keep their minds occupied. As it happens, there is something that is bothering them: money.

Philosopher: Did you know before you left home that getting hold of cash would be a big problem in India?

Logician: I did, and a fat lot of good it did me. But I realised I better find some cash at the airport as soon as I landed, be it at 3 AM. And I managed to get cash too!

Computer Scientist: I had heard, but thought it mattered little since I could use my credit card. And what happens when I try the card here? The connectivity is no good and the connection keeps getting timed out.

Logician: All this because the Indian government announced one day last month, out of the blue, that the 500 and 1000 rupee currency notes would be worthless from that day. Given that 85 percent of circulating cash was in these notes, it demonetised the economy in one stroke.

Computer Scientist: People have until year-end to change the notes they have, but the queues in banks are horrendously long, and most ATMs are dead. Naturally, since printing the new notes and getting them to banks is a logistical nightmare.

Really, if our societies would get used to *digital cash*, all these problems would disappear.

Philosopher: The trouble is that we tend to think of money as a **thing**, that we can possess. It is not only something to *use*, but also something to *have*. Digital money is fine for using, but I am not sure human beings will be happy to lose the element of possession.

Logician: Currency notes are mere statements of contract. How does having currency notes give you the sense of possession you refer to?

Philosopher: Good question. We tend to think of money as a projective space, as a means of linearly ordering the value of all goods—material goods, at least. This is important for trading so that rather than exchanging a ceramic vase for a table, we can talk trade using monetary value. Unfortunately, the abstraction then starts applying to non-material goods, like labour, and then it becomes a thing in its own right.

Computer Scientist: But we now have cryptocurrencies like the *bitcoin*. These are spontaneously generated, simply by evidence of work. After that it is all blockchains of transactions.

Logician: You are saying that a currency note is simply a historical record of all the transactions it has been used in, are you? The central bank spontaneously generates its value, and afterwards its further value is determined by the transactions it participates in.

Computer Scientist: Yes, but not quite. This applies only to the money value that the currency note carries, not to the currency note itself. The note is exchangeable with another freshly issued by the bank, one that carries no history.

Philosopher: So then we are back to the original question. Is money a thing?

The driver asks if they would like to stop for a coffee. They would, but nobody has anything smaller than a 2000 Rupees note, and the driver is not sure whether the small coffee place would take credit cards. They stop anyway, but the credit card payment is unsuccessful.

Computer Scientist: I am really puzzled why the card didn't work. Ah I am getting internet connection on my phone, let me check

Ohmygod, my bank has sent a message that they have blocked my card since they were getting a suspicious transaction request! Now I have neither cash nor a usable card! I will also need to make an international call to unblock it.

Philosopher: Currency does not need trust but credit does. Banks, as holders of trust, are naturally distrustful. (*Laughter. Computer scientist gives a dirty look.*) Oh sorry, I hope your card gets activated soon.

Logician: Interesting. We have talked of money as a thing, as value, as history, as a trust-carrier. Is it all of these in some haphazard manner, or is it an abstraction that somehow underlies all this?

Computer Scientist: It is mainly an *enabler* of economic transactions.

Philosopher: That is a very interesting formulation. It enables not only economics but also social status, pride, distrust, many such things.

Logician: Rohit Parikh makes a plea for *social software*, an enterprise that tries to figure out the software underlying society. Money is of course a crucial component of such software.

Computer Scientist: With currency being the hardware? Isn't that too simplistic?

Philosopher: We are looking for an abstraction that enables and shapes the functioning of society.

Logician: Yes, an abstraction that can then be also manipulated in its own right.

Computer Scientist: That is easy, that is exactly what we call a data structure !

Logician: This is very interesting. Money as data structure underlying social algorithms?

Philosopher: Yes, I agree. Thinking of money as a structuring device is useful. It helps to explain how algorithms in society create value as well as measure value using it, and how the structure carries memory of transactions. But how can a data structure reflect epistemic attitudes and social status?

Computer Scientist: One basic feature of a data structure is that it is essentially defined by what operations are allowed on it. A queue and a stack are similar, but the former is first-in-first-out and the latter is last-in-first-out. It carries social memory, but how that memory is accessed and updated is what we need to understand.

Logician: That it is collective memory is important, I think. This is shaped by what each person knows, what each knows about the other, what is common knowledge in society...

Philosopher: You have got knowledge, dynamics, data structures, social algorithms... Aren't we entering Jan van Eijck territory?

The driver informs them that they are reaching Tiruvannamalai, where Jan has reached previously and is even now climbing the hill.

True Lies and True Love

Hans van Ditmarsch, Ji Ruan, and Yanjing Wang

The famous Zen master Gaiarosa wants to make two friends Heleen and Jan attend a Buddhist meditation retreat. She knows that they are dying to get close to each other. Thus one will come if and only if (s)he believes that the other will come. Obviously, they do not yet wish to admit this to each other, because they are uncertain about each other's feelings. Given the uncertainty about the other attending, both in fact intend not to go to the retreat. Gaiarosa now lies to Heleen that Jan will come to the retreat and she lies to Jan that Heleen will come to the retreat. As a result, they both come to the retreat, declare their love to each other (in between meditation sessions), and live happily ever after. Sadly, shortly afterwards Gaiarosa passed away, but in her next incarnation, she was reborn as twins.

Introduction

You lie if you say something that you know to be false with the intention to make the listener believe that it is true. This analysis of lying goes back to Augustine [2], and every serious work on lying, including Jan van Eijck's [17], starts with this citation. Lying has been a thriving topic in the philosophical community [12, 5, 9], and typically more recent modal logical analyses that can be seen as a continuation of this philosophical tradition include [3, 13, 7, 17, 11, 8, 15]. In modal logics with (only) belief operators one cannot model the intentional aspect of lying.

How do we model a lie in a dynamic epistemic logic? In dynamic epistemic logic a lie is a dynamic operation (with a corresponding dynamic modality) transforming an information state (that encodes the beliefs or the knowledge of the agents) into another information state (wherein they may consequently have different beliefs or knowledge). In *truthful public announcement logic* [10] we cannot announce something false thus making it true. The announced formulas are supposed to be true. 'Truthful' is actually a misnomer, new information is simply assumed to come from an outside, reliable, source. Or, rather more precisely: in this logic we only model the consequences of incorporating new information abstracting from the process that made it reliable in the first place. So in truthful public

announcement logic we cannot model that the source of information is unreliable (because an agent is mistaken or lying). In an alternative semantics for public announcement logic, that of conscious update [6], and that is also known as *believed (public) announcement logic*, the announcement of a formula is independent of its truth. In that logic, we can call an announcement a lie if the announced formula is false. The term 'lie' is justified because the observing/listening agents consider this false announcement to be true, and may therefore incorrectly revise their beliefs. This analysis of lying has been pursued in [17].

What is missing in this analysis of lying in dynamic epistemic logic is that the announcing agent itself is not modelled: in [17], a lie is simply an announcement that is false, not an announcement that is believed to be false by an announcing agent. But there is a solution for that. In public announcement logic, it is common to model the truthful announcement that ϕ by agent a who is modelled in the system, as the truthful public announcement of 'agent a knows ϕ '. This analysis also extends to believed public announcement logic and to 'agent a believes ϕ ' (knowledge and belief are both formalized in our setting as $\Box_a \phi$). A precondition for that epistemic action should then be that agent a believes $\neg\phi$ (formalized as $\Box_a \neg\phi$). Still, the lie that ϕ by agent a cannot simply be the believed public announcement of $\Box_a \phi$, because agent a is also addressing herself with that announcement and surely does not believe her own lie. In a variation on the [17] semantics for lies in believed announcement logic, a *lie by agent a to agent(s) b* can then be modelled as an epistemic action wherein agent b believes that a truthfully informs him of ϕ , whereas agent a herself does not change her beliefs. This *agent lie* (from a to b) is presented in [15]. In this setting, agent a is lying, when saying ϕ but believing $\neg\phi$ (thus, the precondition for the action is $\Box_a \neg\phi$), whereas agent a is telling the truth (is truthful), when saying ϕ and believing ϕ (thus, precondition $\Box_a \phi$). A third option in this setting is when agent a is saying ϕ , and thus acting as if she believes ϕ , but really is ignorant about ϕ : such an action has precondition $\neg(\Box_a \phi \vee \Box_a \neg\phi)$. We can call that *bluffing*. When modelling individual agents lying to other agents, we can thus distinguish agents that are lying from agents that are bluffing or agents that are truthful (or more, such as randomizing agents saying anything whatsoever based on throwing a dice). This approach wherein different types of agents behave in different ways is pursued in [8].

Among public lies there are different sorts of lies. First, let the lie ϕ be that of a propositional variable p . In believed public announcement logic, a public lie that p , with therefore precondition that p is false, results in the agents incorrectly believing that p is true. So, before the lie that ϕ , ϕ is false, and after the lie that ϕ , ϕ is still false. Now consider a public lie that $p \vee \Box p$. Consider a model with designated state (a pointed

Kripke model) wherein the negation of this formula is true: $\neg p \wedge \neg \Box p$, and such that the agent considers it possible that p , so the model contains accessible p -states. If this a $\mathcal{KD45}$ model (a Kripke model wherein all accessibility relations are serial, transitive, and euclidean; which is said to encode the beliefs of agents with consistent beliefs), then none of the accessible states satisfy $\Box p$. The result of the believed public announcement that $p \vee \Box p$ in our model will therefore only preserve accessibility links to p states, such that after this update $\Box p$ is true in the designated state, and therefore $p \vee \Box p$. So, in this case, where $\phi = p \vee \Box p$, before the lie that ϕ , ϕ is false, and after the lie that ϕ , ϕ has become true. One can call this a *true lie*. This is investigated in [1].

The lie by Gaiarosa to Jan and Heleen can be viewed as a true lie in general, since it eventually makes the lie true. However, it is a true lie of a different sort than the above, and on two counts. Firstly, it is not a public lie, to all agents, but a private lie (namely to Jan only, or to Heleen only). Secondly, it involves agents changing their mind (namely Jan, and Heleen), which in dynamic epistemic logic is modelled as ontic change, i.e., as an assignment of different values to certain variables. And apart from that, this assignment is done in private: Jan changes his mind without Heleen knowing, and vice versa. Only the combination of these various events can be interpreted as making something true by lying about it. Such more complex epistemic actions can most elegantly be modelled in *action model logic* [4], such that we can eventually model check their consequences in Jan van Eijck's DEMO [19].

Action model logic

In this section we present action model logic (with factual change). Its language, structures, and semantics are as follows. For more details, see [4, 14, 16] (our presentation follows [16]). Given are a finite set of agents A and a countable set of propositional variables P .

The language \mathcal{L} is inductively defined as

$$\phi ::= p \mid \neg\phi \mid (\phi \wedge \phi) \mid \Box_a \phi \mid [M, s]\phi \text{ where } p \in P, a \in A,$$

and where *epistemic action* (M, s) is defined below (and assumed to be simultaneously defined with the language). Other propositional connectives are defined by abbreviation. For $\Box_a \phi$, read 'agent a believes formula ϕ '. If there is a single agent only, we may omit the index and write $\Box \phi$ instead. Agent variables are a, b, c, \dots . For $[M, s]\psi$, read 'after

execution of epistemic action $(M, s), \psi'$. If $\Box_a \neg \phi$, we say that ϕ is *unbelievable* (for a) and, consequently, if $\neg \Box_a \neg \phi$, for which we write $\Diamond_a \phi$, we say that ϕ is *believable* (for a). This is also read as 'agent a considers it possible that ϕ '.

We continue by defining the structures. An *epistemic model* $M = (S, R, V)$ consists of a *domain* S of states (or 'worlds'), an *accessibility function* $R : A \rightarrow \mathcal{P}(S \times S)$, where each $R(a)$, for which we write R_a , is an accessibility relation, and a *valuation* $V : P \rightarrow \mathcal{P}(S)$, where each $V(p)$ represents the set of states where p is true. For $s \in S$, (M, s) is an *epistemic state*.

An epistemic state is also known as a *pointed Kripke model*. We often omit the parentheses in (M, s) . Without any restrictions we call the model class \mathcal{K} . The class of models where all accessibility relations are transitive, euclidean and serial is called $\mathcal{KD45}$, and the class of models where all accessibility relations are equivalence relations is $\mathcal{S5}$. Class $\mathcal{KD45}$ is said to have the *properties of belief*, and $\mathcal{S5}$ to have the *properties of knowledge*.

An *action model* $M = (S, R, \text{pre}, \text{post})$ for language \mathcal{L} (simultaneously defined above) consists of a domain S of *actions*, an *accessibility function* $R : A \rightarrow \mathcal{P}(S \times S)$, where each $R(a)$, for which we write R_a , is an accessibility relation, a *precondition function* $\text{pre} : S \rightarrow \mathcal{L}$, that assigns to each action its executability precondition, and *postcondition function* $\text{post} : S \rightarrow P \rightarrow \mathcal{L}$, where it is required that each $\text{post}(s)$ only maps a finite subset of all atoms to a formula. For $T \subseteq S$, (M, T) is an *epistemic action* (or multi-pointed action model). For $(M, \{s\})$ we write (M, s) .

For epistemic models and for action models we assume the usual visual conventions for $\mathcal{KD45}$ models: all directed arrows point to *clusters* of indistinguishable nodes for that agent (i.e., for all states $s \in S$ and $a \in A$, $(s, s) \in R_a$ unless there is a state $t \in S$ such that $(s, t) \in R_a$ but $(t, s) \notin R_a$); where a cluster is a set of indistinguishable nodes, such that transitivity, reflexivity, and symmetry can be assumed: instead of directed arrows it suffices to connect indistinguishable nodes with undirected arrows, i.e., links. In the next section we use such visualizations.

Assume an epistemic model $M = (S, R, V)$, a state $s \in S$, an action model $M = (S, R, \text{pre}, \text{post})$ with an action $s \in S$. The interpretation of formulas $\phi \in \mathcal{L}$ is defined by induction (simultaneously with that of action model execution).

$$\begin{array}{ll}
M, s \models p & \text{iff } s \in V_p \\
M, s \models \neg\phi & \text{iff } M, s \not\models \phi \\
M, s \models \phi \wedge \psi & \text{iff } M, s \models \phi \text{ and } M, s \models \psi \\
M, s \models \Box_a \phi & \text{iff for all } t \in S : R_a(s, t) \text{ implies } M, t \models \phi \\
M, s \models [M, \mathbf{s}]\psi & \text{iff } M, s \models \text{pre}(\mathbf{s}) \text{ implies } M \otimes M, (s, \mathbf{s}) \models \psi
\end{array}$$

where $M \otimes M = (S', R', V')$ (known as *update of M with M* , or as the *result of executing M in M*) is such that $S' = \{(s, \mathbf{s}) \mid M, s \models \text{pre}(\mathbf{s})\}$; $((s, \mathbf{s}), (t, \mathbf{t})) \in R_a$ iff $(s, t) \in R_a$ and $(\mathbf{s}, \mathbf{t}) \in R_a$; and $(s, \mathbf{s}) \in V'(p)$ iff $M, s \models \text{post}(\mathbf{s})(p)$ for all p in the domain of $\text{post}(\mathbf{s})$, and otherwise $(s, \mathbf{s}) \in V'(p)$ iff $s \in V(p)$. The semantics of multi-pointed action models is (purely for convenience) defined by abbreviation as $M, s \models [M, \mathbf{T}]\psi$ iff (for all $\mathbf{s} \in \mathbf{T}$, $M, s \models [M, \mathbf{s}]\psi$).

Throwing all further explanations and justifications to the wind, we can now define the action models corresponding to the epistemic actions already referred to in the informal introductory section and that will be used in the continuing analysis of the story on true lies and true love.

Consider an epistemic action (M, \mathbf{T}) , where $M = (S, R, \text{pre}, \text{post})$. We distinguish the following types of epistemic actions. Obviously, postconditions for actions (partial functions) are only specified on their finite domain of propositional variables, that also may be empty.

(continued on next page)

True Lies and True Love

Hans van Ditmarsch, Ji Ruan, and Yanjing Wang

(continued from previous page)

- private lie that ϕ to agent a

$S = \{s, t, u\}$ and $T = \{t\}$; $R_a = \{(t, s), (s, s), (u, u)\}$ and for all $b \in A \setminus \{a\}$, $R_b = \{(s, u), (t, u), (u, u)\}$; $\text{pre}(s) = \phi$, $\text{pre}(t) = \neg\phi$, and $\text{pre}(u) = \top$.

- public assignment that variable p becomes ϕ

$S = T = \{s\}$; $R_a = \{(s, s)\}$ for all $a \in A$; $\text{pre}(s) = \top$; $\text{post}(s)(p) = \phi$.

- private assignment for agent a that variable p becomes ϕ (agent a changing her mind)

$S = \{s, t\}$ and $T = \{s\}$; $R_a = \{(t, t), (s, s)\}$, and $R_b = \{(s, t), (t, t)\}$ for all $b \in A \setminus \{a\}$; $\text{pre}(s) = \text{pre}(t) = \top$; $\text{post}(s)(p) = \phi$ and $\text{post}(t)(p) = p$.

- lie that ϕ by agent a

$S = \{s, t, u\}$ and $T = \{t\}$; $R_a = \{(t, t), (s, s), (u, u)\}$, and $R_b = \{(t, s), (s, s), (u, s)\}$ for all $b \in A \setminus \{a\}$; $\text{pre}(s) = \Box_a \phi$, $\text{pre}(t) = \Box_a \neg\phi$, $\text{pre}(u) = \neg(\Box_a \neg\phi \vee \Box_a \phi)$.

- private announcement of ϕ to agent a

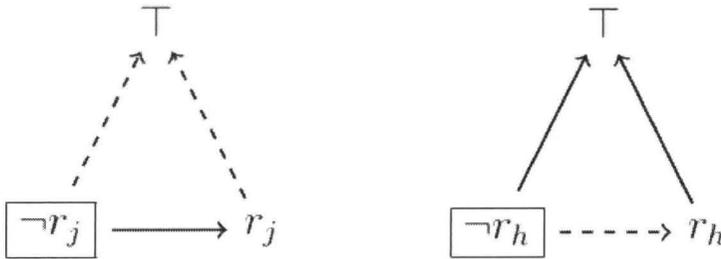
$S = \{s, t\}$ and $T = \{s\}$; $R_a = \{(s, s), (t, t)\}$, and $R_b = \{(s, t), (t, t)\}$ for all $b \in A \setminus \{a\}$; $\text{pre}(s) = \phi$ and $\text{pre}(t) = \top$.

- truthful public announcement of ϕ

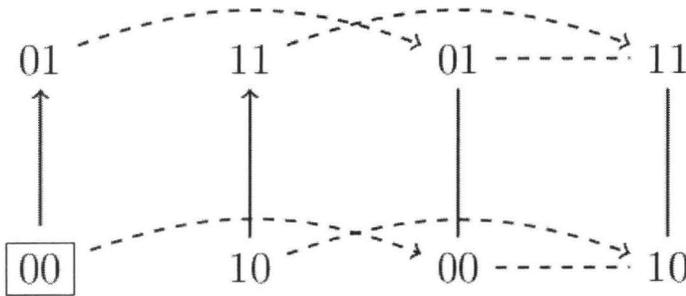
$S = T = \{s\}$; $R_a = \{(s, s)\}$ for all $a \in A$; $\text{pre}(s) = \phi$.

- believed public announcement of ϕ

Gaiarosa now lies to Heleen, in private, that Jan goes to the retreat. For the convenience of the reader informed about action model logic, we can model this as a three-pointed action model as follows, on the left—where for convenience we have put the similar private lie to Jan about Heleen next to it, on the right. (We recall that we consequently use the $\mathcal{KD45}$ visualization where for all states x , $(x, x) \in R_a$ unless there is a state y such that $(x, y) \in R_a$ but $(y, x) \notin R_a$, for $a = j, h$.)



The result of the first of these lying actions is as follows.



Now Gaiarosa lies privately to Jan that Heleen goes to the retreat. The result of that action is shown in Figure 1. For the convenience of the reader we also depict (on the right) the restriction of this model to the submodel generated by the point $\$00\$$.

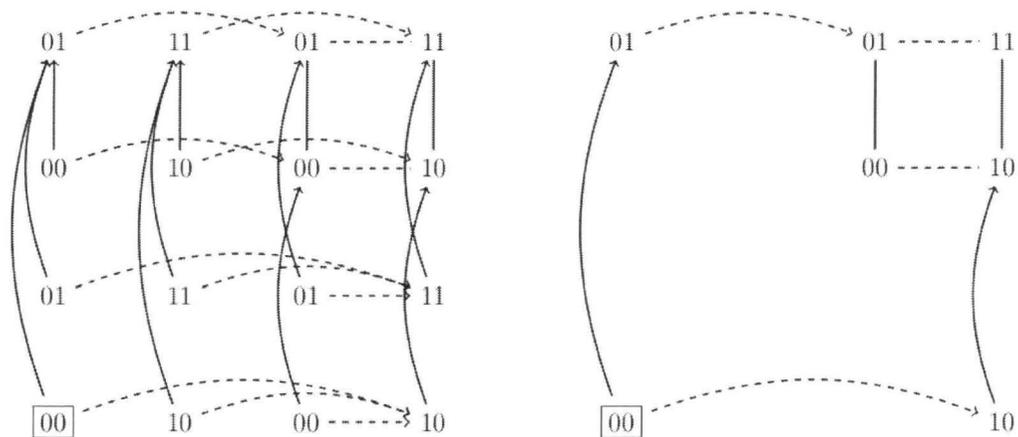


Figure 1: The resulting model after two lies and its generated submodel under bisimulation `\label{modelresult:twolies}`. This model formalizes that: Heleen is not going to the retreat, believes that Jan goes to the retreat, and believes that Jan is uncertain whether she goes to the retreat; and that: Jan is not going to the retreat, believes that Heleen goes to the retreat, and believes that Heleen is uncertain whether he goes to the retreat. See the DEMO implementation in the next section. We now let first Heleen and then Jan change their mind. Heleen changing her mind can again be formalized as an action model, namely as a private assignment to Heleen; and similarly, Jan changing his mind as a private assignment to Jan. It is important here that a *public* assignment is an improper way to formalize this action: a public assignment of Heleen going to the retreat if she believes that Jan goes to the retreat would be informative to Jan in case he were to believe that she believed that he was going to the retreat. Because in case he was uncertain if she would go to the retreat, he would then learn from this public assignment that she would come to the retreat for his sake. Boring. Because exactly the absence of this sort of knowledge of the other's intentions makes first lovers' meetings so thrilling. That kind of uncertainty should *not* be resolved. Therefore, we formalize it as a private assignment. Interestingly, in the current model the result of a public and of a private assignment (the result of Heleen privately changing her mind or publicly changing her mind) is the same. But that is because both Heleen and Jan believe that the other is uncertain whether they go to the retreat. We will of course again corroborate this in DEMO. Below on the left is the action model for Heleen changing her mind, and on the right, the one for Jan changing his mind. (So, for example, according to our conventions, in the left action model the solid relation, that of Heleen, has identity access, and the dashed relation, for Jan, has only a reflexive arrow in the point that the dashed arrow is pointing to.)

$$\boxed{r_h := \square_h r_j} \dashrightarrow \top \qquad \boxed{r_j := \square_j r_h} \longrightarrow \top$$

We now depict in Figure 2, from left to right, once more the model before they change their minds, the model resulting from executing the action of Heleen changing her mind, and the model resulting from Jan changing his mind, where once again we restrict the actually resulting models to the point-generated subframes.

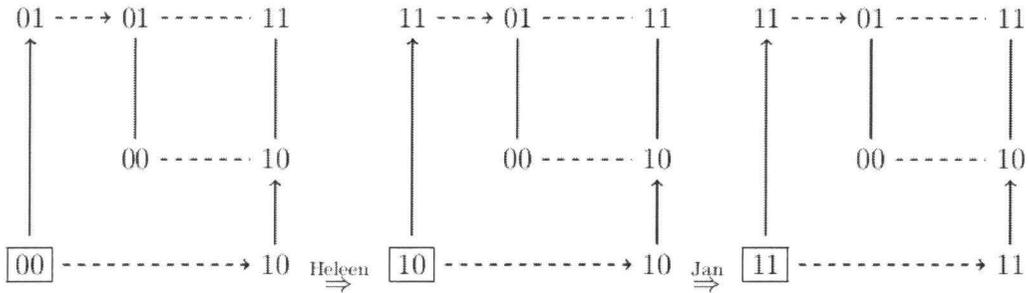
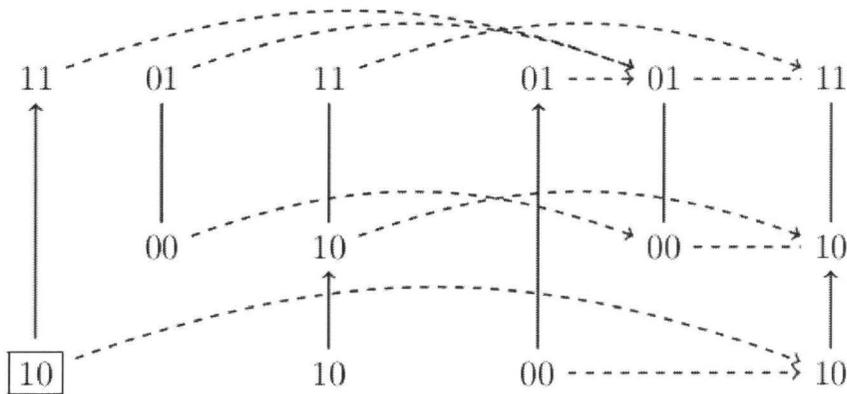


Figure 2: The models before and after Heleen and Jan changed their minds Now that Heleen and Jan have changed their minds, the lies have become the truth! They both go to the retreat, and they both expect the other to be surprised to find them at the retreat.^[^2] They declare their love to each other and they both live happily ever after. We conclude this section with two further technical observations on the model constructions and the analysis. Firstly, one can imagine Heleen already changed her mind before Gaiarosa informs Jan that Heleen is going to the retreat—in which case Gaiarosa would no longer have been lying. But that is a modelling artifact. To avoid such a scenario we simply assume that Gaiarosa simultaneously sends two *letters* to Heleen and to Jan containing the lies. At that moment both are indeed lies. Then again, the information change affected in Heleen and Jan depends on the moment the letter is opened... It does not greatly matter: Heleen changing her mind can be modelled both before and after Gaiarosa talking to Jan. But lying twice is far more interesting than lying once only. Secondly, as mentioned, when executing the private assignments we restricted ourselves to point-generated subframes. Without that restriction, for example, based on the model on the left above, the model in the middle above would look as follows:



Clearly the simplified visualization is better.

DEMO implementation

We implement the above modelling and verify the proposed properties in the DEMO (short for Dynamic Epistemic MOdelling) [19], which is a model checking tool developed mainly by Jan van Eijck, with contributions from his students since 2004. Our code runs on a slightly modified version of DEMO Light [18] and the Glasgow Haskell Compiler GHC, version 8.0.2.

Here is the declaration of the module.

```

module Truelieslove where

import Data.List
import ModelsVocab hiding (m0)
import ActionVocab
import ChangeVocab hiding (public, preconditions, voc)
import ChangePerception

```

We first define two special agents h and j and the atomic propositions that representing their come to the retreat. We omit the Zen master Gaiarosa as her knowledge and belief are not the focus of the modelling.

continued

```
h, heleen, j, jan :: Agent
h = Ag 5; heleen = Ag 5
j = Ag 6; jan = Ag 6

rh, rj :: Form
rh = Prp (RH) -- this represents Heleen comes to the retreat
rj = Prp (RJ) -- this represents Jan comes to the retreat
```

The initial epistemic model is defined as `mod0` .

```
mod0 = (Mo [0..3] [h,j] [RH, RJ] val accs points)
  where
    val = [(0,[]), (1,[RJ]), (2,[RH]), (3,[RH,RJ])]
    accs = [(h,x,y) | x <- [0,1], y <- [0,1]] ++
           [(h,x,y) | x <- [2,3], y <- [2,3]] ++
           [(j,x,y) | x <- [0,2], y <- [0,2]] ++
           [(j,x,y) | x <- [1,3], y <- [1,3]]
    points = [0]
```

The private lie to Heleen that Jan goes to the retreat is encoded in the action model `amlie2h` .

```
amlie2h :: FAM State
amlie2h ags = Am [0,1,2] ags [(0,(Neg rj)), (1,rj), (2, Top)]
                [(h,0,1), (h,1,1), (h,2,2), (j,0,2), (j,1,2), (j,2,2)] [0]
]
```

The private lie to Jan is encoded in action model `amlie2j` in a similar fashion.

```
amlie2j :: FAM State
amlie2j ags = Am [0,1,2] ags [(0,(Neg rh)), (1,rh), (2, Top)]
                [(j,0,1), (j,1,1), (j,2,2), (h,0,2), (h,1,2), (h,2,2)] [0]
]
```

The resulting epistemic models of two lies are then kept in `mod1` and `mod2` . The corresponding `mod1'` and `mod2'` are the submodels generated under bisimulation:

```

mod1 = up mod0 amlie2h
mod1' = bisim mod1
mod2 = up mod1' amlie2j
mod2' = bisim mod2

```

The following shows `mod2'`, which corresponds to the generated submodel in Figure 1.

```

*Truelieslove> mod2'
Mo [0,1,2,3,4,5,6] [h,j] [rh,rj]
[(0,[]), (1,[]), (2,[rj]), (3,[rj]), (4,[rh]), (5,[rh]), (6,[rh,rj])]
[(h,0,2), (h,1,1), (h,1,3), (h,2,2), (h,3,1), (h,3,3), (h,4,5), (h,4,6), (h,5,5),
 (h,5,6), (h,6,5), (h,6,6), (j,0,4), (j,1,1), (j,1,5), (j,2,3), (j,2,6), (j,3,3),
 (j,3,6), (j,4,4), (j,5,1), (j,5,5), (j,6,3), (j,6,6)] [0]

```

We formalize in `heleenbelief` the following statement "Heleen is not going to the retreat, believes that Jan goes to the retreat, and believes that Jan is uncertain whether she goes to the retreat". Similarly we have `janbelief` for Jan. Here the operator κ represents belief as it is interpreted over $\mathcal{KD45}$ models.

```

heleenbelief = Conj [Neg rh, K h rj, K h januncertain]
janbelief    = Conj [Neg rj, K j rh, K j heleenuncertain]

januncertain = Neg (Disj [(K j rh), (K j (Neg rh))])
heleenuncertain = Neg (Disj [(K h rj), (K h (Neg rj))])

```

We verify that the above two properties are both true in the model `mod2`. The results with respect to `mod2'` are the same due to bisimulation.

```

*Truelieslove> isTrue mod2 heleenbelief
Just True
*Truelieslove> isTrue mod2 janbelief
Just True
*Truelieslove> isTrue mod2' heleenbelief
Just True
*Truelieslove> isTrue mod2' janbelief
Just True

```

We now turn to the mind changing actions. The following action model `acm4h` encodes that Heleen changes her mind privately, and `acm4j` encodes Jan's private mind changing.

continued

```
acm4h :: FACM State
acm4h ags = Acm [0,1] ags [(0,(Top,[(RH, K h rj]))), (1,(Top,[]))]
      [ (h,0,0), (h,1,1), (j,0,1), (j,1,1) ] [0]

acm4j :: FACM State
acm4j ags = Acm [0,1] ags [(0,(Top,[(RJ, K h rh]))), (1,(Top,[]))]
      [ (j,0,0), (j,1,1), (h,0,1), (h,1,1) ] [0]
```

The resulting epistemic models from these two actions are kept in `mod3` and `mod4`. Their generated submodels `mod3'` and `mod4'` refer to the second and third models depicted in Figure 2.

```
mod3 = upc mod2' acm4h
mod3' = bisim mod3
mod4 = upc mod3' acm4j
mod4' = bisim mod4
```

Both Heleen and Jan now believe that the other comes to the retreat. But Heleen believes that Jan does not believe that she plans to come, and further more, she believes that Jan believes that Heleen is uncertain whether Jan comes. This can be encoded in the following formula `heleenbeliefnew`. Similarly Jan's belief is encoded in `janbeliefnew`.

```
heleenbeliefnew = Conj [K h rj, K h (Neg (K j rh)), K h (K j heleenuncertain)]
janbeliefnew    = Conj [K j rh, K j (Neg (K h rj)), K j (K h januncertain)]
```

We can verify that they are both true in `mod4`.

```
*Truelieslove> isTrue mod4 heleenbeliefnew
Just True
*Truelieslove> isTrue mod4 janbeliefnew
Just True
```

Acknowledgements

We dedicate this little piece of analysis to Jan van Eijck, in grateful acknowledgements for many different interactions over many years.

- *Hans van Ditmarsch* recalls Jan van Eijck from an ESSLLI summer school in Barcelona, in 1998, where he was impressed by Jan going around on a bicycle (and where Jan still had a mop of black curly hair; those were the days), and he recalls that this bicycle later was stolen. However, we did not really know each other at that time, I think. Not so long afterward, it must have been in 1999 (or maybe even before ESSLLI, it is all mixed up in my mind), we had a joint meeting in Johan van Benthem's office at ILLC (at Plantage Muidergracht), at a time when I was much struggling with my PhD. Jan then made the memorable statement that I had learnt swimming but apparently was afraid to plunge and go for it, and kept standing on the shore or hold my hands to the railing while in the water. No progress! Well, something to that effect, in Dutch, involving fear of flying and swimming. He was right! Shortly afterwards, I plunged. Since then, we have collaborated on many occasions (a shared ESSLLI course, publications on the riddle of 100 prisoners, the *Logic in Action* project, two Lorentz Center workshops, etc., etc.). Last but not least, I have also been Jan and Heleen's guest on many occasions, both in Amsterdam and in Lavidalle (arriving, of course, on bicycle), which has always been greatly stimulating for my musical state of mind. Thanks to you both! To all four of you!
- *Ji Ruan* was introduced to Jan by Johan van Benthem when Ji was working on his master thesis in 2004. I became very interested in Jan's DEMO project and we worked on a structural characterization of two action models being equivalent. Jan was the person leading the direction and I felt very fortunate for being inspired and encouraged by him. On finishing my thesis, Jan then introduced me to Hans van Ditmarsch, who at the time was working at University of Otago in New Zealand. Hans subsequently supported me to work with him in Otago for three months, and that was my first time to the home of middle-earth, as a fan of *The Lord of the Rings*! Like Hans, I have also been Jan and Heleen's guest on several occasions, when Gaia and Rosa were still little. I really enjoyed the time with you and the homemade dutch soups were tasty. Now that I live and work in Auckland, you will be my guest if you visit this beautiful country someday.
- *Yanjing Wang* did his Ph.D. with Jan, but he met Jan for the first time long before that (and it was actually due to Ji). On the second day after I arrived in Amsterdam for my master study in 2004, Jan kindly helped Ji to move (lots of stuff) to my place by his car. Both Ji and I were clearly last-minute people, in particular on that day, but Jan has been such a patient and optimistic person, who encouraged and helped us in all kinds of moves in our lives. I still remember vividly that after my Ph.D. interview,

Jan took me to one of those coffee machines at CWI and proposed to share an Earl Grey tea bag with me (my first time to share that!). Indeed, in the next couple of years, we shared much more than tea. Besides work, I especially enjoyed the philosophical conversations with Jan and Heleen in their lovely house. Even after my Ph.D., we still witnessed many important moments in each other's lives. I am looking forward to hearing exciting stories from Jan after his moving to a new stage of life.

Notes

¹. This is a story adapted from an example in [1]. ↵

². In the actual world **11** of the third model in Figure 2, Heleen believes that Jan does not know that she plans to come, and vice versa for Jan, although they both in fact believe that the other is coming. ↵

References

- [1] T. Ågotnes, H. van Ditmarsch, and Y. Wang. True lies. *Synthese*, 2017. to appear.
- [2] St. Augustine. De Mendacio. In P. Schaff, editor, *A Select Library of the Nicene and Post-Nicene Fathers of the Christian Church*, volume 3. Eerdmans, 1956, 1988.
- [3] A. Baltag. A logic for suspicious players: Epistemic actions and belief updates in games. *Bulletin of Economic Research*, 54(1):1–45, 2002.
- [4] A. Baltag, L.S. Moss, and S. Solecki. The logic of public announcements, common knowledge, and private suspicions. In *Proc. of 7th TARK*, pages 43–56. Morgan Kaufmann, 1998.
- [5] S. Bok. *Lying: Moral Choice in Public and Private Life*. Random House, New York, 1978.
- [6] J.D. Gerbrandy and W. Groeneveld. Reasoning about information change. *Journal of Logic, Language, and Information*, 6:147–169, 1997.
- [7] B. Kooi and B. Renne. Arrow update logic. *Review of Symbolic Logic*, 4(4):536–559, 2011.

- [8] F. Liu and Y. Wang. Reasoning about agent types and the hardest logic puzzle ever. *Minds and Machines*, 23(1):123–161, 2013.
- [9] J.E. Mahon. Two definitions of lying. *Journal of Applied Philosophy*, 22(2):21–230, 2006.
- [10] J.A. Plaza. Logics of public communications. In *Proc. of the 4th ISMIS*, pages 201–216. Oak Ridge National Laboratory, 1989.
- [11] C. Sakama. Formal definitions of lying. *Proc. of 14th TRUST*, 2011.
- [12] F.A. Siegler. Lying. *American Philosophical Quarterly*, 3:128–136, 1966.
- [13] D. Steiner. A system for consistency preserving belief change. In *Proc. of the ESSLLI Workshop on Rationality and Knowledge*, pages 133–144, 2006.
- [14] J. van Benthem, J. van Eijck, and B. Kooi. Logics of communication and change. *Information and Computation*, 204(11):1620–1662, 2006.
- [15] H. van Ditmarsch. Dynamics of lying. *Synthese*, 191(5):745–777, 2014.
- [16] H. van Ditmarsch, W. van der Hoek, and B. Kooi. *Dynamic Epistemic Logic*, volume 337 of *Synthese Library*. Springer, 2007.
- [17] H. van Ditmarsch, J. van Eijck, F. Sietsma, and Y. Wang. On the logic of lying. In *Games, Actions and Social Software*, LNCS 7010, pages 41–72. Springer, 2012.
- [18] J. van Eijck. Demo light for composing models. Technical report, CWI, 2011.
- [19] J. van Eijck. DEMO—a demo of epistemic modelling. In J. van Benthem, D. Gabbay, and B. Löwe, editors, *Interactive Logic—Proc. of the 7th Augustus de Morgan Workshop*, pages 305–363. Amsterdam University Press, 2007. Texts in Logic and Games 1.

Hintikka's world

François Schwarzentruber

Abstract

We present an online software for playing with higher-order knowledge of agents. Playgrounds (muddy children, Sally and Anne, etc.) are described in Dynamic Epistemic Logic.

1 Introduction

Higher-order knowledge of agents is relevant in many applications: game theory [3], robotics ([14], [9]), specifications of distributed systems [11], etc. Dynamic Epistemic Logic (DEL) ([4], [17], [16]) extends epistemic logic ([13], [12]) for describing and reasoning about epistemic properties and information change. Recently, model checking in DEL has been proven to be possible in practice [18] via the tool DEMO¹, even in with symbolic techniques [15].

Nevertheless, none of these tools have a graphical interface that may be used by roboticists, game theorists, psychologists, etc. In this paper, we present such a tool called *Hintikka's world* along the lines of *Tarski's world* [5] and *Kripke's worlds* [10].

Section 2 presents the graphical user interface. Section 3 explains the architecture of the software and how new examples can easily be implemented. Section 4 discusses the perspectives.

2 Graphical user interface

Figure 1 shows the graphical user interface of *Hintikka's world*². The example taken here is the muddy children where two agents $\$a\$$ and $\$b\$$ are muddy and it is common knowledge one sees the state of the other agent while not knowing its own state.

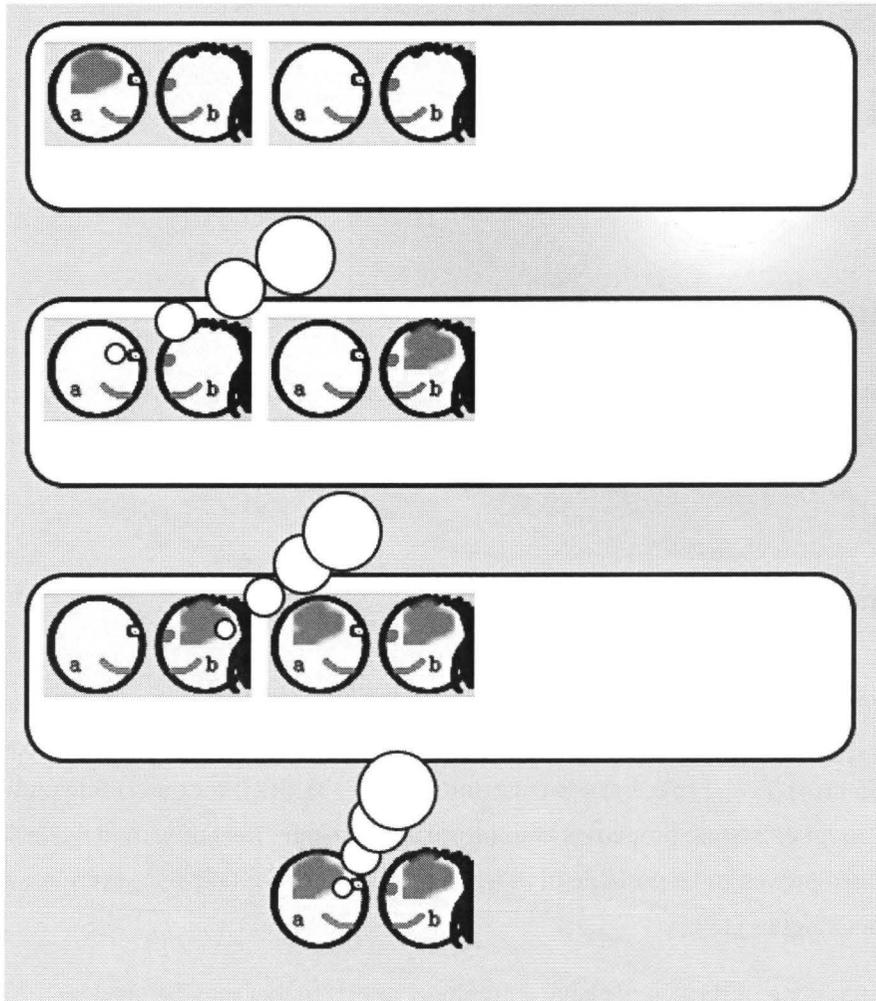


Figure 1: Graphical user interface of *Hintikka's world*

By clicking on agents, the interface shows the possible worlds for that agent and it shows the unfolding of the current pointed Kripke model that models the current situation.

On the right, the software shows buttons for possible actions (public announcement, public actions, private actions, etc.). Actions are modeled by pointed event models. By clicking on a button, the corresponding action is executed: the product of the pointed Kripke model and the pointed event model becomes now the current pointed Kripke model.

3 Architecture

Figure 2 shows the main part of the architecture of *Hintikka's world*. The interesting part is the fact that the graphical user interface (GUI) is independent from the current example that is running (muddy children, Sally and Anne, etc.). In particular, adding a new example only requires to add a new class that inherits from `World` and to implement the method for drawing the scene from data (valuations, numbers, etc.) that are members of the class.

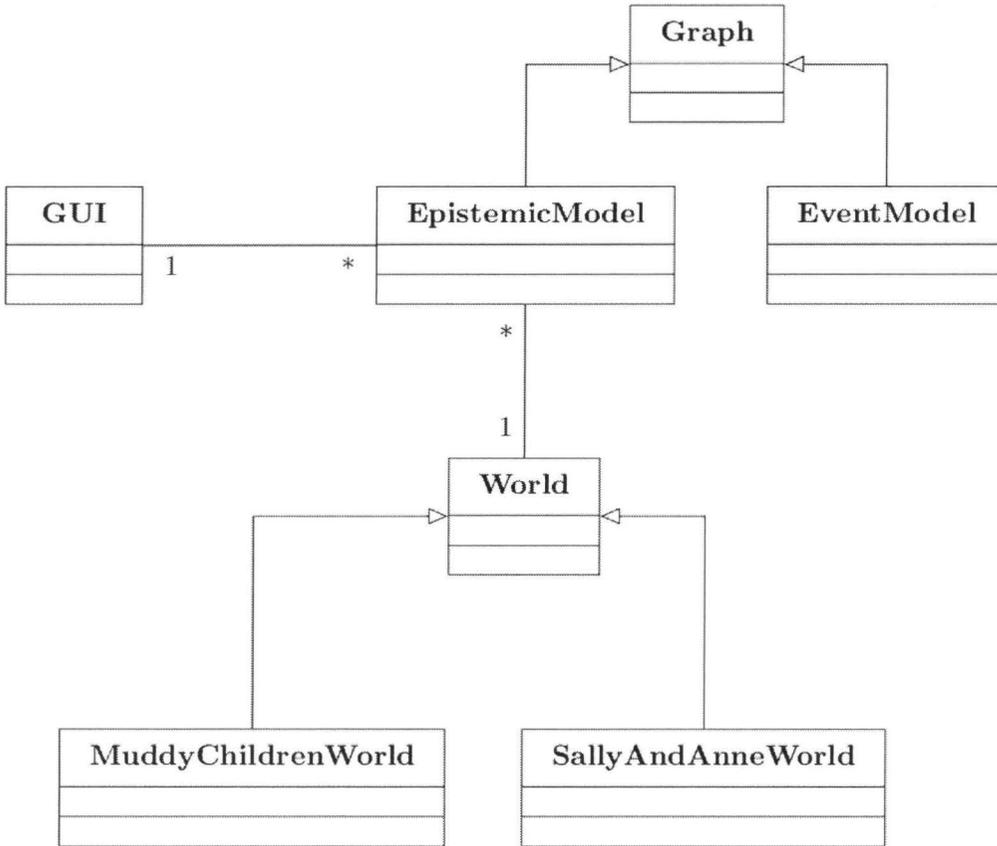


Figure 2: Architecture of **Hintikka's world**

4 Perspectives

Improving the reasoning tool inside *Hintikka's world*.

We believe that *Hintikka's world* should embed the model checker DEMO ([18], [15]). We also want to extend the tool by implementing algorithms for epistemic planning (even bounded epistemic planning because epistemic planning is undecidable in the general case ([6], [2], [7])) and arbitrary public announcements, also in the succinct cases ([8]).

Improving the graphical user interface.

By clicking on agents, the interface shows some possible worlds. We want to implement heuristics for displaying the most relevant epistemic worlds when there are too many possible worlds for a given agent.

Applications of the tools.

We plan two kind of applications. First, the tool may help children for learning to reason about higher-order knowledge (see [1]). Secondly, the tool may be an interface for displaying mental states of real human-aware robots ([14], [9]).

Notes

¹. http://homepages.cwi.nl/~jve/software/demo_s5/,
<http://homepages.cwi.nl/~jve/software/prodemo/> ↩

². <http://people.irisa.fr/Francois.Schwarzentruher/hintikkasworld/> ↩

References

- [1] Burcu Arslan, Rineke Verbrugge, Niels Taatgen, and Bart Hollebrandse. Teaching children to attribute second-order false beliefs: A training study with feedback. In *Proceedings of the 37th Annual Meeting of the Cognitive Science Society, CogSci 2015, Pasadena, California, USA, July 22-25, 2015*, 2015.
- [2] Guillaume Aucher and Thomas Bolander. Undecidability in epistemic planning. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013.

- [3] Robert J. Aumann. Interactive epistemology I: knowledge. *Int. J. Game Theory*, 28(3):263–300, 1999.
- [4] Alexandru Baltag, Lawrence S Moss, and Slawomir Solecki. The logic of public announcements, common knowledge, and private suspicions. In *Proceedings of the 7th conference on Theoretical aspects of rationality and knowledge*, pages 43–56. Morgan Kaufmann Publishers Inc., 1998.
- [5] David Barker-Plummer, Jon Barwise, and John Etchemendy. *world: Revised and expanded*. 2007.
- [6] Thomas Bolander and Mikkel Birkegaard Andersen. Epistemic planning for single and multi-agent systems. *Journal of Applied Non-Classical Logics*, 21(1):9–34, 2011.
- [7] Tristan Charrier, Bastien Maubert, and François Schwarzentruber. On the impact of modal depth in epistemic planning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, USA, July 12-15, 2016*, 2016.
- [8] Tristan Charrier and François Schwarzentruber. Arbitrary public announcement logic with mental programs. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, Istanbul, Turkey, May 4-8, 2015*, pages 1471–1479, 2015.
- [9] Sandra Devin and Rachid Alami. An implemented theory of mind to improve human-robot shared plans execution. In *The Eleventh ACM/IEEE International Conference on Human Robot Interaction, HRI 2016, Christchurch, New Zealand, March 7-10, 2016*, pages 319–326, 2016.
- [10] Olivier Gasquet, Andreas Herzig, Bilal Said, and François Schwarzentruber. *Kripke's Worlds - An Introduction to Modal Logics via Tableaux*. Studies in Universal Logic. Birkhäuser, 2014.
- [11] Joseph Y. Halpern and Ronald Fagin. Modeling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–177, 1989.
- [12] Jaakko Hintikka. Reasoning about knowledge in philosophy: The paradigm of epistemic logic. In *Proceedings of the 1st Conference on Theoretical Aspects of Reasoning about Knowledge, Monterey, CA, March 1986*, pages 63–80, 1986.

- [13] Saul A Kripke. Semantical analysis of modal logic | normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963.
- [14] Brian Scassellati. Theory of mind for a humanoid robot. *Auton. Robots*, 12(1):13–24, 2002.
- [15] Johan van Benthem, Jan van Eijck, Malvin Gattinger, and Kaile Su. Symbolic model checking for dynamic epistemic logic. In *Logic, Rationality, and Interaction - 5th International Workshop, LORI 2015 Taipei, Taiwan, October 28-31, 2015, Proceedings*, pages 366–378, 2015.
- [16] Johan van Benthem, Jan van Eijck, and Barteld P. Kooi. Logics of communication and change. *Inf. Comput.*, 204(11):1620–1662, 2006.
- [17] Hans van Ditmarsch, Wiebe van der Hoek, and Barteld Kooi. *Dynamic Epistemic Logic*. Springer, Dordecht, 2008.
- [18] Hans van Ditmarsch, Jan van Eijck, Ignacio Hernández-Antón, Floor Sietsma, Sunil Simon, and Fernando Soler-Toscano. Modelling cryptographic keys in dynamic epistemic logic with DEMO. In *Highlights on Practical Applications of Agents and Multi-Agent Systems - 10th International Conference on Practical Applications of Agents and Multi-Agent Systems, PAAMS 2012 Special Sessions, Salamanca, Spain, 28-30 March, 2012*, pages 155– 162, 2012.

Beste Jan,

Vandaag vieren wij de voltooiing van een prachtige loopbaan. Ik weet zeker dat je met jouw sprankelende persoonlijkheid velen hebt geïnspireerd. Mij in ieder geval wel. Tijdens mijn promotieonderzoek stond jij altijd voor me klaar, en je bent een grote steun voor me geweest zowel op inhoudelijk gebied als met alles eromheen. Jong als ik was had ik het gevoel dat jij op de achtergrond een oogje in het zeil hield, en mij tegelijkertijd de vrijheid gaf om het op mijn eigen manier te doen. Ik hoop dat de komende jaren je veel plezier gaan brengen en de start van een prachtige nieuwe reis, wat de bestemming ook mag zijn. Als bijdrage aan dit Festschrift heb ik twee gedichten uitgezocht, die me doen denken aan jou en aan de fase waarin jij je nu bevindt. Ik ben erg benieuwd om te zien welk vervolg je nu zult kiezen, en ik wens je veel wijsheid en geluk toe in de komende jaren.

Lieve groet,

Floor



Foto: Maurice Hertog

Zo'n gelukkige dag.

De mist was vroeg gezakt, ik werkte in de tuin.

De kolibries stonden stil boven de bloeiende kamperfoelie.

Er was geen ding op aarde dat ik zou willen hebben.

Ik kende niemand die het benijden waard was.
Wat aan kwaad was geschied, had ik vergeten.
Ik schaamde me niet bij de gedachte dat ik was wie ik ben.
Ik voelde nergens in mijn lichaam pijn.
Toen ik mij oprichtte, zag ik de blauwe zee en de zeilen.

Czesław Miłosz (1972) Poolse dichter en nobelprijswinnaar



I have heard what the talkers were talking, the talk of the beginning and the end,
But I do not talk of the beginning or the end.
There was never any more inception than there is now,
Nor any more youth or age than there is now,
And will never be any more perfection than there is now,
Nor any more heaven or hell than there is now.

Walt Whitman (1855) Amerikaanse dichter

A Tale of Two Jans¹

Martin Stokhof

There is the formal Jan, there is the informal Jan, there is the Jan of the *Haskell Road to Logic Programming*, there is the Jan of *Filosofie: Een Inleiding*, there is the analytic Jan who wants nothing to do with Heidegger, there is the Jan who studies buddhism and practises aikido, there is the Jan at the CWI research centre, there is the Jan who once considered teaching philosophy to prison inmates, there is the Jan who writes highly specialised and technical papers, there is the Jan who writes books for high-school students and philosophy for the millions, there is the Jan who proves theorems, there is the Jan who paints and makes music, there is the Jan who organises scientific conferences, there is the Jan who is still restoring a farm house in France—in short, there is Jan van Eijck: there is his curiosity and enthusiasm, in exploring and investigating, in sharing and communicating, there is his relativism and self-irony, in evaluating and combining work & life, there is his inspiration and knowledge, shared with students and fellow scientists alike, and there is his optimism and trustworthiness, as a colleague and as a friend.

Two Jans, many Jans, but still one Jan: the one I am proud and happy to have known and worked with, and that I hope to know for many more years, in all his many guises.

Notes

¹. With sincere apologies to the estate of Charles Dickens. ↵

Personal Note

Elias Thijssse

I owe a lot to Jan. Few people have been more important for my academic career than Jan van Eijck. Although, due to our shared interest in Montague Grammar, we might have met earlier, my first clear recollection of Jan's influence was a course on recent developments in linguistics, given by him and Nico van der Zee at Groningen University in 1979 or 1980. One paper, written by Barwise and Cooper, by then only available in manuscript form, was explained by Jan in his typical well-informed and enthusiastic manner. Since I considered myself to be a die-hard Montagovian by that time, I was quite sceptical about this paper: it seemed partly a reformulation of Montague Grammar, partly an empirically weak account of linguistic universals some of which were tautological and some were incorrect. I expected a negative comment by Jan on my critical review, but he encouraged me to carry on. Looking deeper into the matter, the research on Generalized Quantifiers proved to reveal various new logical, mathematical and linguistics aspects. Moreover, the open culture and the intensive interaction (students, lecturers, professors discussing matters on an almost even terms) during the Groningen semantic wave of the eighties made it fun to work in this area. To cut a long story short, I managed to publish several papers and write a Master Thesis on Generalized Quantifiers, and Jan was certainly not to blame for me not getting a PhD scholarship for pursuing this research.

Before I finished studying, Jan left Groningen for Tilburg University and I became a maths teacher. But it was no coincidence that some years later, I followed Jan's professional move and became a research assistant in Tilburg. Only a few years later Jan left Tilburg again, now heading for SRI Cambridge and I succeeded Jan as teacher of several of his courses on logic, semantics and grammar. This also paved the way for converting Jan's lecture notes to a joint Dutch text book on logic for students in linguistics and computer science. Since I left academia some years ago to become a maths teacher again, contacts have dropped but good memories survived.

Unfortunately, I did not share Jan's enthusiasm in programming, so it may not be a coincidence that I have not been able to retrieve my old and fairly simple ALGOL, Pascal and Prolog programs for the present occasion.

Jan's good spirit and humour should be emphasized here. Jan's hospitality in Groningen, Tilburg, Cambridge and Amsterdam should be praised and one wishes many more colleagues to act in such a friendly matter. And I consider it a privilege that Jan asked my advice, also on rather personal matters, and even seemed to consider the advice valuable. Jan is very good person and I hope he can enjoy his retirement for many more years to follow.

Hup! Hup!

Christina Unger

```
-- 'But one must be careful to read Haskell code as one would read poetry'  
  
-- (William Cook, https://github.com/w7cook/AoPL)  
  
type Poem = [String] -- TODO Probably there's more to this...  
  
type Dank = String  
type Wens = String  
  
type VoorAlles      = Harte -> [Dank] -> Jan  
type VoorDeToekomst = Harte -> [Wens] -> Jan  
  
type Harte = [Int] -- slight simplification for the sake of implementability  
  
data Jan = Jan1 | Jan2 -- | ...  
-- 'Two Jans, many Jans, but still one Jan'  
-- cf. Martin Stokhof's contribution  
  
-- Dear Jan,  
  
jubel :: Maybe Poem  
jubel = let best = head  
        in Just $ replicate 10 $ best $ wishes  
        ++ take 2 (repeat "Hup! Hup!")  
  
wishes = "altijd geluk en plezier"  
        : joy ("life" ++ "rest")  
        : helemaal_niets_slechts  
  
joy = foldl (++) "" . words  
helemaal_niets_slechts = []  
  
-- Yours, Christina
```



Working with Jan in Four Movements

Johan van Benthem

I feel that I just met Jan van Eijck yesterday, in the turbulent revolutionary Groningen of 1976—but now, the editors of this festschrift are telling me that I have not kept track of time, and that we live 40 years later, our long hairs gone, close to retirement. Academic life is a sheltered cave. We Dutch are all Rip van Winkles if it comes to it.

Over those years, Jan turned from a student into a colleague, ally, and friend whose vigor and continued growth kept impressing me. I could write about his dissertation on quantifiers, his work on natural logic in natural language, his record in computational semantics and computational logic, in dynamic-epistemic logics, or in broad social software. I could write about his didactic gifts, leading to a string of textbooks, all the way to our most recent collaboration on the internet course "Logic in Action". I could write about his broad intellectual interests, from general philosophy where he wrote a widely used and often reprinted textbook in his early years to the austere world of computer programming in his mature period. I could write about his stylistic talents, from technical papers to engaging public dialogues. And finally, I could write about his leadership in the national research program "Semantic Parallels in Natural Language and Computation", the Education team of my Spinoza Award project, or the "Games, Action and Social Software" project at the Netherlands Institute for Advanced Studies NIAS. And all this from someone who, when I first met him, sported a luxuriant black hairstyle presaging Marx and Engels rather than Hilbert and Gödel.

This string of laudatory modalities is not just my own personal view, or that of a local circle. Even Jan's relatively short foreign stays left their traces. Just a few years ago I met the somewhat intimidating CEO of a well-known internet start-up in San Francisco who became positively warm and jubilant the moment Jan's name came up. They had worked together in Cambridge, and his appreciation had stayed ever since.

But for this festschrift, I want to look at something else, namely, the public record of my collaborations with Jan. What did we write together, and what became of it?

The first paper we wrote, called "The Dynamics of Interpretation", appeared in 1982 as the first item in the first volume of the new *Journal of Semantics*. 'Dynamics' is a popular term these days (it may already be past its prime, as youth turns into cliché), but we may have been among the first to use it. By that time, classical Montague Grammar was challenged by new richer semantic frameworks for natural language, and these frameworks were busy developing content and attracting followers. Instead of adding one more, Jan and I wanted to understand what was going on. In the paper, we mention a conference in Cleves that brought together leading innovators of the time like Jaakko Hintikka and Hans Kamp, but also, in another tradition, Pieter Seuren. In our paper, we try to bring logical clarity to these approaches, including then current views of discourse representations as schematic pictures of the real world and connections with semantic tableaux, and we explore consequences for logical theory. Specifically, we show that a picture metaphor can only work for a small existential fragment of a language: in more modern terms, the domain of model-checking methods for inference in knowledge representation. We also analyze to which extent semantic tableaux can be viewed as generalized pictures, with not just individual facts but also rules, and prove in which precise sense tableaux manipulate open branches as schematic representations of (classes of) standard models. Finally, we question the conservative tendency of much semantic innovation at the time of insisting on tight connections with complete classical models, and suggest that a more radical approach would just stick with partial models as one's semantic universe.

Looking back at these themes, we were too early. Pictures and diagrams as a vehicle for reasoning only became a big topic in the 1990s, and partial possibilities semantics in its own right did not sweep the world, although it is making a comeback these years. Moreover, we made a tactical mistake in giving an extensive comparison between Hintikka's game-theoretic semantics and Kamp's discourse representation theory and suggesting that these frameworks could learn from each other. This is of course anathema to founders of new religions that are on the march.

Despite this irenic tendency, there are also striking critical undertones in the paper. For instance, on the status of representations, we write "some view them as syntactic constructs, some as psychological ones, and some just prefer to remain confused on this issue." Nowadays, several decades later, I would never write anything like that, having learnt to sugarcoat criticism for thin academic skins, suppress emotion, count to 10, 100 or whatever power of 10 it takes, and realizing that every ironical remark will come back to haunt you. But oh my, how I enjoy the barbs in this paper!

All in all, I believe that methodological clarity and a cooperative stance are crucial for a healthy field: but one can overdo it. Competition between schools, including unfair propaganda, may well be a prime mover that we should not stifle.

Are there still things to think about for Jan and me? Our paper is about the dynamics of interpretation, not about dynamifying meanings, and this distinction in locus for capturing the dynamics of language use still seems relevant. I wonder what Jan thinks about it, decades later, as a much more experienced computational semanticist.

In the 1980s, our interests turned from natural language by itself to include action and computation in general. Computer science became a new source of inspiration, in its fast development of new paradigms for sequential and parallel computation. This fit is natural given the many parallels between the study of language and computation, a trademark item at the ILLC. Our second paper came some ten years later, written with Vera Stebletsova from Moscow. Again, our goal was comparative and systematic. We contribute to connecting two realms that were already interacting: the tradition of labeled transition systems in computer science, in particular process algebra, and polymodal and dynamic logic over relational models. We give a general perspective on the proliferation of process equivalences at the time in terms of matching logical languages, showing how classical results on definability go through. Going beyond that, we introduce the notion of safety for bisimulation, a logical take on program or process operations that fit a chosen invariance notion—though a full first-order characterization theorem came only later in the 1990s. And finally, we take parallel processing seriously by proposing a new notion of bisimulation for concurrent PDL.

I could say more (the paper is 45 pages long), but these points are of course mainstream by now. Computer science and modal logic have drawn ever closer, and qua methods, the paper is typical Amsterdam School in its emphasis on model theory as a source of generality. So, no story of unjust neglect here: we were just helping build a highway, and the computer science influence stayed in our own later work.

An open end for Jan to ponder today is our treatment of concurrency. Modeling parallel action is a very live issue – unless you think your religion has solved it all (sects also abound at the interface of logic and CS). With some colleagues, I am puzzling over natural bisimulations for games these days, and questions from our paper return.

But Jan and I did not always just streamline what was already happening: we also ventured into new territory. In a 1993 report "Changing Preferences", written together with Alla Frolova from Moscow (again a Russian logician, the Moscow connection was strong in those days), we point out how, in addition to action and information, preference is crucial to agency. True to form, we then propose a general logic of order minimization for analyzing various notions of consequence proposed at the time in AI and semantics of natural language, that combined information and preference (be it as the rather bleak notion of what would nowadays be called relative plausibility). We show how many further notions of consequence can be defined in our framework, demonstrating the free spirit that comes with logical abstraction. There is also an extensive discussion of dynamics of new information, and of technical translations for laws of dynamic scenarios into the language of static logics, in the spirit of things Jan and I were doing separately at the time for systems of dynamic semantics.

Again, this sort of approach is mainstream by now. However, the period is pre-DEL, and the technique of the paper is the one we used back then, inspired by the work of Edsger Dijkstra, the only Dutch winner of the Turing Award to date. We use transformations of propositions as sets of worlds, connecting pre-and postconditions recursively. Nowadays this would be seen as a 'lifted version' of DEL-style update, I guess, though the proposition-transformer approach is more general in principle – and it did feature prominently in the work of our joint student Marc Pauly. It is a long time since I have discussed this methodological shift with Jan, and I sometimes wonder why we gave up the Old Way (or did it just happen?). Of course, there must be an answer, since we are talking logic and mathematics, and if there is one, Jan is sure to have it.

But to me the most striking part of the report comes toward the end. We propose two operations that change preference order, that would nowadays be called 'suggestion' and 'radical upgrade', and suggest that these are central notions to study. But we did not. We did not even publish this paper. Why? Perhaps the reason can be found at the start of the paper: we apologize for taking preference seriously, as being subjective and fleeting (a shallow objection one still finds in some circles today). This was a mistake. Only some 10 years later, preference became crucial to my own work on logic and games, and I am sure that Jan, too, has seen the light. And our operations for preference change turned out to be central to the creation of dynamic-epistemic preference logic in the work of Fenrong Liu around 2006. So, we missed our chance.

But to those who stay afloat, the great stream of research always carries fresh opportunities. The emergence of dynamic-epistemic logic in our community around 2000 affected us deeply, as it was an ideal vehicle for pursuing the dynamic and computational themes that had been there in our work from the start. Here, too, a modeling challenge plus a search for systematization combined to produce my final example, the 2006 paper "Logics of Communication and Change", written with Barteld Kooi.

This paper arose from the needs of group knowledge, and in particular, finding the right recursion laws for common knowledge after update. I had already found one solution for the special case of public announcement logic: extending the base logic with conditional common knowledge, and next, with Barteld, I had found a solution for the much harder case of DEL product update in terms of finite product automata. However, joining forces with Jan made us see the power and elegance of making two further changes in the set up: working with an epistemic version of propositional dynamic logic in the base, and using a beautiful algorithm for finding recursion laws based on the proof of Kleene's Theorem for finite automata. In addition, we showed how factual change can be taken on board in DEL, as long as it can be modeled by changing truth values for proposition letters according to some definable recipe.

I believe that this is still about the most elegant formulation of DEL in its generality, though some friends feel we either did too much (the base language becomes hyper-baroque) or too little (we did not arrive at generalized dynamics in the style of Girard, Liu & Seligman, and we did not cover the modal μ -calculus, something I did later with Daisuke Ikegami). I do not think our paper got the attention it deserved, and in fact, in the years since 2006, I have seen many publications on DEL reinventing the wheel (or even worse, square or polygonal versions thereof) that do not seem to be improvements. However, truth is a slow but sure traveler, and our time will come.

That does not mean that our paper is the last word: the road goes on. Right now, I am interested in fixpoint logics where DEL programs can be defined by recursion (for *cognoscenti*: the way things happened in PAL*), and then we need to spring the bounds of all this. However, these logics are highly complex computationally, at least in their current versions, so I am not sure if the programmer in Jan will approve.

These four papers show what working with Jan van Eijck produces. I believe that their methodology and content are still alive, with new questions if you add up all asides in the above. But the process was as pleasant as the product. Here are some qualities one experiences when working with Jan. He likes to be broadly informed across a topic, and

then shed light by seeing patterns across notions, results, and schools. He likes to ascend to well-tested logical abstraction levels where things become clear and comparable, rather than engage in ad-hoc modeling. And in his approach to both ideas and the people creating them, he tends toward generosity, and collaboration.

Of course, Jan also has habits that are more alien to me, such as his insistence on practical programming skills. These are like prowess in jogging: I admire people who do it, but I find it hard to follow. But I do marvel at how the programmer in Jan turns abstract logic into concrete methods, say, when he computed recursion laws for difficult communication scenarios that I could never keep straight with just brain power. And I have seen with my own eyes how adding programming to logic courses, another recommendation of his, can turn hostile mobs into engaged students.

Jan's academic road has many more milestones than the four I have placed in this short piece, but I am sure that this festschrift in total will reveal a long scenic road.

It will be clear that Jan and I have been travel companions for a long time. There are many more personal things one could say about success and failure, hope and fear. But not everything that is announced is valuable, and not everything that is valuable should be announced. I wish Jan all the best with designing a meaningful life beyond retirement, avoiding the trap he dreads of 'more of the same'. Now is the time!

References

J. van Benthem & J. van Eijck, 1982, 'The Dynamics of Interpretation', *Journal of Semantics* 1:1, 3–19.

J. van Benthem, J. van Eijck & A. Frolova, 1993, 'Changing Preferences', Report CS-R9310, Center for Mathematics and Computer Science CWI, Amsterdam.

J. van Benthem, J. van Eijck & V. Stebletsova, 1994, 'Modal Logic, Transition Systems and Processes', *Journal of Logic and Computation* 4:5, 811–855.

J. van Benthem, J. van Eijck & B. Kooi, 2006, 'Logics of Communication and Change', *Information and Computation* 204:11, 1620–1662.

The Value of Alternative Semantics

Tijs van der Storm

Abstract

Starting his return to philosophy, Jan wrote an interesting blog about alternative facts. In the article he writes that "Alternative facts do not exist". A statement is considered a fact, when it corresponds to reality, otherwise, it's a falsehood (or a lie if dishonesty is involved). In a certain sense this is a matter of dependency: a statement's being a fact *depends* on the world, but the reverse is not true: you can have different states of the world or different possible worlds compatible with the same facts. In programming language design, there seems to be a similar dependency between semantics on the one hand, and syntax on the other: it's not possible to have different kinds of syntax for the same semantics since semantics is defined in terms of syntax. It *is*, however, very well possible to have different (imagined) semantics for the same syntax. In short: alternative syntax does not exist, but alternative semantics do.

In this essay I'd like to explore the value of alternative semantics for the benefit of the programmer. Value here has two meanings: value as in "useful", or "a good thing", but also value in the functional programming sense, where values are immutable data objects resulting from evaluating expressions. The idea is to partition a language according to different but related concerns, and then have different interpretations for the whole language, where each interpretation addresses only one concern. I'll illustrate with a simple domain-specific language (DSL) for Web-based user interface programming, called TwoStones^{TwoStones}.

All the code can be found online here: <https://github.com/cwi-swat/TwoStones>.

Introduction

The "Right View"

The idea of using two interpretations to untangle what's going on in typical web-based UIs is inspired by Evan Czaplicki's work in *Elm*. *Elm* is a functional programming language that separates a UI app into two parts: a view function that takes an immutable application model and produces an HTML tree structure for rendering, and an "update" function, interpreting events into a (functional) modifications of the application model. These functions are called alternatingly, providing a clean model of stepping through an application.

Recently, I've ported the *Elm* architecture to the *Rascal* language, resulting in the *Salix* library. Both update and view functions can be written in *Rascal*, but the UI is executed in the browser. Observing the amount of boilerplate I had to write to evolve a model in the update function, I revisited an idea originally applied in the *Ensō* system and *Recap*: instead of having two *functions* for render and update, we'll use a single program, but with two *semantics*, effectively turning the *Elm* model inside out. As a result, the programmer writes a single "function" addressing both concerns at once, but the runtime will separate these concerns in different executions of the program. My goal here is to show how this can be achieved without losing the benefits of functional programming.

To make this more concrete, here are two function signatures describing the two semantics. The first is "rendering":

```
render : Program × Model → Node
```

So the render semantics takes a program, an application model, and produces a HTML node, which can subsequently be rendered in a browser.

The handling concern is captured by the following signature:

```
handle : Program × Model × Event → Model
```

In this case, the function takes a program, a model, and an event value, and produces an updated model.

A UI program then consists of an (infinite) top-level driver loop calling `render` and `handle` in alternating fashion, like this:

```

view_0 = render(p, m_0);
... event e_0 happens ...
m_1 = handle(p, m_0, e_0)
view_1 = render(p, m_1)
... etc.

```

The sequence starts with rendering an initial model `m_0`. Then some event `e_0` happens, which is fed into `handle`. The function `handle` is invoked with the original model (`m_0`), and produces the new model (`m_1`). The new model is then rendered producing the new view `view_1`, and the cycle repeats.

(Aside: this might look very expensive in terms of computation; however, rerendering the whole GUI "virtually" and then patching the actual GUI based on the difference ("diff") between `view_{i+1}` and `view_i` is what state of the art frameworks like Facebook React, Om, and Elm all do. The approach represented here is fully compatible with this technique.)

As a result, the programmer only writes a single program, but may still enjoy the benefits of functional programming with immutable values. Next I'll describe a simple prototype language, called TwoStones in more detail. Basically, we'll be filling in the definitions of both `render` and `handle`.

Rendering Views with TwoStones

Let's start with the first concern: constructing the visual presentation of the user interface, or: rendering. In the following I'll use the `Rascal` to present the abstract syntax of TwoStones, as well as the definitions of standard interpreters to represent the semantics. Most code snippets will be easy to follow for anyone familiar with functional programming idioms; where needed, I'll explain Rascal specific constructs.

The core of TwoStones is captured in the following abstract syntax definition of statements (`stm`):

```

data Stm
  = elt(str name, Stm body)
  | ifThenElse(Expr cond, Stm then, Stm els)
  | forEach(str x, Expr lst, Stm body)
  | output(Expr text)
  | block(list[Stm] stms);

```

The `elt` statement produces an HTML element node, tagged with `name` and containing child nodes produced by `body`. The construct `ifThenElse` is used for conditional evaluation. To loop through a list, one can use `forEach` which iterates over the elements, and binds each element to the variable `x` in scope of `body`. The leaves of the node tree are represented by text nodes, produced by the `output` statement. Finally, statements can be grouped using the `block` construct.

For now we assume we have a standard expression language (`Expr`) which is used in `ifThenElse`, `forEach` and `output`. We assume further that there's an `eval` function to evaluate expressions to values of type `value`. Values include primitives (`int`, `str`, `bool`; wrapped in constructor `prim`), as well as lists (`array(list[value])`) and records (JSON-like sets of properties; `record(map[str,value])`). The expression language contains constructs for accessing list elements by index (`x[i]`) and fields of records (`x.f`), and the usual operations on booleans, integers, and strings.

The result of rendering a `TwoStones` statement will be a `Node`, which represents a simplified version of HTML. For now, node attributes are omitted, except for a map associating identities (integers) to event types (strings). This `events` map is a *keyword parameter* which makes it optional; retrieving the keyword parameter from a constructor will return the default value if it hasn't been explicitly set. Here's the definition of `Node`:

```

data Node
  = element(str name, list[Node] kids, map[int,str] events = ())
  | text(str text);

```

The `render` signature introduced above takes a model value and a program and produces a `Node`; this is the signature of the main entry point as shown earlier. The `render` function I'll introduce next has a slightly different signature, consuming a statement, and

environment (containing the model), a parent `Node`, and an identity generator function `next` (which will be explained below). The `parent` argument allows nested statements to append new nodes to their enclosing element.

Here's the definition of `render` for TwoStones statements:

```
Node render(elt(str name, Stm body), Env env, Node parent, int() next)
  = parent[kids = parent.kids + [n]]
  when
    Node n := render(body, env, element(name, []), next);

Node render(output(Expr e), Env env, Node parent, int() next)
  = parent[kids = parent.kids + [text("<v>")]]
  when
    prim(value v) := eval(e, env))

Node render(ifThenElse(Expr cond, Stm then, Stm els), Env env, Node parent, int() next)
  = render(v ? then : els, env, parent, next)
  when
    prim(bool v) := eval(cond, env)

Node render(forEach(str x, Expr lst, Stm body), Env env, Node parent, int() next)
  = ( parent | render(body, env + (x: v), it, next) | Value v <- vals )
  when
    array(list[Value] vals) := eval(lst, env);

Node render(block(list[Stm] body), Env env, Node parent, int() next)
  = ( parent | render(s, env, it, next) | Stm s <- body );
```

Some notes on Rascal notation: `:=` represents pattern matching, postfix assignment `t[x = e]` is used to functionally update a component of a tuple or constructor; the construct `(init | accu | gen)` is sugar for `reduce`, where `init` is the initial value, `accu` the accumulator, and `gen` represents an arbitrary sequence of generators, like in list comprehensions. In each iteration the intermediate result is bound to the keyword `it` in the context of `accu`.

Rendering the `elt` statement simply renders the `body` statement with a new parent node, and the result is added to the current parent. Same for `output`, except that the argument expression is converted to a string (using Rascal's string interpolation) and

wrapped in a `text` node. Note that only primitive values can be output. Conditional execution selects either `then` or `else` based on the result of evaluating `cond`. To render each element of a list individually, the `forEach` statement accumulates child nodes through the use of a "reducer" by executing the `body` of the loop in the context of a new environment binding `x` to consecutive values of the list `lst`. Finally, the `block` statement executes each statement in its body, appending child nodes to the current `parent`.

The definition of `render` can be used to produce passive nodes from `TwoStones` programs, but what about interaction? This is where the `next` function starts to be relevant. First, however, we extend the syntax with one more statement to specify the effect of an event:

```
data Stm
  = ...
  | on(str kind, Expr x, Expr v);
```

The `on` statement states that when an event of type `kind` happens (where `kind` could be "click", or "change", etc.), it should update the location in the model designated by expression `x` to be the value of `v`. In other words, the `on` statement associates an event occurrence to some update of a sub tree of the model. How this is realized is outside the scope of `render`. Nevertheless, in order for `handle` to know which effect to apply when an event occurs, `render` leaves a small trace in the node structure of UI, by associating a unique identity to the node surrounding the `on` statement, as well as the type of event:

```
Node render(on(str kind, Expr _, Expr _), Env env, Node parent, int() next)
  = parent[events = parent.events + (next(): kind)];
```

So the only thing `render` does, is getting the next id, and linking it to the event type in the `events` map of the surrounding element. The identity will later be used in `handle` to identify where in the node tree a particular event occurred.

The `next` function is a stateful closure that gives a unique integer on every invocation. Purists might consider this to be a violation of functional programming, but I could have threaded an integer through the whole computation just as easily, at the cost of additional

boilerplate. Another way to simulate node identity is to use a canonical representation of the execution trace up till the point it is needed (i.e., a path).

Time to look at an example: a simple counter app. This app shows an integer value with two buttons labeled "▲" and "▼" respectively. Clicking on those buttons increases resp. decreases the counter value. The application model is a simple integer, and we'll assume it resides in the variable "model".

Here's the AST representing the counter app:

```
elt("div", block([
  elt("button", block([
    on("click", var("model"), add(var("model"), integer(1))),
    output(string("▲"))
  ])),
  output(var("model")),
  elt("button", block([
    on("click", var("model"), sub(var("model"), integer(1))),
    output(string("▼"))
  ]))
]))
```

Rendering this program produces the following node structure:

```
element("div", [
  element("button", [text("▲")], events=(0: "click")),
  text("0"),
  element("button", [text("▼")], events=(1: "click"))
])
```

Such node values can be converted to HTML and rendered in the browser. For instance like this:



Each element that has an `events` map will have Javascript event handlers for each kind of event in the map. These handlers are all of the same type: they return an object capturing the `id`, the type of event, and any additional data (e.g., the text entered in a

text box). In turn, these event objects are then fed into the event handling process, which I'll describe now.

Handling Events and Updating Models

As seen above, the `render` semantics simply skips any event-handling logic, except for leaving a tiny trace in the view so that the `handle` semantics knows where in the node tree an event has occurred. This context is found by having `handle` execute the exact same control-flow as the `render` code, when given the same model value (see the trace above), and providing it with a fresh id generator as `next`. Because the control-flow is exactly the same and TwoStones programs are purely functional, the function `next` will be invoked in the exact same context as `render` called it. As a result, if a generated id matches the id of the event given to `handle`, we'll know we're at the right place.

The `handle` function not only consumes a TwoStones statement, an environment and an id generator, but also an event object. Events carry the id of the originating element and their kind; they are defined as `data Event = event(int id, str kind)`. For now we abstract from the actual additional data that events may carry, such as entered text, or mouse coordinates.

Handling an event results in a list of zero or one updated `value`; I'm using `list`s here as the Optional monad so that I can use Rascal's splicing (using `*`) in combination with list comprehensions to simulate the monadic operations `bind` and `return`. When `handle` returns the empty list, the event was not handled; otherwise, the single element in the list is the updated version of the model.

Here's the definition of `handle`:

```
list[Value] handle(elt(str name, Stm body), Env env, Event event, int() next)
= handle(body, env, event, next);
```

```
list[Value] handle(ifThenElse(Expr cond, Stm then, Stm els), Env env, Event event, int() next)
= handle(v ? then : els, event, next)
when
  prim(bool v) := eval(cond, env);
```

```
list[Value] handle(forEach(str x, Expr lst, Stm body), Env env, Event event, int() next)
= [ *handle(body, env + (x: v), event, next) | Value v <- vals ]
when
  array(list[Value] vals) := eval(lst, env);
```

```
list[Value] handle(block(list[Stm] body), Env env, Event event, int() next) {
= [ *handle(s, env, event, next) | Stm s <- body ];
```

```
list[Value] handle(output(Expr e), Env env, Event event, int() next) = [];
```

The code looks very similar to the code of `render`. Note, however, that this time the `output` and `elt` constructs are skipped, since they should not contribute to the result. Conversely, however, the `on` statement is now the place where the actual work is done:

```
list[Value] handle(on(str kind, Expr x, Expr v), Env env, Event event, int() next)
= [ ... | event.id == next(), event.kind == kind ]
```

Here, we check if the event that occurred (`event`) is of the right type and that it is the event that should be handled at this point of execution via the `next` function and the event's `id`. But what should be done at the ellipsis in this code? Somehow, the location designated by `x` needs to be updated to the new value `v`, but how should this be propagated up to the top-level to obtain a complete new model? The answer is *zipper*.

Originally invented by Gérard Huet^{Zipper} (but at a time that Henry Lieberman was exploring similar ideas^{Marcottage}), a zipper represents a sub element of a data structure that is somehow in focus, together with functions to update this element in its context and

to move around further in the data structure. Zippers are widely used in functional programming (cf. [HaskellWiki](#))—below I'll describe a simplified, "untyped" version of the zipper, dubbed "cursors", to avoid buying into too much detail of Huet's original.

Cursors to the Rescue

A cursor can be represented using the following type aliases:

```
alias Put = Value(Value);
alias Cursor = tuple[Value get, Put put];
```

A cursor thus encapsulates a value indicated by the `get` component of the tuple, and a function to "update" this value in its context, using the `put` component of the tuple. How this works will be clear when discussing how values are traversed during evaluation. But first, we have to change the standard `eval` function and `Env` data type to reflect that we won't be processing bare values, but actually values of type `cursor`. So we'll assume `Env` is a map from string to `Cursor`, and `eval` returns `Cursor` instead of `Value`. (Note that `render` defined above could also reuse this new `eval`, by simply always projecting out the first component of a cursor, and binding values with the identity function as `put` in the environment.)

The relevant expressions that traverse data are indexing of a list using a subscript index (e.g., like `x[i]`), or accessing a field of a record value (like `x.f`):

```
data Expr
= ...
| field(Expr rec, str name)
| index(Expr lst, Expr idx);
```

The `eval` function for these cases:

```

Cursor eval(field(Expr rec, str name), Env env)
  = <props[name], Value(Value v) { return put(record(props + (name: v)));
  }>
  when
    <record(map[str,Value] props), Put put> := eval(rec, env);

Cursor eval(index(Expr lst, Expr expr), Env env)
  = <vals[idx], Value(Value v) { return put(array(replaceAt(vals, idx, v)
  )); }>
  when
    <array(list[Value] vals), Put put> := eval(lst, env),
    <prim(int idx), _> := eval(expr, env);

```

Some explanation is needed here. In the first case, evaluating the expression `rec` results in an record value cursor: the left component indicates the actual value of the record, the right component (`put`) represents its context. The result of accessing the property `name` is then a cursor consisting of the actual property value, and a `Put` function to "update it". Updating the property in its original context with a new value `v` entails updating the `props` map with the new value, and calling the `put` function (captured in the `when` - clause) of its original context. The same mechanism applies to list indexing: the `put` component updates the indexed element in the original list (using `replaceAt`), and then calls the parent `put` function to propagate upwards.

So cursors can be interpreted as the functional equivalent of a pointer to a location in memory: traversing an object moves the pointer along; calling the the `put` function updates the location pointed to. Of course, cursors do this in a purely functional way: calling `put` produces an updated version of the data structure you started with at the top of the program.

It's now clear what needs to be done when handling an event:

```

list[Value] handle(on(str kind, Expr x, Expr v), Env env, Event event, in
t() next)
  = [ eval(x, env).put(eval(v, env).get) | event.id == next(), event.kind
  == kind ];

```

Updating the model thus boils down to calling `put` on the left expression, with the value of the right expression. This will return a new value, that reflects the updated model.

Unfortunately, there's one issue we've overlooked: since the `handle` case for `forEach` simply binds elements of the list to the loop variable, we've lost the connection to the container list in the cursor. As a result, updates to elements will not be reflected back into the list, and from there on up. So as a final (essential!) tweak, the `forEach` implementation needs to be modified as follows:

```
list[Value] handle(forEach(str x, Expr lst, Stm body), Env env, Event event, int() next)
  = [ *handle(body, env + (x: subscript(vals, idx, put)), event, next) |
    int idx <- [0..size(vals)] ]
    when
      <array(list[Value] vals), Put put> := eval(lst, env);

Cursor subscript(list[Value] vals, int idx, Put put)
  = <vals[idx], Value(Value v) { return put(array(replaceAt(vals, idx, v)
  )); }>;
```

So instead of simply binding successive values, the code now simulates `index`-based access to list elements. The cursor returned by the helper function `subscript` propagates changes to the element, back into the container list, and upwards from there using `put`.

Counter App

Recall the output of the counter app above:

```
element("div", [
  element("button", [text("▲")], events=(0: "click")),
  text("0"),
  element("button", [text("▼")], events=(1: "click"))
])
```

Let's assume further that the initial counter model is 0, and stored in the environment as follows:

```
Env env = ("model": <prim(0), Value(Value v) { return v; }>);
```

This means, that the initial cursor simply returns the new value for the model. Now let's say the user has clicked the increment button (▲), which is represented as the event `event0 : event(0, "click")`. At the top level, this event is then handled using the following code:

```
handle(counterApp(), env, event0, idGen());
```

The result of evaluating this code will be `[prim(1)]`, as expected. This very simple example does not involve model navigation, so it doesn't really show the power of cursors in action. The example in the next section, however, fully exploits cursors in a user interface for recursive data.

A More Elaborate Example: Recursive Data

Up till now we've only looked at simple statements and control-flow using conditional and `forEach`. However, the idea is easily extended to support functional abstraction, and even higher-order functions. The key point is that computed values carry their original context with them, so whenever the model is traversed, it doesn't matter in what ways a value ends up in the left-hand side of the `on` construct; an update will also be reflected on the *whole* model.

Let's say we'd like to define an editor for labeled tree structures, with boolean values at the leaves. This type could be defined as follows:

```
data Tree
  = node(str label, list[Tree] kids)
  | leaf(bool val);
```

In the dynamically typed TwoStones language, values of this type would be represented using records and lists. Here's a simple example in JSON-like notation:

```
{label: "Top", kids: [
  {label: "Sub1", kids: [
    {val: true},
    {val: false}
  ]},
  {label: "Sub2", kids: [
    {val: false}
  ]}
]}
```

Here's an example of two view functions to render such trees. (Note for the sake of reability, we use a concrete pseudo syntax here; it should be easily mapped to the abstract syntax defined above, with additional constructs for function definition and invocation):

```
def treeView(tree)
  output(tree.label)
  elt("ul") for (kid: tree.kids)
    itemView(kid)
end

def itemView(item)
  elt("li") if (item.kids)
    treeView(item)
  else
    elt("button") {
      output("toggle")
      on("click") item.val := !item.val
    }
end
```

The `tree` function starts by rendering the tree's label, and then unfolding the list of children below an unordered list element (`ul`). Each child is rendered using `itemView` . The function `itemView` creates a list item element (`li`) and then checks whether this node has any children, and if so, recursively calls `treeView` . If the current node does not have children, it is a leaf, and a button is rendered to toggle the boolean value attached to it.

Because cursors maintain a relation between a rendered element (and its `on` handler) and the position in the model, a click event on one of the toggle buttons will update the appropriate, corresponding leaf value.

Composing Views

The previous section highlighted an important benefit of TwoStones' approach to view maintenance: the use of functional abstraction to organize view programming. The tree example exploited mutually recursive functions to render recursive data. But functional abstraction can be put to good use for other common use cases as well. Some of these are illustrated below.

Different Views for the Same Model Element

Let's say we have two functions for creating different views for the same kind of data. For instance, representing a number as a text field or using a slider. They can be simply combined by calling them on the same model expression:

```
textboxView(model.value)  
sliderView(model.value)
```

Since TwoStones handles only a single event at a time, an update to the model will result from *either* `textboxView`, *or* `sliderView`. And since we're rerendering after every update, there's no possibility that either view will ever display data that is out of date.

The Same View for Different Model Elements

Alternatively, sometimes an application will have multiple instances of the same type of data, all to be represented using the same view. For instance, a shop application might store both an invoice address and a shipping address. Showing both addresses can be realized as follows:

```
addressView(model.invoiceAddress)  
addressView(model.shippingAddress)
```

In this case, since the cursors resulting from `model.invoiceAddress` and `model.shippingAddress` will be different, events resulting from different `addressView` renderings will update only the corresponding parts of the model. In other words: changing the invoice address will not affect the shipping address, and vice versa.

Higher-Order Views

It's also possible to define higher-order views, which consume other view functions. For instance, let's say we want to define a generic list-view: it will take care of rendering and updating the list. However, to abstract from the type of elements in the list, we'd like this view to be parameterized by another function, dealing with element-level rendering and handling.

For instance, like so:

```
def eltView(x)
  ...
end

def listView(lst, view)
  elt("ul") for (x: lst)
    elt("li") view(x)
end

listView(model.list, eltView)
```

Again, the use of cursors instead of plain values traveling through the data flow of the program ensures that updates resulting from each respective invocation of `eltView`, will update the correct element in the list.

All three examples illustrate an extremely powerful way of component-based UI development, where good old functional abstraction is used to define highly reusable UI components. This is not merely cute: it means that UI programming in this style *scales*.

Discussion

Assessment

Although the syntax of `TwoStones` might "smell" imperative through the use of loops and sequencing, `TwoStone` programs are still purely functional: there is no way to mutate data or re-assign variables. In a certain sense, the syntax represents a light syntactic sugar over monadic folds over some data type, producing different values in `render` and `handle`, respectively.

One benefit is that there's no boilerplate resulting from encoding such folds in a general purpose language; this is all encapsulated in the semantics instead. Furthermore, the code is clearly structured like the `render` output it produces. The simple control-flow is easy to understand and debug, much like plain, procedural code, but without the side-effects.

The use of cursors in the semantics of `handle` enables that functional abstraction is sufficient for rich patterns of view composition, both first-order and higher-order. In contrast, when nesting view components in `Elm`, the messages originating from the child component need to be "mapped" into messages of the parent component, so that the user-provided `update` function that interprets these messages can route them to the appropriate (sub) element of the model. None of that is needed in `TwoStones`: events will automatically be routed to the right `on` handler, which will update the right model value.

Meanwhile, the fact that `TwoStones` programs consume and produce only immutable values, makes advanced time-travel debugging features easy to implement: simply store successive model versions, and the events that produced them, and you'll be able to go back and forward in time. It's also trivial to visually highlight the activated event handler, by simply locating the appropriate `id` in the rendered node.

One could argue that a limitation of `TwoStone`'s approach is that it's limited to dealing with one event at a time, one update at a time, and one model value at a time. It's not possible to batch up events, and handle them all at once. Likewise, it's not possible to update different parts of the model while dealing with a single event. Finally, the model value returned from `handle` (if any) can only be *the* model value, because there is no way to know to which model it would belong, if the process would start with more than one model to begin with. Nevertheless, these limitations do not significantly reduce expressivity of `TwoStones`.

Related Work

`TwoStones` combines many existing ideas, often in simplified form. If there's anything novel, it lies in the combination of these ideas, rather than the ideas themselves. I'm very much indebted to William Cook for many ideas presented here, which have evolved from numerous discussions on web programming in the `Ensō` system^{Ensō}, the full potential of which has not yet been realized.

Multiple semantics are employed in the area of "two-tier" computations. For instance, *batches*^{Batches} allow application code to be partitioned in code that will be executed on a client, and code that executes remotely (e.g., in an SQL server). Similar partitioning is applied in the context of web programming, where some code executes in the browser, while other parts run on the server (e.g., as in *Hop*^{Hop}).

The use of cursors is further inspired by David Nolen's initial version of *Om*, which presented an immediate mode UI framework on top of Facebook React, with an emphasis on single-value, immutable application state, just like described here. Cursors themselves originate from Huet's zippers and are related to functional lenses^{Lenses}. Many of these ideas can be traced back to Meertens' seminal work on constraint maintainers^{Meertens}, developed partly at CWI in the context of the Views system^{Views}. As far as I know, Greenberg's honor's thesis is the only work directly applying lenses in the context of UI programming^{Greenberg}.

Further Directions

The TwoStones prototype described in this article only scratches the surface in terms of what kinds of "back propagation" through `put` is possible. Apart from field selection on records and indexing and looping on lists, cursors could support many other operations, as long as they can be meaningfully "reverted". In the scalar case, the programmer can simply invoke whatever "inverse" function is needed in the last argument to `on`. For instance, the programmer would call `fahrenheit2celsius` if the model stores temperature in celsius but the user edits in fahrenheit. If you apply an operation, for instance, on lists, however, and then iterate through it using `forEach`, we'd like to maintain the origin relation in the cursors of each of the elements.

As an example:

```
for (x: reverse(lst))
  # do something with x ...
```

In this case, evaluating the `reverse` application could be implemented as follows.

```

Cursor eval(apply("reverse", Expr e), Env env)
  = <array(reverse(vals), Value(Value v) { return put(array(reverse(v.val
s))); }>
  when
    <array(list[Value] vals), Put put> := eval(e, env);

```

Whenever the `put` component is invoked, the list is simply reversed back to represent the original order. Other examples include, projecting out a sub-list of a list (useful for pagination) or list permutation (for instance, if you don't want to sort the model list itself, but only present it according to some order). Often, the cursor needs to maintain additional information to faithfully realize `put`, especially when an operation is lossy (like projection). In the implementation of field access on records and list indexing shown earlier, this is realized by having the `put`-closure capture the necessary values from the "old" value.

Conclusion

"Alternative facts do not exist". Yet, as Jan is well aware, there is value in possible worlds. Design is about exploring such possible worlds, and programming language design is no different. I hope I have succeeded in showing that defining alternative semantics for languages can sometimes clarify and simplify complex programming tasks. The simple TwoStones user interface DSL is supported by two semantics, one for rendering, and one for event handling. The first one constructs the visual presentation of the user interface, whereas the second one updates the application model whenever an user event occurs. Applying these alternative interpretations in alternating fashion, isolates these two concerns, and eliminates a significant amount of boilerplate code, without sacrificing the benefits of functional programming.

Notes

^{TwoStones}. "TwoStones" refers to the fact that it allows you to "kill one bird with two stones"... ↩

^{Enso}. Alex Loh, Tijs van der Storm, William R. Cook, *Managed data: modular strategies for data abstraction*. Onward! 2012: 179-194 ↩

Lenses

Lenses¹. Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, Benjamin C. Pierce. *Matching lenses: alignment and view update*. ICFP 2010: 193-204 ↩

Batches². William R. Cook, Ben Wiedermann, *Remote Batch Invocation for SQL Databases*, DBPL 2011. ↩

Greenberg³. Michael Greenberg, *Declarative, composable views*, Honor's thesis, 2007, Brown, [pdf] [code] ↩

Zipper⁴. Gérard P. Huet, *The Zipper*, J. Funct. Program. 7(5): 549-554 (1997) ↩

Meertens⁵. Lambert Meertens, *Designing Constraint Maintainers for User Interaction*, 1998, <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.3250> ↩

Views⁶. Steven Pemberton, *The Views application environment*, CWI Technical Report CS-R9257, 1992. ↩

Hop⁷. Manuel Serrano, Erick Gallesio, Florian Loitsch, *Hop: a language for programming the web 2.0*, OOPSLA Companion 2006: 975-985 ↩

Marcottage⁸. Henry Lieberman, *Marcottage: A Navigational Approach to Object Networks*, 1997, unpublished draft:

<http://web.media.mit.edu/~lieber/Lieberary/OOP/Marcottage/Marcottage.html>.

Ironically, Lieberman wrote this while he was in Paris, when Huet was there and developed the zipper structure. ↩

Rambling with Minkowski

Stijn van Dongen

Around October 1995 I started my PhD under supervision of Jan van Eijck and Michiel Hazewinkel at the CWI, the Dutch Centre for Mathematics and Computer Science in Amsterdam, finishing it in May 2000. Many are the happy memories from that period, and Jan played a large part in these. You are an inquisitive scientist, always ready to establish and show connections and work out new ideas in a number of different fields. I could see this easily even from my vantage point, which was less than ideal as my background had only little overlap with the realms of logic, language and computation where you roam freely. During the first year of my PhD I followed courses with you in Utrecht to widen my horizon (it needed much widening). You are also an engaging and inspiring educator. I vividly remember the course where small fragments of Haskell kept popping up and effortlessly developed into ever more sophisticated mathematical and computational machinery. Other happy memories include the dinners you organised with Heleen, which always had a welcoming and lively atmosphere with an interesting crowd of people.

I must admit that I never picked up Haskell, even though its appeal is clear. This may be partly because of a lack of suitable projects, partly because monads scare me, but more likely my mind is not quite functional and tends to ramble. To keep it in check, I use imperative languages, with an affinity for low-level specimens such as C and PostScript. Over the years I have regularly returned to writing little bits of (quite simple) PostScript, often to experiment with visualisations of tessellations, spirals, fractals, deformations, and combinations of these. A somewhat unlikely conviction has taken hold of me, namely that PostScript is a perfectly good vehicle for engaging children in programming. If the child is interested in programming to begin with, then for some the response to simple PostScript files immediately producing interesting patterns is very gratifying, more so than text scrolling in a terminal, my otherwise preferred display medium. These days we have many other languages aimed at young or novice programmers. Two that I am somewhat familiar with through my own children are Scratch and Processing, and we highly recommend both. These are visually oriented languages—another popular language in the household is Python, which began life at the CWI.

In this article, I will explore the Minkowski fractal in a fairly rambling manner. It will incorporate simple bits of linear algebra, rewrite systems and logarithmic spirals, and is best thought of as recreational mathematics. PostScript is used throughout with a little bit of Python at the end to verify some numerical identities. A Planck limit to a Minkowski spiral is mooted with the question as to what might lie inside it.

Returning to PostScript, it has a delightfully simple syntax, entirely dictated by a stack. To compute the Euclidean distance from the origin to some point, say the point (44, 117), you would enter (for example interactively with `gs`)

```
44 2 exp 117 2 exp add sqrt pstack
```

Here `exp` , `add` , `sqrt` and `pstack` are all PostScript operators, and the last one displays what is on the stack. With the `gs` program, it should display `125.0` . In the remainder of this article I will show various bits of PostScript and each of these is nearly self-contained. They all require the following preamble starting the file.

```
%!

28.3464567 dup scale           % change to metric, centimeters.
/pageheight 29.7039 def
/pagewidth 20.9903 def
pagewidth 0.5 mul pageheight 0.5 mul translate    % origin in centre.
```

To draw a simple square, one would use a few graphics operators:

```
newpath
0.1 setlinewidth
-1 -1 moveto      % This is now our current point.
0 2 rlineto       % 'relative line to'; relative to the current point, wh
ereas
2 0 rlineto       % lineto uses coordinates in the current coordinate sys
tem.
0 -2 rlineto
closepath         % closepath takes us back to the start of our path
stroke           % this draws the outline; 'fill' would fill the shape.
```

We change our square drawing code by specifying the midpoint and the side length, we make it a procedure and add some comments.

```

/draw_square {                               % our procedure, expects length x y on the
stack.
  newpath moveto                             % stack: 1 (length)
  dup 0.5 mul dup rmoveto                    % stack: 1           position top right
  dup -1 mul dup 0 rlineto                  % stack: 1 -1         position top left
  0 exch rlineto                             % stack: 1           position bottom left
t
  0 rlineto                                  % stack: _ (empty)   position bottom right
ht
  closepath
  stroke
} def

```

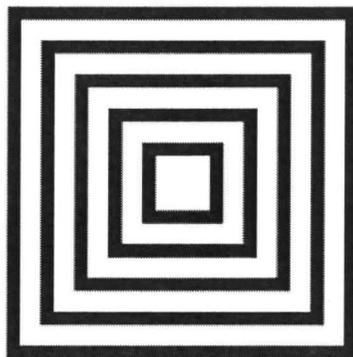
At this point, we include the preamble from the top, the code above for `draw_square` and this code:

```

0.1 setlinewidth
1 1 5 { % this will put the numbers 1 2 3 4 5 successively on the stack
  2 div
  0 0 draw_square
} for
showpage

```

We then obtain this image.

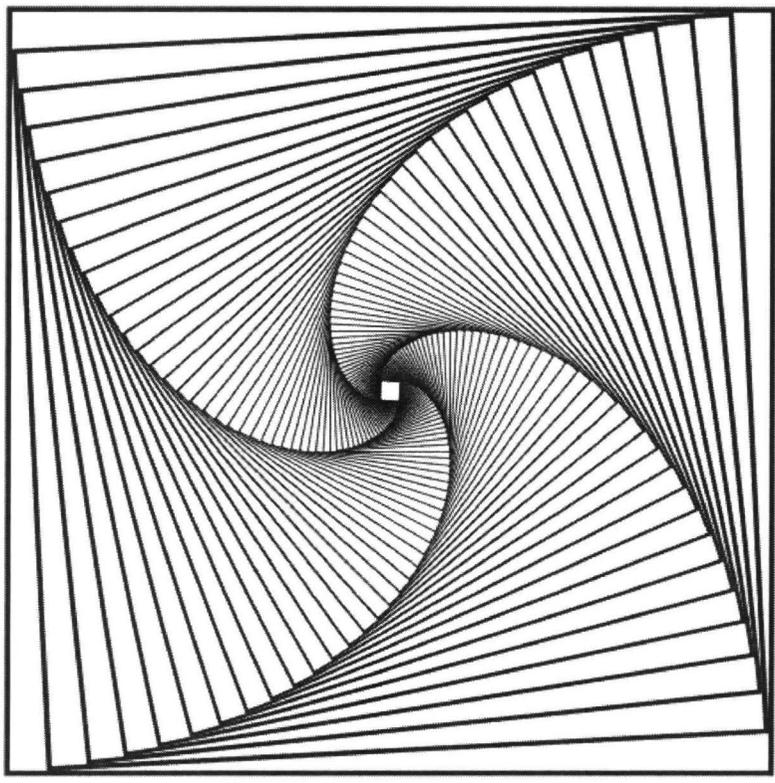


To conclude this introduction to PostScript, we introduce `scale` and `rotate`; these change the *graphics state* and do what they promise. Combined with the simple `repeat` loop, this shows how visual effects can be easily achieved.

```
0.03 setlinewidth
60 {
  4 0 0 drawsquare
  0.94 0.94 scale
  3 rotate
} repeat
showpage
```

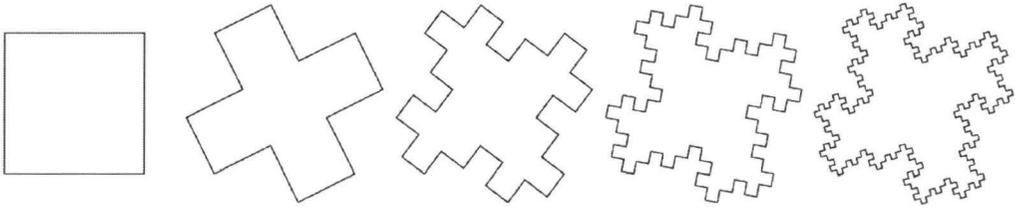
% repeat this 60 times:
% draw the same square
% but change the ground
% under its feet.

Giving:

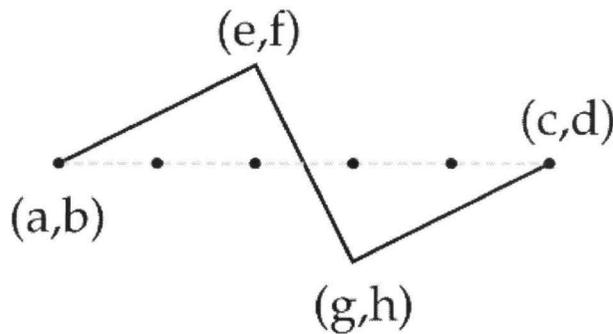


Recently I discovered the book *Mathematical Illustrations: A Manual of Geometry and PostScript* by Bill Casselman [1], also available online.¹ It is an excellent introduction to PostScript, builds up sophisticated applications in geometry, and creates powerful PostScript idioms and abstractions. An example is the `mkpath` procedure that can be used to draw parametric curves. It creates the curve as a PostScript path of concatenated Bezier curves, using the derivatives of the parametric functions for the x and y coordinates to create appropriate Bezier control points.

Let us now turn to the Minkowski fractal.² It is derived by a series of transformations of the unit square.



The full fractal is obtained by continuing this transformation *ad infinitum*. Here I am only concerned with the different stages of transformation. A closer look at the transformation undergone by each edge:



The pairs (a, b) and so on denote Cartesian coordinates in the plane. By inspecting the transformation we derive the following traits of the n^{th} Minkowski transformation.

- The number of line segments equals $4 \cdot 3^n$.
- Each transformation can be constructed from 5 copies of the previous transform, when viewed as areas.
- The number of unit tiles in the interior hence equals 5^n .
- The length of a line segment is $(1/\sqrt{5})^n$, if the start tile has side 1.

In the first of our two approaches to coding this, let us compute the necessary coordinate transformations. We can compute the transformations at a certain depth by iteratively transforming a source edge into three new edges. To this end, we need to specify the new

coordinates (e, f) and (g, h) in terms of the known coordinates (a, b) and (c, d) . This can be done by utilizing the vector $\begin{pmatrix} c-a \\ d-b \end{pmatrix}$ for the displacement along the original edge, and by utilizing the vector $\begin{pmatrix} -d+b \\ c-a \end{pmatrix}$ for the displacement orthogonal to it. These dependencies are as follows, leaving the workings as an exercise for the reader.

$$\begin{bmatrix} e \\ f \\ g \\ h \end{bmatrix} = \frac{1}{10} \begin{bmatrix} 6 & 2 & 4 & -2 \\ -2 & 6 & 2 & 4 \\ 4 & -2 & 6 & 2 \\ 2 & 4 & -2 & 6 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

The above indicates for example that e can be computed as the inner product of the first row of the matrix with the vector $[a, b, c, d]^T$, that is, $0.6a + 0.2b + 0.4c - 0.2d$. Encode an edge from (x, y) to (u, v) in a PostScript array `[x, y, u, v]`. It is handy to have a PostScript procedure that computes inner products, introduced below as `ip4ary`.

```
% compute inner product between two arrays of length 4.
% expected stack: [a b c d] [
u v x y]
/ip4ary {
  aload pop 5 4 roll aload pop
  7 4 roll
  5 3 roll
  3 2 roll
  mul 7 1 roll
  mul 6 1 roll
  mul 5 1 roll
  mul add add add
} def
% state of stack:
% a b c d u v x y
% a u v x y b c d
% a u v b c d x y
% a u v b c x y d
% y*d a u v b c x
% c*x y*d a u v b
% v*b c*x y*d a u
% v*b+c*x+y*d+a*u
```

There may be better ways to do this, but this is certainly a stacky solution. We proceed with PostScript code that accepts an initial edge and a certain depth, then recursively calls itself to compute the Minkowski transformation to that depth. Although PostScript has a facility for name spaces in the form of dictionaries, rendering performance degrades quickly when used in conjunction with recursion. Hence we seek a solution that depends solely on stack operations. In this case, the desired state of the stack was written first as a comment, and from that the necessary code was derived - a common way to derive

PostScript programs³. From a starting edge, the code works out the three new edges, pushes them onto the stack in the right order, and calls itself again on the three new edges until the recursion depth is reached.

```

/cf_e [0.6 0.2 0.4 -0.2] def      % inner product coefficients for e,
/cf_f [-0.2 0.6 0.2 0.4] def    %   f,
/cf_g [0.4 -0.2 0.6 0.2] def    %   g,
/cf_h [0.2 0.4 -0.2 0.6] def    % and h.

/minkowski_geom {               % [a b c d] depth on stack, abbreviat
e [] dep
  dup                            % [] dep dep
  0 gt {                          % [] dep
    -1 add dup dup 4 3 roll      % dep-1 {3} [] - {3} indicating thr
ee instances
    dup dup dup dup             % dep-1 {3} [] [] [] [] []
    cf_e ip4ary exch cf_f ip4ary % dep-1 {3} [] [] [] e f
    2 copy 7 4 roll             % dep-1 {3} e f e f [] [] []
    cf_g ip4ary exch cf_h ip4ary % dep-1 {3} e f e f [] g h
    2 copy 5 4 roll             % dep-1 {3} e f e f g h g h []
    aload pop                   % dep-1 {3} e f e f g h g h a b c d
    4 2 roll                    % dep-1 {3} e f e f g h g h c d a b
    6 2 roll 4 array astore     % dep-1 {3} e f e f g h a b [g h c d]
    12 1 roll                   % [g h c d] dep-1 {3} e f e f g h a b
    6 2 roll 4 array astore     % [g h c d] dep-1 {3} e f a b [e f g
h]
    7 1 roll                    % [g h c d] dep-1 [e f g h] dep-1 dep
-1 e f a b
    4 2 roll 4 array astore exch % [g h c d] dep-1 [e f g h] dep-1 [a
b e f] dep-1
    minkowski_geom              % [g h c d] dep-1 [e f g h] dep-1
    minkowski_geom              % [g h c d] dep-1
    minkowski_geom              %
  } {
    pop
    dup 2 get exch 3 get lineto  % lineto [c d]; previous code got us
to [a d];
  } ifelse
} def

```

Noteworthy is that the PostScript primitive `lineto` adds to the current path. The path is only drawn once `stroke` has been issued. Thus the path is added to throughout the recursive traversal of the stack and completely assembled before actually being drawn,

illustrating that PostScript neatly separates the graphics state from computations. The following fragment uses this code to draw the five different Minkowski squares seen earlier. Fractal computing in PostScript tends to be costly. On my desktop computer the display software starts to struggle noticeably when the fractal depth reaches 12. At that stage, the path is composed of a respectable 2,125,764 segments. Equally, it is no longer possible to visualise such a fractal such that both its (identical) macro and micro features are visible.

```
0.02 setlinewidth
/maxdepth 4 def
newpath
0 1 maxdepth {
  /depth exch def
  gsave
    depth -2 add 3 mul 0
    translate
    newpath 0 0 moveto
    [0 0 0 2] depth minkowski_geom
    [0 2 2 2] depth minkowski_geom
    [2 2 2 0] depth minkowski_geom
    [2 0 0 0] depth minkowski_geom
    stroke
  grestore
} for
showpage
```

Now for a second approach, that utilises the PostScript graphics operators to a greater extent and computes the Minkowski fractal inside-out, as it were. If we forget about in-place transformations of the fractal and look at the unit square as a series of four connected steps, it can be observed that the Minkowski transformation turns a right step R into two right steps followed by a left step, that is, RRL . From the second transformation we see that a left step L is transformed into LRL .

The relationship between fractals and rewrite systems is well established. Many fractals can be derived using L -systems, named after Aristid Lindenmayer.⁴ I know only little about it—the Wikipedia entry is a good place to start and has a cornucopia of pretty pictures. Here we simply code these rules directly into verbose PostScript code as follows, using the PostScript primitives `translate` and `rotate` to implement turning left and right.

```

/right 1 def
/left -1 def

/minkowski_rewrite { % expects [r g b] x y depth on stack
  /thedept h exch def
  gsave translate newpath
  /mycol exch def
  0 0 moveto
  right thedepth right thedepth right thedepth right thedepth
  { count 0 eq { exit } if
    /d exch def
    /where exch def
    d 0 gt {
      where right eq {
        right d -1 add right d -1 add left d -1 add
      } {
        left d -1 add right d -1 add left d -1 add
      } ifelse
    } {
      0 1 lineto
      0 1 translate
      90 where mul rotate
    } ifelse
  } loop
  mycol aload pop setrgbcolor
  closepath stroke grestore
} def

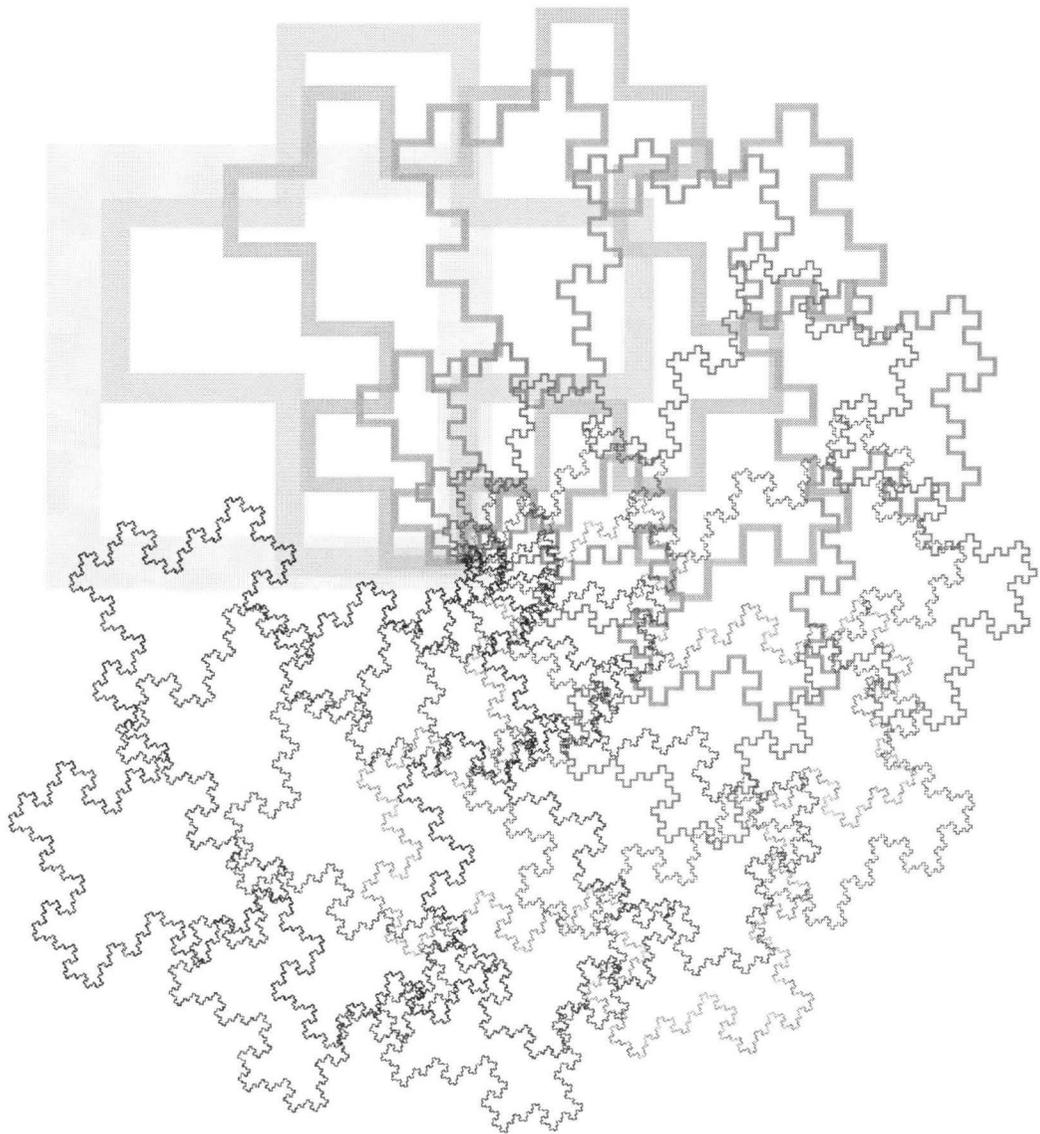
```

The loop code executes indefinitely as long as the stack is not empty (more robust code would compare with the stack size before entering the routine). In the figure below we depict successive Minkowski iterations, each starting at the exact same point, and scaled so that they occupy roughly the same amount of space, using the scaling factor from our previous exploration. The successive Minkowski transformations move around the origin in a clockwise manner, and after about the sixth iteration no new level of detail can be observed. With a little bit more code, we enter a dream-like state of Minkowski transformations wandering around.

```
0 0 translate
5.0 5.0 scale
/maxdepth 10 def
```

```
0.14 setlinewidth
```

```
0 1 maxdepth {
  dup /mydepth exch def
  /num exch def
  /rgb num -0.08 mul 0.9 add def
  [ rgb dup dup ] 0 0 mydepth minkowski_rewrite
  currentlinewidth 1.2 mul setlinewidth
  5 -0.5 exp dup scale
} for
showpage
```



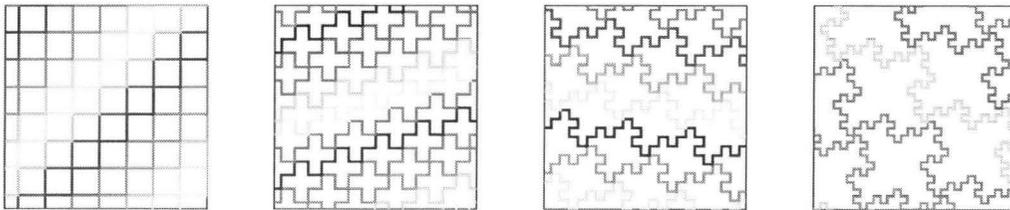
(continued on the next page)

Rambling with Minkowski

Stijn van Dongen

(continued from the previous page)

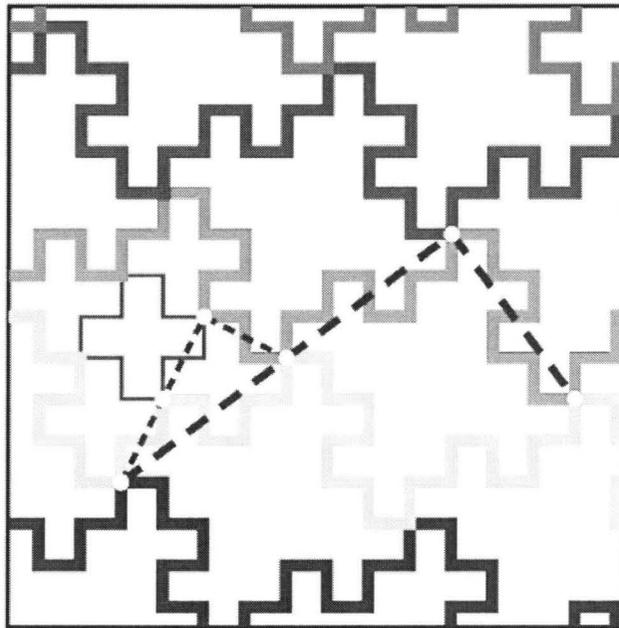
I first noticed this wandering habit in a different context, namely when I looked into tiling the plane with Minkowski transformations. A simple way to tile the plane, once the code for a single transformation has been written, is to iterate and reuse that code at different (x, y) offsets. This requires one to iterate across two dimensions if we aim to tile the plane. We can do better with fairly little effort, by using a staircase, say with starting symbols *RLRLRLRLRLRLRL* (extended to a convenient length). Such a staircase only needs repeating across a single dimension in order to tile the plane (and shows the flexibility of the rewrite approach). The first few staircase tilings of the plane are shown below, followed by a table listing the offsets between successive staircases in these tilings, for increasing depth of the Minkowski transformation.



Iteration	x -displacement	y -displacement
0	0	1
1	1	2
2	4	3
3	11	2
4	24	-7
5	41	-38

When first considering these offsets they may look fairly haphazard. However, they are tightly constrained, as illustrated a little below. They can also be found (if we allow some fiddling of signs) in the On-Line Encyclopedia of Integer Sequences⁵ (OEIS), where one of the listed properties is that each pair (x, y) in row n is the unique solution to

$x^2 + y^2 = 5^n$ with x and y relatively prime. For example, $2^2 + 11^2 = 5^3$. The value 5^n corresponds to the number of unit tiles in a Minkowski transformation, and the displacement (x, y) can be viewed as the vector encoding of its side. It can be seen in the diagram below that the square that has (x, y) as its side has the same area as the Minkowski transformation, and hence we can confirm independently that $x^2 + y^2 = 5^n$. The fact that x and y are relatively prime translates to the fact that the vector (x, y) can not be split in identical smaller vectors with integer coordinates. This shows that the Minkowski curve can not be further decomposed into equal parts, confirming its fractal unruliness⁶. We now establish a simple linear relationship between successive displacements.



Observe that for the ground case of the Minkowski tiling (the unit square), a working set of values for the staircase displacement is $(0, 1)$. Then, for the first iteration $(1, 2)$ can be taken. In the figure above, it is shown that the displacement for the second iteration $(4, 3)$ can be constructed as the vector sum of twice the displacement of the first iteration plus a displacement that is orthogonal to it. In the same figure, this construction is repeated to reveal the displacement for the third iteration. In words, the new x displacement is twice the previous x displacement plus the previous y displacement. The new y displacement is twice the previous y displacement minus the previous x displacement. Finally, this can be understood by seeing that each new Minkowski iteration (viewed as an area) is

constructed as five copies of the previous one, four of them centered around the fifth, and by observing that the required displacement travels a quarter of the perimeter of the new Minkowski transformation, starting at a juncture where four current transformations meet.

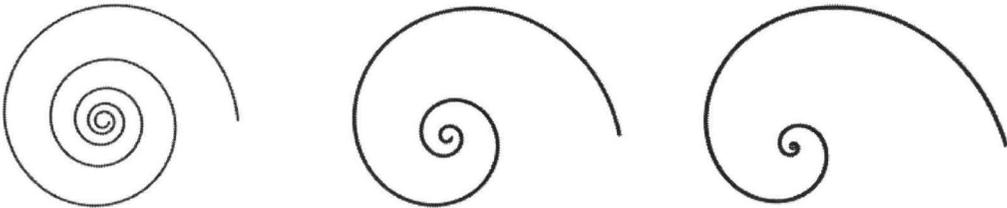
The displacements listed above are thus related by a linear transformation:

$$\begin{aligned} \begin{bmatrix} dx_{n+1} \\ dy_{n+1} \end{bmatrix} &= \begin{bmatrix} 2 & 1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} dx_n \\ dy_n \end{bmatrix} \\ &= \sqrt{5} \begin{bmatrix} \frac{2}{5}\sqrt{5} & \frac{1}{5}\sqrt{5} \\ -\frac{1}{5}\sqrt{5} & \frac{2}{5}\sqrt{5} \end{bmatrix} \begin{bmatrix} dx_n \\ dy_n \end{bmatrix} \\ &= \sqrt{5} \begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} dx_n \\ dy_n \end{bmatrix} \end{aligned}$$

The linear transformation was rewritten as a combination of a linear scaling by a factor of $\sqrt{5}$ and a (clockwise) rotation matrix corresponding to an angle of rotation $\phi = \arctan(\frac{1}{2})$. This is intriguing, as scalings and rotation matrices are intimately related to spirals, more specifically, logarithmic spirals. After a short introduction to logarithmic spirals, we shall revisit their presence in the Minkowski landscape. In parametric form, a right-turning logarithmic spiral is described by

$$\begin{aligned} x(t) &= ae^{bt} \cos(t) \\ y(t) &= -ae^{bt} \sin(t) \end{aligned}$$

The parameter b controls the growth rate. Below three spirals are shown⁷ with increasing values for b of 0.1, 0.2, and 0.3, leading to geometric growth factors every full turn of respectively 1.87, 3.51, and 6.59.

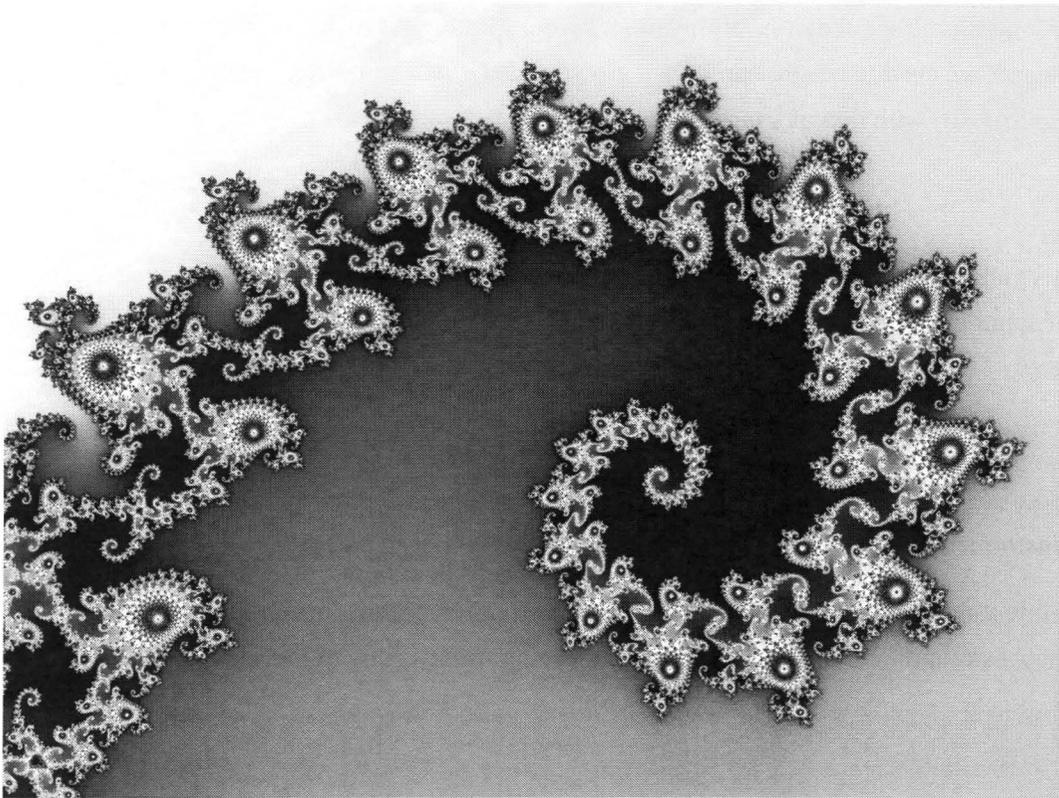


The logarithmic spiral was first described by Descartes⁸ and later extensively investigated by Jacob Bernoulli, who called it *Spira mirabilis*, the marvelous spiral. It has a number of remarkable properties. Scaling by any constant $e^{2k\pi b}$ (where k is an integer number)

results in the same spiral. Scaling by any other constant results in the same spiral, but rotated. Zooming in on the centre reveals the spiral winding its way endlessly around the origin, never getting there, never changing its shape. Nevertheless, the length of the path towards the centre is finite—a point that lies at straight-line distance r from the origin, lies at distance $r / \cos(\phi)$ along the spiraling path. It was the self-similarity at all scales that fascinated Bernoulli and led him to instruct that the spiral be engraved on his headstone along with the inscription *Eadem mutata resurgo* (although changed, I shall arise the same). The logarithmic spiral does not budge when subjected to a plethora of further transformations. It is in fact its own caustic, evolute, inverse, involute, orthoptic, pedal and radial.⁹

On this evidence, I should like to deform the accepted history of fractals slightly and propose that the logarithmic spiral is perhaps the earliest known mathematical object that can be ascribed a (part) fractal nature and was recognised as such, instilling a sense of transcendence¹⁰. The history of fractals¹¹ is traced to stirrings with Leibniz, gaining traction with Weierstrass and Cantor, and assumes a more familiar modern form with the Koch snowflake, the Sierpinski triangle and the Julia and Fatou sets. Minkowski lived in the era that produced these last fractals, but it is surprisingly hard to find exactly when and where he proposed the eponymous fractal. It is not mentioned in his collected works [2]. According to Mandelbrot the origin of the curve is uncertain and dates back at least to Minkowski.

The concept of the continuum allows self-similarity at all scales. The logarithmic spiral embodies this using a singular singularity. In the last century, fractals erupted into the public consciousness, perhaps none more so than the Mandelbrot set, dense with singularities everywhere. Pleasingly, logarithmic spirals are found in this set¹².



We return to the Minkowski displacements found earlier, and consider the associated scaling plus rotation

$$\sqrt{5} \begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix}$$

where the angle ϕ has the value $\arctan(\frac{1}{2})$. It is known that the right combination of scaling and rotation leaves a logarithmic spiral invariant, that is, a spiral with growth parameter b is left invariant by any matrix of the form:

$$e^{b\alpha} \begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

This follows from working out the matrix product for the x -component as $ae^{bt} e^{b\alpha} (\cos(t) \cos(\alpha) - \sin(t) \sin(\alpha))$, which simplifies to $ae^{b(t+\alpha)} \cos(t + \alpha)$ using the cosine addition trigonometric identity. The y -component is similarly computed. It follows that the matrix maps any point on the spiral to another point on the spiral that is obtained by rotating over the angle α whilst moving along the spiral. Now we seek to find

the logarithmic spiral that is left invariant by our Minkowski rotary expansion matrix. To this end, we need to equate our known growth rate (which equals $\sqrt{5}$ across an angle of $\arctan(\frac{1}{2})$) with that of a logarithmic spiral.

$$\sqrt{5} = e^{b \arctan(\frac{1}{2})}$$

This yields the following suitably brutalist constant, where $b \approx 1.7356$. The growth rate per spiral turn is thus $e^{2\pi b} \approx 54462$.

$$b = \frac{\log(5)}{2\arctan(\frac{1}{2})}$$

These considerations raise the question, can we arrange different Minkowski transformations into a spiraling arrangement? This is indeed possible.

Below a single spiral arm is shown¹³, composed of transformations of increasing size. The Australia-like shape on the right is a close-up of the centre of the spiral.

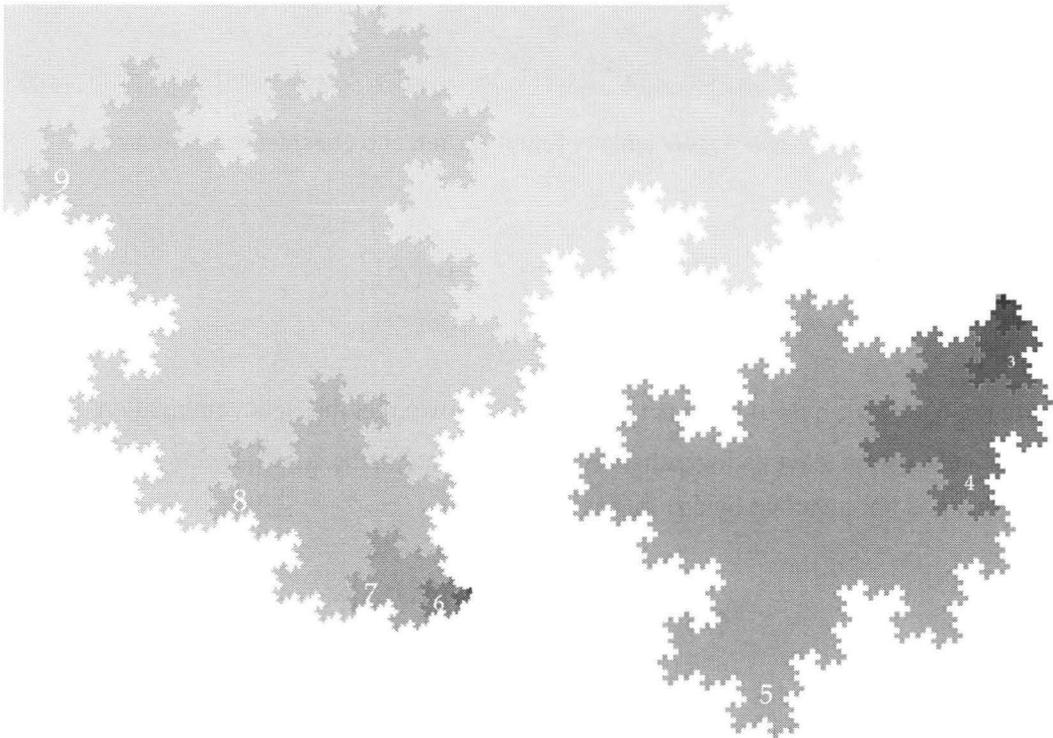


Figure 1 has four such spiral arms interlocking and filling space. To add some interest a certain amount of recursion was introduced to re-insert smaller transformations into larger ones, still leaving each individually coloured area to be a Minkowski transformation.

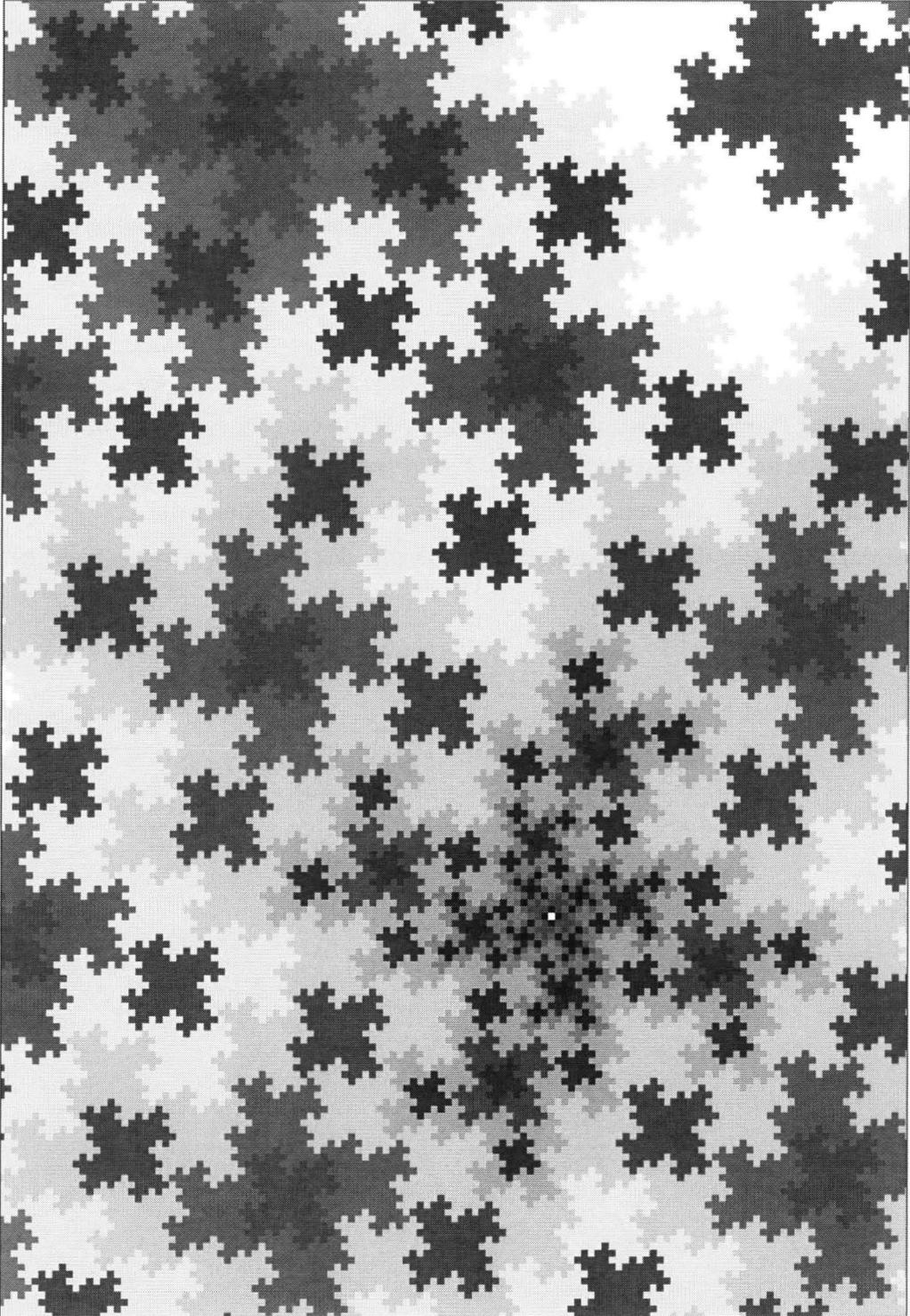


Figure 1: A spiraling flock of Minkowski transformations

Each turn fits slightly less than 14 Minkowski transformations, calculated as $2\pi/\arctan(\frac{1}{2}) \approx 13.55$. However, the figure only shows eight different levels of transformations. The last one is partially shown in the top right corner and belongs to the arm starting at the tile south to the spiral centre.

To arrive at the logarithmic spiral parameter a for the Minkowski spiral, the tile in the centre is fixed equidistantly around the origin at coordinates $(\pm\frac{1}{2}, \pm\frac{1}{2})$, yielding with a bit of work:

$$a = \frac{1}{\frac{b\pi}{e^4 \cdot \sqrt{2}}}$$

The following Python code uses these values for a and b to compute spiral coordinates for the Minkowski logarithmic spiral at angle increments of $\arctan(\frac{1}{2})$, starting at coordinate $(\frac{1}{2}, -\frac{1}{2})$. Additionally it computes the offsets of the Minkowski transformations using the linear transformation to verify that the two approaches yield the same values.

continued

```
import math

qpi = math.atan(1)
b = 0.5 * math.log(5) / math.atan(0.5)
a = 1 / (math.exp(b*qpi) * math.sqrt(2))

phi = math.atan(0.5) # The angle distance covered by our rotary expansion matrix.
startangle = qpi # Corresponds with (0.5, -0.5) (because spiral turns right).

(pxt, pyt) = (0,0) # Previous x(t), previous y(t)
(xm, ym) = (0,0) # x matrix-based, y matrix-based

for i in range (0,7):

    t = startangle + i * phi
    xt = a * math.exp(b * t) * math.cos(t) #
    yt = - a * math.exp(b * t) * math.sin(t) # A right-turning spiral (hence minus)

    if i == 0:
        (xm, ym) = (0.5, -0.5)
    else:
        (xm, ym) = (2*xm + ym, - xm + 2*ym)

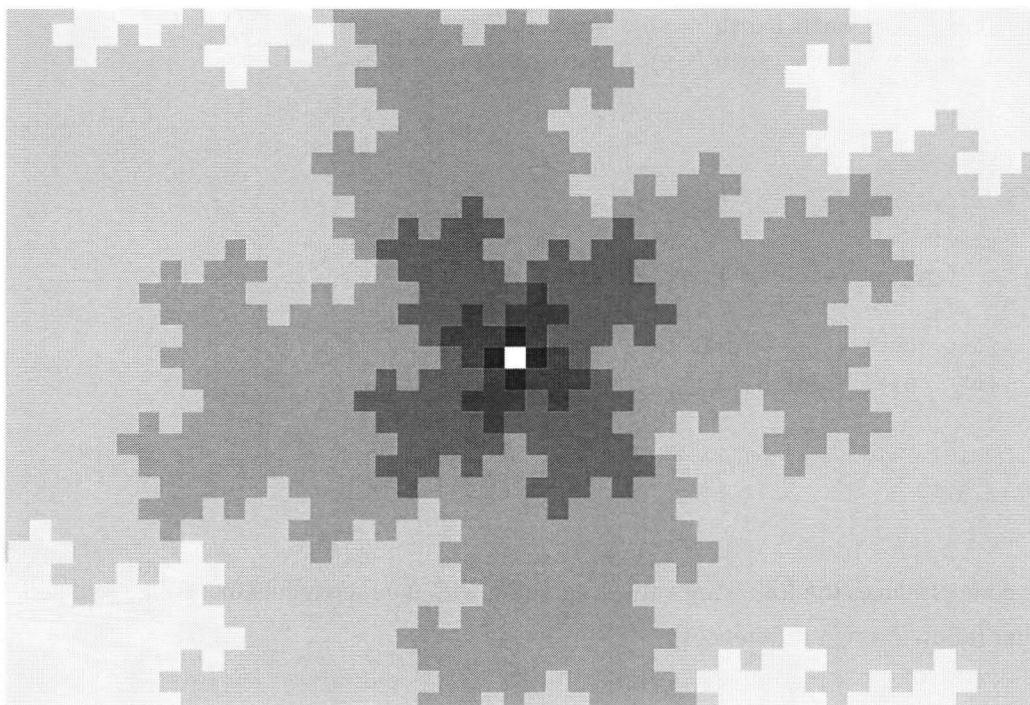
    (dxt, dyt) = (xt - pxt, yt - pyt)
    (pxt, pyt) = (xt, yt)

    print "%1d %7.2f %7.2f %7.2f %7.2f %7.2f %7.2f" % (i, xt, yt, xm, ym, dxt, dyt)
```

The code produces the following values, agreeing with the increments that were tabulated earlier (allowing sign differences).

Iteration	x(t)	y(t)	x- Minkowski	y- Minkowski	x- increment	y- increment
0	0.50	-0.50	0.50	-0.50	—	—
1	0.50	-1.50	0.50	-1.50	0.00	-1.00
2	-0.50	-3.50	-0.50	-3.50	-1.00	-2.00
3	-4.50	-6.50	-4.50	-6.50	-4.00	-3.00
4	-15.50	-8.50	-15.50	-8.50	-11.00	-2.00
5	-39.50	-1.50	-39.50	-1.50	-24.00	7.00
6	-80.50	36.50	-80.50	36.50	-41.00	38.00

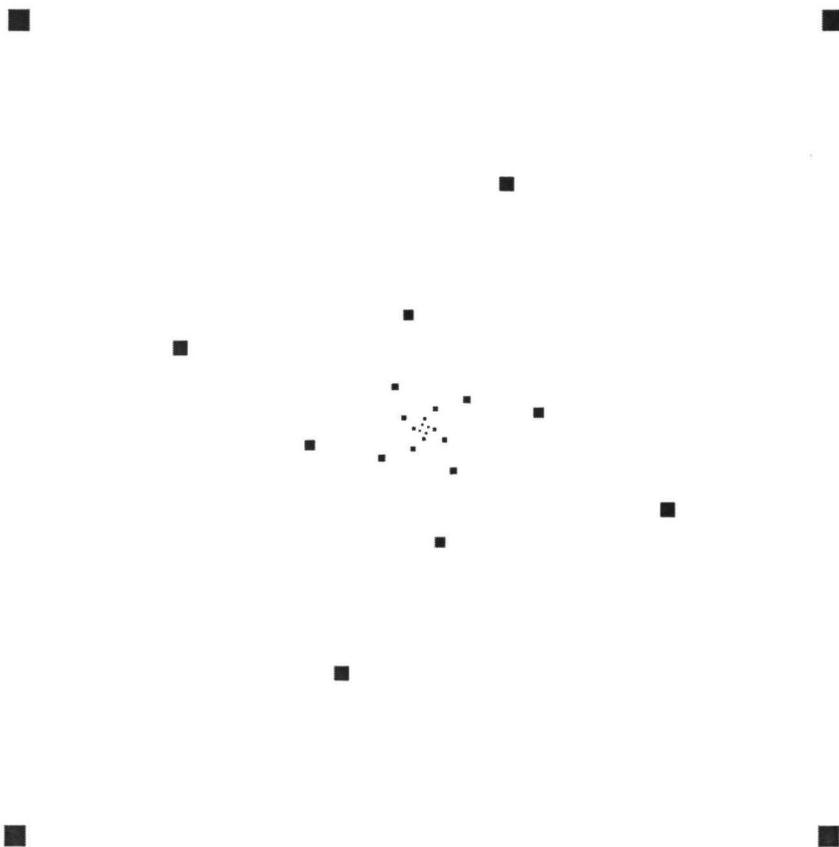
The centre tile in the spiral in Figure 1 is not accounted for by any of the four arms. Zoomed in and de-cluttered it looks like this¹⁴:



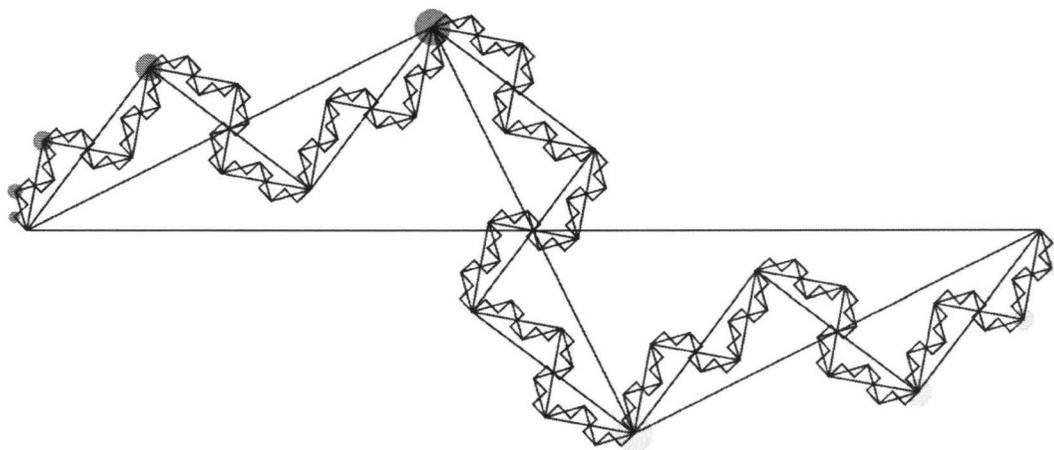
The central tile is the genesis tile of the Minkowski transformation, indivisible, the Planck limit of the Minkowski landscape. The logarithmic spiral associated with it knows no such limitations however, and will forever dreamily wind its way towards the origin. Using the inverse of the rotary expansion matrix, we can even plot the coordinates where the Minkowski transformations of negative index should be found. The area of such a transformation of negative index $-n$ should have 5^{-n} tiles (doable) but it should

continued

additionally have $4 \cdot 3^{-n}$ line segments (tricky). Nonetheless, fractals are the domain of fractional dimensions where concepts such as area and length transcend their everyday meaning. Perhaps there is a way.



Jan, het ga je goed, ik hoop dat onze wegen zich weer kruisen!



Epilogue Spirals are one possible key to understanding the full Minkowski fractal. The line segments are really a distraction. The two new points introduced in each edge transformation persist throughout all subsequent transformations. The infinite union of all these points forms the Minkowski fractal. Each edge transformation induces two spirals that approximate the end points defining the edge, as indicated in the figure. These spirals have the same growth rate as derived above, except that we observe it as a shrinkage rate. This principle applies to all the end points of all the edge transformations at every level. Every point in the fractal is the singularity of a spiraling arrangement of other points in the fractal, each of which... it's spirals all the way.

Notes

¹. <http://www.math.ubc.ca/~cass/graphics/manual/> ↩

². It is often called the Minkowski sausage. This seems an unfortunate case of entrenched confusion with the method of approximating a fractal curve with overlapping circles (i.e. with a Minkowski cover or Minkowski sausage) to arrive at the Minkowski dimension. ↩

³. Sadly, the specification is not the program. ↩

⁴. <https://en.wikipedia.org/wiki/L-system> ↩

⁵. <https://oeis.org/A098122> ↩

⁶. In this case, I see no easy way to confirm the relative primeness of x and y by just considering the Minkowski transformation. ↩

⁷. The `mkpath` procedure from [1] was used to create these. ↩

⁸. https://en.wikipedia.org/wiki/Logarithmic_spiral ↩

⁹. <http://www.2dcurves.com/spiral/spirallo.html> ↩

¹⁰. An interesting contrast with the horrors and non-disclosure agreements visited upon the Pythagoreans when they discovered Pandora's box of irrational numbers. ↩

¹¹. <https://en.wikipedia.org/wiki/Fractal> ↩

¹². https://commons.wikimedia.org/wiki/File:Mandel_zoom_04_seehorse_tail.jpg ↩

¹³. There is a small technicality, in that the spiraling Minkowski coordinates are partial sums of the displacements, not the displacements themselves. However, these partial sums are still related by the same linear transformation. ↩

¹⁴. This figure also shows how the Minkowski spiral is easily understood by seeing how each transformation sits inside its successor transformation. ↩

References

[1] B. Casselman, *Mathematical Illustrations: A Manual of Geometry and PostScript*, Cambridge University Press, 2005.

[2] H. Minkowski, *Gesammelte Abhandlungen von Hermann Minkowski*, Chelsea Pub. Co., New York, 1967.

Route 65

Heleen Verleur

Route 65

(for Jan, on the occasion of his retirement of the CWI)

Heleen Verleur
Januari 2017
www.heleenverleur.org

The musical score for "Route 65" is presented in two systems. The first system features a Clarinet in Bb and a Piano. The Clarinet part begins with a tempo marking of $\text{♩} = 176$ and a dynamic of *mp*. A performance instruction above the staff reads: "Quavers played uneven, with a 'jazzy' feeling". The Piano part also starts with *mp* and includes a performance instruction: "All quavers played uneven, with a 'jazzy' feeling". The second system features a Clarinet (Cl.) and Piano (Pno.). The Clarinet part begins with a dynamic of *mf*. The Piano part includes a triplet of eighth notes in the first measure, with fingerings 3, 1, 2, 4, 5, 1 indicated below. The score is written in 5/4 time and consists of 17 measures.

(The full score can be found at the end of the book.)

The syntax of truth: a grammar-based approximation of satisfiability

Jurgen J. Vinju

Dear Jan, I have written this essay for you, because the topic fits very well with three of your expertises: language [11], functional programming [13] and logic [12]. It entails a purposefully ambiguous context-free grammar for *true* Boolean formulas, which can be used to produce derivation trees in expected linear time, worst-case cubic time. The derivation trees are then processed again in worst case exponential time to decide satisfiability, but the expected execution time is much less. Due to the acyclic graph structure of the representation of the ambiguous derivations, which exploits sharing, often polynomial behavior can be expected. Another interesting aspect of this encoding is that it works without the traditional canonical conjunctive normal form; the algorithm maintains the "natural" decomposition of the input formula throughout the entire process. This property can perhaps lead to some unexpected short-cuts to deciding satisfiability. The code in this paper is complete and fully executable on Rascal version 0.8.4. *Thanks for being a great colleague at CWI.*

A grammar for all Boolean formulas

For the sake of reference and before we dive into further detail, Figure 1 defines an extended context-free grammar (in Rascal notation [8]) for open Boolean formulas to unambiguously define their syntax. I leave the definition of what an extended context-free grammar in Rascal notation is to your imagination. In the next section we will specialize this grammar, such that only closed formulas which evaluate to *true* can be accepted, *ceteris paribus*.

Lemma 1 (Unambiguity) *We claim without further proof that, given the explicitly defined priority relation and associativity relations [2], and the reservation of the *True* and *False* literals from the language of atoms, the extended context-free grammar in Figure*

1 is unambiguous for all Boolean formulas containing atomic words, *True* and *False* terminals, the negation, the two operators *or* and *and*, and arbitrary brackets around formulas.

In the following, we will discuss input sentences for other grammars which describe appropriate sub-languages of $L(\mathbf{Formula})$. The notation $L(\mathit{nonterminal})$ is used to identify the set of sentences generated from a non-terminal in an extended context-free grammar. Unless otherwise stated, we assume each input sentence we discuss in this essay is a member of $L(\mathbf{Formula})$.

```

1  module AllBooleans
2
3  syntax Formula
4    = "true"
5    | "false"
6    | Atom
7    | "(" Formula ")"
8    | "not" Formula
9    > left Formula "and" Formula
10   > left Formula "or" Formula
11   ;
12
13 layout Whitespace = [\t\n\ ]+;
14 lexical Atom = [a-z]+ \ "true" \ "false";

```

Figure 1: An unambiguous extended context-free grammar for Boolean formulas in Rascal.

A grammar for *true* Boolean formulas

```

1  module ClosedBooleans
2
3  syntax True
4  = "true"
5  | "(" True ")"
6  | "not" False
7  > left True "and" True
8  > left ( True "or" True
9          | False "or" True
10         | True "or" False
11         );
12
13 syntax False
14 = "false"
15 | "(" False ")"
16 | "not" True
17 > left ( False "and" False
18         | True "and" False
19         | False "and" True
20         )
21 > False "or" False;
22
23 layout Whitespace = [\t\n\ ]*;

```

Figure 2: A context-free grammar for true and false closed Boolean formulas

Figure 2 describes a derived grammar for the same textual language of Boolean formulas, but without any atomic words, and without any false formulas, nota bene! By introducing separate non-terminals for `True` and `False` formulas, and by encoding standard axioms for Boolean logic as context-free production rules, we constructed a grammar for the `True` non-terminal which only accepts *true* Boolean formulas and does not accept any *false* formulas. Vice versa, the `False` non-terminal accepts only *false* formulas and no *true* formulas.

Lemma 2 (Language inclusion) *The rules for `True` and `False` in Figure 2 are constructed such that $L(\text{True}) \subset L(\text{Formula})$ and $L(\text{False}) \subset L(\text{Formula})$.*

I leave the proof of this lemma to you Jan, but *QuickCheck* [5] for Rascal can also be a effective way of finding out whether I was wrong. Nonterminal types in Rascal range over the derivation trees for the given non-terminal. The square brackets notation generates and calls a parser for a specific non-terminal on an input string, and the angular brackets in a string "unparse" a derivation tree back to the input string:

```

1  import AllBooleans;
2  import ClosedBooleans;
3  test Formula TrueSubFormula(True x) = [Formula] "<x>";
4  test Formula FalseSubFormula(False x) = [Formula] "<x>";

```

To evaluate a closed formula, to tests its trivial satisfiability so to say, we can run a parser for the `True` non-terminal on a given formula input sentence. For example:

```

1 rascal>[True] "true and true"
2 True: 'true and true'
3 rascal>[True] "true and false"
4 ParseError(prompt:/// (0, 23, <1, 0>, <1, 23>))
5       at $root$ (|prompt:/// (0, 22, <1, 0>, <1, 22>))

```

Lemma 3 (Soundness) *The parser for `True` accepts an input sentence $s \in L(\text{Formula})$ if and only if the Boolean valuation of s is `_true` and it produces a parse error if and only if the valuation of s is `false`.*

The grammar for `True` is not ambiguous, but it is rather "non-deterministic", as they say: it is not an element of the deterministic LL or LR classes of grammars. A parser generated from this grammar would need to be backed by an algorithm which branches or backtracks when faced with non-deterministic choice in order to search for the single acceptable derivation. There is no need for alarm though, since we will first make this grammar even more non-deterministic and also even ambiguous before doing anything useful with it.

Given an appropriately smart parser generator and parsing algorithm [6], this grammar can be used to test the truth of any closed Boolean formula in linear time (in the length of the formula). The generator may discover shared prefixes of the rules and the parser will avoid double work to arrive at this linear behavior, but this is neither the topic nor the contribution of the current essay. In any case, a hard upper-bound for any appropriately general context-free parser architecture [3,10] is that sentences for this grammar can be parsed in $O(n^4)$ where n is the length of the input formula. If the grammar rules are brought in Chomsky normal form internally by the parsing algorithm ("binarised") the worst case execution time can be bound by $O(n^3)$ [7,9]. For this particular grammar in Figure 2, a (much) tighter bound may exist given a specific parsing algorithm, but let's leave this open question to the interested reader.

An ambiguous grammar for open *true* formulas

To make matters more interesting we now extend the previous grammar to include words for atoms again, rendering satisfiability of the generated formulas a lot more interesting (see Figure 4). Note that the `extend` keyword can be read as literally including the

contents of the other grammar module here.

Now, as opposed to the previous unambiguous grammar for closed formulas, this new grammar with open formulas is very ambiguous. We define ambiguity of a context-free grammar as the existence of multiple derivations for the same input sentence. Consider the input string "p or q" it can be interpreted by either of the three rules for or and to derive p on the left-hand side or q on the right-hand side via either the True atom or the False rule for an Atom. Figure 3 graphically illustrates the ambiguity.

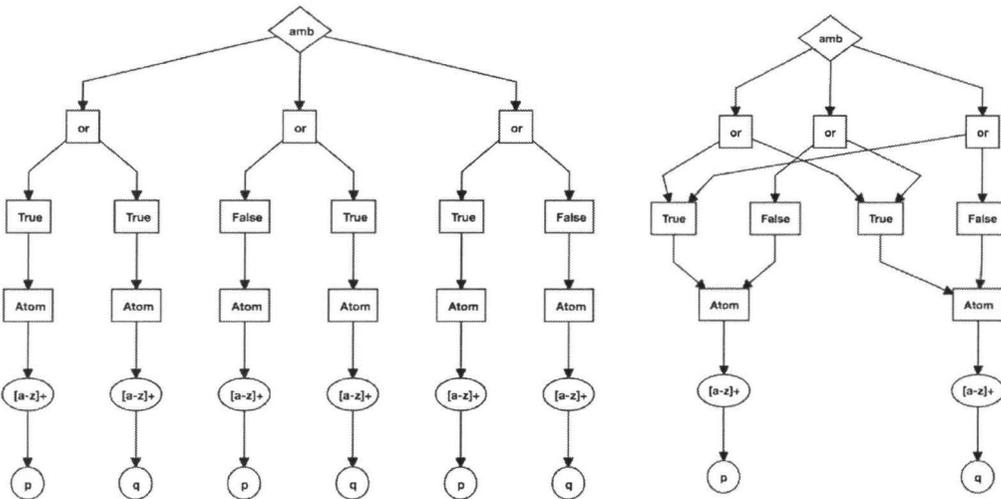


Figure 3: The ambiguous derivations of True for the input p or q. On the left the full derivation trees are shown and on the right a graph which illustrates the possible sharing of sub-derivations between alternates.

```

1 module OpenBooleans
2 extend ClosedBooleans
3
4 syntax True = Atom;
5 syntax False = Atom;
6
7 lexical Atom = [a-z]+ \ "true" \ "false";

```

Figure 4: Extending the closed Boolean formulas (Figure 2) with atoms again.

There exists an exponential number of derivations for arbitrary open Boolean formulas with this ambiguous grammar (in the length of the formula). Since satisfiability is NP-complete anyway, so far I'm not too worried to make matters worse.

Lemma 4 (Completeness) *For the grammar in Figure 4 it holds that:*
 $L(\text{True}) \cup L(\text{False}) = L(\text{Formula})$, and still Lemma 2 holds as well.

Luckily, the algorithms for general context-free parsing, i.e. Earley's algorithm (with fixed parse tree production [4]), or the fixed and binarized Tomita algorithm [9] or top-down general LL parsing (GLL [1]) can produce derivation trees or parse errors for any input for this grammar in polynomial time and space. The intuition on how this works is that the parser uses a smart dynamic programming trick (a call-stack structured as a directed acyclic graph) to avoid doing the same work twice, and derivation trees are stored as directed acyclic graphs as well (see Figure 3).

Both graphs have guaranteed enough sharing to keep within the polynomial bound, while they encode in fact an exponential amount of derivation trees. It is important to know that ambiguous sub-sentences in the derivation trees are represented using "ambiguity nodes" holding sets of alternative derivations for each sub-sentence.

Lemma 5 (Unsoundness) *Alas, Lemma 3 does not hold for the grammar in Figure 4. E.g. the unsatisfiable sentence "p and not p" is accepted by True without further ado by simply deriving True for the first instance of p and False for the second instance of p. The reason is the parser has no notion of a consistent valuation of the same atoms throughout a single formula: there is no context in a context-free parser.*

In the following we will complete the parser with a functional program to filter all the unsatisfiable interpretations due to inconsistent valuations of atoms from the derivation graph. The result should again be a sound decision procedure for satisfiability with some interesting aspects.

When we parse syntactically correct formulas with the parser generated from Figure 4 there are three possible categories of outcomes:

1. The first possibility is that the parser does not accept the sentence, in which case the formula is not satisfiable. Incidentally, this may happen for both open and closed formulas, but "enough" constants must be present in order to make the parse fail. In general, for a set of uniformly randomly breadth-first generated sentences, this amounts to about 25% of a sufficiently large set. However we expect that in real application scenarios, when formulas are generated by other means and the booleans constants are much more rare, the number comes closer to 0% than to 25%.
2. The second possible outcome is a single unambiguous derivation tree, which indicates the formula may be satisfiable. It is not 100% guaranteed to be satisfiable yet, because there could be contradictory valuations where in the same formula a `Atom P` is derived both by `True` and `False`. Unambiguous sentences are even

more rare as the previous unacceptable sentences, since unambiguity would only be possible with closed formulas or "almost closed" formulas with atoms only in very specific harmless positions, such as "true and p" .

3. The third and final option is when the parse produces an ambiguous forest of derivations where for each nested derivation of `True` and `False` non-terminals several possibilities exist. This is the common case. For each of the derivations represented by this packed and compressed directed acyclic graph, there could exist contradictory valuations of the previous kind.

Ergo, the parser for `True` simulates a polynomial over-approximation of satisfiability which in most cases says "maybe" (cases (3) and (2)), or "unsatisfiable" (case (1)).

For the second case (2) the algorithm in Figure 5 decides satisfiability in linear time. For any given atomic letter, if it is derived by the rule `True = Atom` it is interpreted as being valued true, while if it is derived by `False = Atom` it is interpreted as being valued false. If in any derivation tree a single letter is valued both true and false, the entire formula is necessarily not satisfiable. The Rascal code uses the deep match operator `/` to search in all sub-trees to find a match, detecting atoms produced by either the `True` or `False` non-terminal.

The rest of the essay we explore the practicalities of checking satisfiability of formulas which fall in the hard third category (3), for which we naturally need a generalization of the algorithm in Figure 5.

```

1  module UnambiguousSatisfiability
2
3  import ParseTree;
4  import OpenBooleans;
5
6  bool satisfiable(False f) = false;
7  bool satisfiable(True f) {
8      assert /amb(_) != f; // f is not ambiguous anywhere
9
10     trues  = {p | /(True) '<Atom p>' := f}; // collect true atoms
11     falses = {p | /(False) '<Atom p>' := f}; // collect false atoms
12
13     return trues & falses == {}; // intersection is empty
14 }

```

Figure 5: Satisfiability of unambiguous `True` formulas

Satisfiability of ambiguous formulas in $L(\text{True})$

Remember the ambiguous formula is stored as a cubic-sized directed acyclic graph, with internal "ambiguity nodes" wrapping sets of alternative derivation trees for sub-formulas. Such ambiguity nodes may be arbitrarily nested. Figure 6 graphically depicts such a "derivation forest" for the input sentence p and $(q \text{ or } \text{not } p)$, which to us is obviously satisfiable but nevertheless the generated forest contains one unsatisfiable derivation.

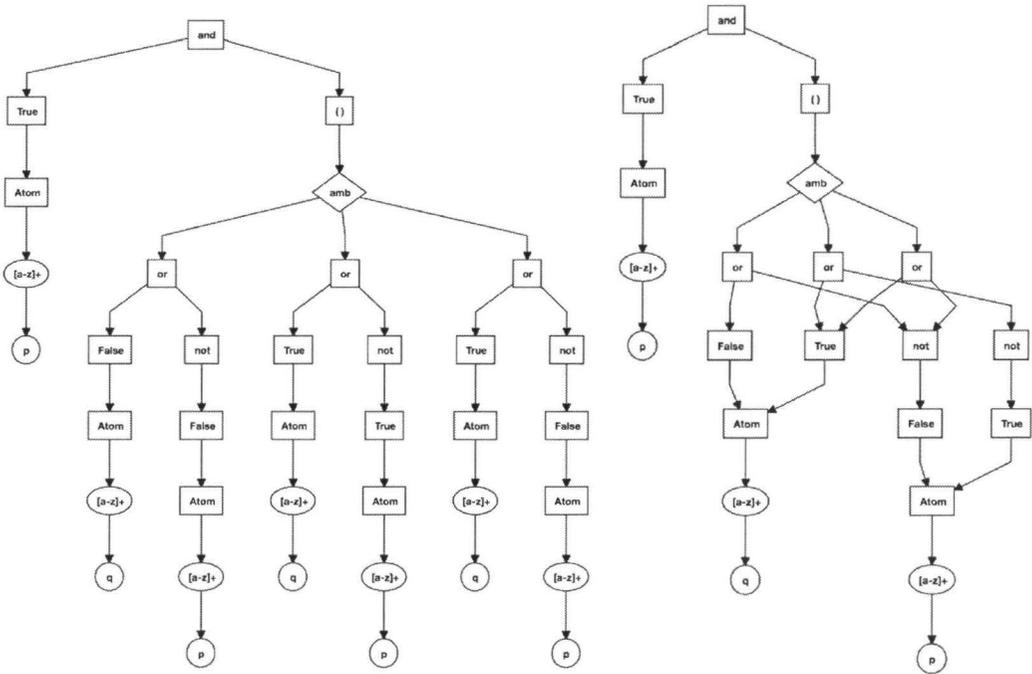


Figure 6: True derivations for p and $(q \text{ or } \text{not } p)$, non-shared and shared.

Now imagine any ambiguous sub-sentence derived by True : each such alternative derivation represents a formula valued as true, but internally each alternative will value the same atomic letters differently.

Lemma 6 (Alternatives) *We will make use of the following insights in creating a decision procedure for satisfiability of derivations for True :*

1. *Each alternative derivation for the same sub-sentence uses exactly the same atoms. This is by virtue of what an ambiguous derivation for a context-free grammar is: a derivation of the same sub-sentence for the same non-terminal using different rules.*
2. *For each pair of (different) alternatives in the same cluster, at least one of the valuations for a given atomic letter is different between the two derivations, but not all of them have to be different.*
3. *Each separate alternative may be internally inconsistent in the sense of category (2), which can be decided (if this alternative contains no nested ambiguity) using the procedure of Figure 5.*

The Rascal function `satisfiable` in Figure 7 defines a direct decision procedure for `True` derivation trees, by filtering all inconsistent valuations. If any derivation remains, the formula was satisfiable and the derivations represent possible valuations.

- We define the intermediate language `val` to encode valuations derived from sub-formulas. A `simple` valuation maps atomic letters to their Boolean value, while a `complex` valuation captures alternative valuations which need to be filtered in a later stage.
- The rewrite rules for the `complex` constructor are essential: they flatten out arbitrary nesting, filter intermediate contradictions in the presence of satisfying valuations and bubble up contradictions in case no valid valuations remain.
- The `val` function simply recurses over the formula and collects valuations. Most cases trivially fold over the formula structure, but the case for `amb` captures that ambiguous derivations correspond to alternate valuations, and the cases for all boolean connectors tie together *consistency* between the left part of a formula and the right part of the formula.
- The `merge` function finally has the responsibility for computing the cartesian product on complex valuations and detecting any inconsistent valuations.

Given that the original tree structure in memory can be expected to be an acyclic graph with a lot of sharing, and that Rascal functions can *memo-ize* results, the current implementations might perform reasonably well on "naturally occurring" formulas.

Still, there is room for improvement:

- The cross-product does not employ the symmetries of the commutative `and` and `or` operators yet, which could half the work. Similarly the algebraic laws of idempotence and associativity may be used to avoid redundant work.
- The work now splits over all 8 possible instances of `and` and `or`, while their semantics is not different. An intermediate pass to introduce more sharing between these 8 cases would have significant impact on efficiency.

Let's conclude this paper with the following claim:

Lemma 7 *We claim that the `satisfiable` function of Figure 7 is a sound implementation of SAT, when composed with a correct context-free general parser for the `True` non-terminal.*

```

1  module AmbiguousSatisfiability
2
3  import ParseTree;
4  import OpenBooleans;
5
6  bool satisfiable(False _) = false;
7  bool satisfiable(True f)  = val(f) != contradiction();
8
9  data Val
10     = simple(map[str atom, bool val] m)
11     | complex(set[Val] alts) | contradiction();
12
13 Val complex({*Val a, complex(set[Val] b)}) = complex({*a, *b});
14 Val complex({*Val a, contradiction()})    = complex(a);
15 Val complex({})                            = contradiction();
16
17 Val val(amb(set[Val] alts)) = complex({val(a) | a ← alts});
18
19 Val val((True) `true`)           = simple({});
20 Val val((False) `false`)        = simple({});
21 Val val((True) ``))      = simple({"<p>":true});
22 Val val((False) ``))    = simple({"<p>":false});
23 Val val((True) `<<True f>`))    = val(f);
24 Val val((False) `<<False f>`))  = val(f);
25 Val val((True) `not <False f>`) = val(f);
26 Val val((False) `not <True t>`) = val(t);
27 Val val((True) `set[Val] lm), complex(set[Val] rm))
40     = complex({merge(a,b) | <Val a, Val b> ← lm * rm});
41
42 Val merge(simple(map[str atom, bool val] lm),
43         simple(map[str atom, bool val] rm))
44     = {a ← (lm<atom> & rm<atom>) && lm[a] != rm[a]}
45     ? contradiction()
46     : simple(lm + rm);
47
48 Val merge(Val s:simple(map[str, bool] _),
49         Val a:complex(set[Val] _)) = merge(complex({s}), a);
50 Val merge(Val a:complex(set[Val] _),
51         Val s:simple(map[str, bool] _)) = merge(a , complex({s}));

```

Figure 7: Filtering inconsistent pairs of ambiguous derivations to decide satisfiability.

Conclusion

I admit this essay may contain the most roundabout route to deciding satisfiability ever conceived. Also I must admit that I am still not well-versed in the literature on satisfiability. Yet, it was a fun essay to write, and I hope you have had fun in reading it as well, Jan.

The algorithm first over-approximates satisfiability using context-free general parsing. Alternate valuations are simulated as ambiguous derivation trees by this process. A subsequent filtering phase removes all contradicting valuations, and if at least one remains the formula was satisfiable. The filter was implemented as a functional program in Rascal.

The presented "algorithm" retains the natural shape of an input formula without a priori normalization until the very latests moment. If different parts of large formulas have cohesion in terms of the set of variables they use and low coupling to other parts in the same sense, then this algorithm benefits by detecting contradictions only on the quickly computed common set of variables. I believe that typically formulas generated from statically analyzing program semantics have such modular sub-structures.

The practicality of first over-approximating SAT with a context-free general parser and subsequently filtering the inconsistent valuations remains to be seen.

References

- [1] A. Johnstone and E. Scott. Modelling GLL parser implementations. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, SLE, volume 6563 of *Lecture Notes in Computer Science*, pages 42–61. Springer, 2010.
- [2] A. Afroozeh, M. van den Brand, A. Johnstone, E. Scott, and J. Vinju. Safe specification of operator precedence rules. In *International Conference on Software Language Engineering (SLE)*, LNCS. Springer, 2013.
- [3] J. Aycock. Why Bison is becoming extinct. *ACM Crossroads*, Xrds-7.5, 2002.
- [4] J. Aycock and R.N. Horspool. Directly-executable earley parsing. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 299–243, Genova, Italy, 2001. Springer-Verlag.

- [5] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN notices*, 46(4):53–64, 2011.
- [6] D. Grune and C. Jacobs. *Parsing Techniques - a practical guide*. Springer-Verlag New York, Inc., 2006.
- [7] M. Johnson. The computational complexity of GLR parsing. In M. Tomita, editor, *Generalized LR Parsing*, pages 35–42. Kluwer, Boston, 1991.
- [8] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177. IEEE Computer Society, 2009.
- [9] E. Scott, A. Johnstone, and R. Economopoulos. BRNGLR: a cubic Tomita-style GLR parsing algorithm. *Acta informatica*, 44(6):427–461, 2007.
- [10] M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
- [11] J. van Eijck. Natural logic for natural language. In *Logic, Language, and Computation, 6th International Tbilisi Symposium on Logic, Language, and Computation, TbiLLC 2005, Batumi, Georgia, September 12-16, 2005. Revised Selected Papers*, pages 216–230, 2005.
- [12] J. van Eijck. Strategies in social software. In *Models of Strategic Reasoning - Logics, Games, and Communities*, pages 292–317. 2015.
- [13] J. van Eijck and C. Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

The Story of Urgh*

Albert Visser

This contribution is dedicated to Jan van Eijck on the occasion of his retirement. Jan and I share a long-standing interest in the interrelated subjects of dynamic semantics and the philosophy of language.

1 Urgh Mammoth Death

Urgh mammoth death said Grmph. The members of the Oowao tribe cheered. Urgh himself stood beaming to the side. Several Oowao hugged him. Killing a mammoth was no mean feat even among such accomplished hunters as the Oowao.

Urgh mammoth death said Grmph. The members of the Oowao tribe started wailing. Urgh's mangled body was carried by Fralp and Grad. The effects of mammoth tusks could be horrible. Urgh would be deeply missed among the Oowao. He was a capable hunter with a substantial contribution to the tribe's food supply. Moreover, his sunny character helped making the harsh life of the tribe more bearable.

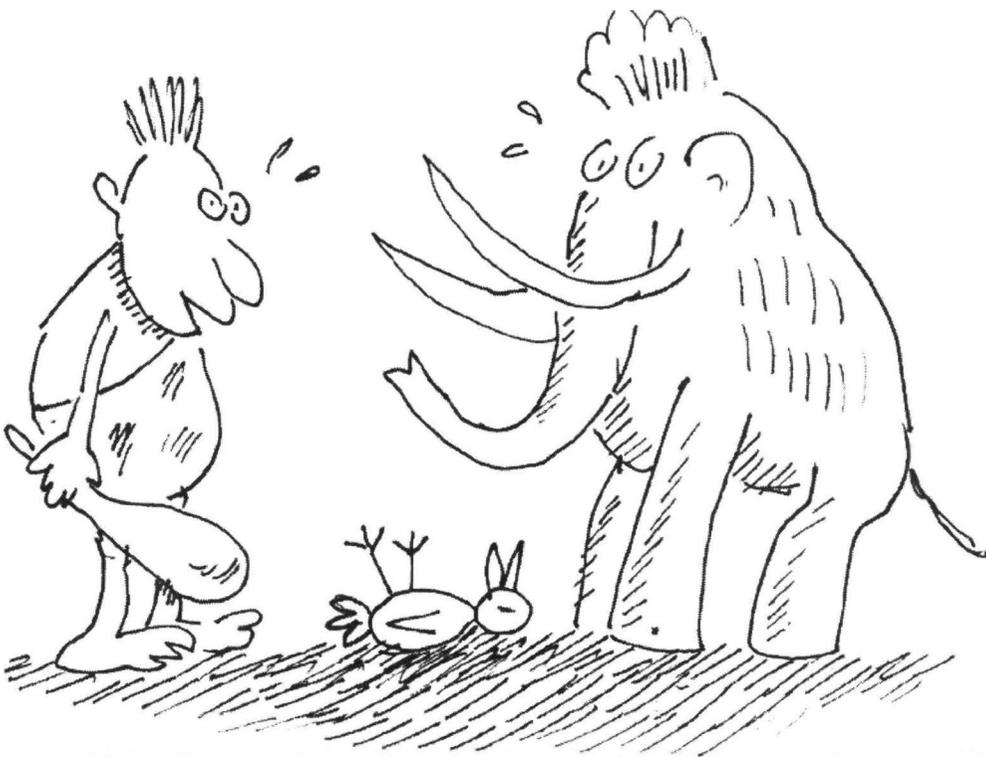
Urgh mammoth death said Grmph. The members of the Oowao tribe looked with awe at Urgh while he made his triumphant entrance seated on top of his favorite mammoth Cecilia. The two of them had played a decisive role in a pitched battle between the Oowao and the neighboring tribe of the Frksuml.

Urgh mammoth death said Grmph. The members of the Oowao tribe looked somewhat jealously at Urgh. To be assigned the leading role in the mammoth hunt is a rare honor, especially for one so young. In spite of his excitement Urgh managed to keep a stern and dignified demeanor, worthy of the important task just given him.

Urgh mammoth death said Grmph. The members of the Oowao tribe sat around the fire. It was a dark and stormy night. The Oowao listened attentively to the story about their favorite mythical hero Urgh who allegedly killed a mammoth with his bare hands. Later

Urgh would become a god and most members of the tribe would convert to Urghism. But that was many years in the future. For now they just enjoyed the warmth of the fire and savored the gripping story.

Urgh mammoth death said Grmph. The members of the Oowao tribe burst out laughing. Grmph's rueful tone made clear who he thought would be the dealer and who the recipient of death. Everybody had been somewhat annoyed by Urgh who, shaking his spear, had boastfully claimed *Urgh mammoth death* at the beginning of the hunt. Grmph's playful reversal of the interpretation of Urgh's words released the tension. Chuckling the Oowao continued on their way to the hunting ground. Only Urgh was a bit of a grump for the rest of the day.



2 A Brief Introduction to Oowao

In the beginning was the meaningful word.

This piece contains my philosophical fantasy about an ur-language. My main aim here is to bring the fantasy on the board. The central message of the fantasy is: *semantics first*. There is a good case that semantics is first in language in a conceptual sense. I want to say

that it is also first in language genesis.

Of course, there is the question whether the fantasy can be true. However, the story already has bite, when it is merely coherent. If the story is coherent, then it is worthy of further exploration.

There are all kinds of arguments pro and contra the coherence of the story. I decided not to address these in the restricted scope of this article. Also, I imagine that a more informed fantasy could be possible. For example, there is a lot of relevant work by Michael Tomasello and his group that I am not discussing. One important message of Tomasello's work is that a high level system of intention tracking was already in place before the advent of language. Some such fact is surely needed for my story to make sense. The study of Pidgins and Creoles (especially of the famous Nicaraguan sign language) will surely be relevant, even if the data from this study should be evaluated with great care. After all, even the Nicaraguan sign language came into being in an environment where there was already language.

The Oowao language has no syntax. It is a holophrastic language. There are no sentences in our sense. Single words have as much claim to being sentence analogues as certain natural word groupings like *Urgh mammoth death*. As a consequence, Oowao has nothing like recursion. On the other hand, the language is not finite. Any string of words can occur as the subsequent words of a discourse¹ and there are potentially infinitely many strings. The Oowao manage to say a lot with their discourses.

Speaking a word in Oowao is making the thing it stands for present.² Saying *mammoth* brings a mammoth before the mind's eye. Oowao has no distinction between verbs, nouns and adjectives (qua syntactical categories). The word *death* signals the presence of death. In context, it can take the role of our verb *kill* or our noun *death* or our adjective *dead*. The word *Urgh* does function as a name semantically in the sense that, when we paraphrase an Oowao discourse, our word corresponding with an Oowao use of *Urgh* will be our name *Urgh*.³ However, there is nothing in Oowao like the syntactical role of a name in a sentence.⁴

Saying a word can have many functions. Which function depends heavily on context.

Suppose *Urgh* approaches *Grmph* who is talking to a group of people. *Grmph* says *Urgh*. If the members of the group do not know *Urgh*, *Grmph*'s pronouncement serves to make *Urgh*-qua-*Urgh* present for these members. If *Urgh* is known, *Grmph*'s pronouncement

serves as a greeting and an acknowledgment. It makes Urgh present in the shared attention of the group, a group that now includes Urgh himself.

Suppose Grmph says *Urgh* when no Urgh is anywhere in sight. This could have all kinds of meanings and its interpretation can make a substantial demand on the contextual knowledge of the receiver of this short message. If Grmph is sitting opposite to Xixia and is, while pronouncing *Urgh*, looking expectantly at her, the pronouncement could well mean: bring Urgh. Here the imagined situation of Urgh's presence should be made real: *adequatio rei ad intellectum*. Alternatively, the cave could have been a mess. It is known that Urgh was just in the cave. Also Urgh is known to be not a great fan of cleaning up. Perhaps, Grmph says *Urgh* with resignation in his voice. In this case surely a situation is envisaged in which Urgh made a mess and left without cleaning up. The criterium of correctness of Urgh's utterance in this case is that Urgh indeed made a mess: *adequation intellectus ad rem*.

But what connects the words? What makes a Oowao discourse more than just a bunch of disconnected words? What is the intra-word-glue?

Let us look at our primary example *Urgh mammoth death*. The crucial thing is that the closeness of the words suggests co-presence of the meanings. Of course, *Urgh mammoth death* allows us to imagine completely disjoint Urgh-, mammoth- and death-situations, but the contiguous utterance of *Urgh mammoth death* makes an interpretation where Urgh, a mammoth and death are brought together, so to say *on one tray* or *in one location*, an obvious choice. There we have it: a situation containing Urgh and a mammoth and death. The Oowao know many such situations: Urgh finding a dead mammoth, Urgh killing a mammoth, Urgh being killed by a mammoth; take your pick. Which situation to choose? Here context kicks in, as illustrated in the examples of Section 1.

If two sequences of words are uttered by Grmph with some time-distance between them, in many cases the obvious choice will be to imagine the meaning of each sequence to be on a different trays. So, something a bit like syntax is emerging in Oowao. Word sequences that are viewed as pertaining to copresent situations are grouped together by being temporally contiguous. However, this minimal kind of syntax is perhaps more like discourse structure than like sentence structure. Still one can imagine that sentences could grow out of word sequences with co-present interpretation in a further development of Oowao.

There is the possibility that in hearing a discourse consisting of a long sequence of words, the Oowao will reset an internal time parameter, thus making the event associated to a word uttered later occur later than an event associated to a word uttered earlier.

The various readings of *Urgh mammoth death* invite some further reflection. We have three copresent situations: an Urgh-situation, a mammoth-situation and a death-situation. The Oowao will know that the copresent situations really form one inclusive situation in which the three sub-situations are each contained in their own way. For example, if Urgh killed the mammoth, then Urgh is the agent, the Mammoth is the patient and death is the Mammoth's fate. Thus, the way the overall situation is built-up from its 'constituents' foreshadows the thematic roles in sentences in more familiar languages.

The intellectual ability to assemble various partial situations into one whole is the converse of a basic ability needed for language: the ability to disassemble the given reality-as-a-whole into different meaningful parts.

The interword glue is founded in the fact that words *ab initio* only reflect partial situations, situations that are part of a more encompassing one.

3 Linking

Diachronic samenesses are at the heart of language. We will denote the process of identifying things as the same across time by the expression *linking*.

When are two uses of a word uses of the same word? There should be something identifiable in reality to help us identify uses as uses of the same word. In spoken language this is a sound. We note that the story about what precisely constitutes a sound(-type) is far from easy. This story is studied in phonetics and phonology. We will ignore all complexities here.

There can be considerable variation in the sounds activating the same word. Moreover, if an individual sound only accidentally conforms to the sound-type connected with a given word, then that individual sound does not 'carry' or 'label' or 'activate' the word at hand. (*) For an individual sound x of a sound-type X to activate the word ξ we need intentional production: the speaker must intend to say, using the sound x of type X , the word ξ that is for her connected with earlier uses x' of type X .⁵ We note the circularity in our description: to be an occurrence of ξ , the intention to produce ξ is needed. This circular

identity is typical for objects of the social ontology: for an occurrence to be an occurrence of *it*, the occurrence has to be intentionally produced to be an occurrence of *it*. There can be considerable variation in the sounds activating the same word. Moreover, if an individual sound only accidentally conforms to the sound-type connected with a given word, then that individual sound does not 'carry' or 'label' or 'activate' the word at hand. (*) For an individual sound x of a sound-type X to activate the word ξ we need intentional production: the speaker must intend to say, using the sound x of type X , the word ξ that is for her connected with earlier uses x' of type X .⁵ We note the circularity in our description: to be an occurrence of ξ , the intention to produce ξ is needed. This circular identity is typical for objects of the social ontology: for an occurrence to be an occurrence of *it*, the occurrence has to be intentionally produced to be an occurrence of *it*.

Consider the following discourse *Urgh seeing mammoth death*. This could, in the right context, be interpreted as *Urgh first sees and then kills a mammoth*. Now suppose that Grmph says *Urgh seeing mammoth ... Urgh mammoth death*. In this case the first use of *mammoth* and the second one are linked with each other as the same word. But under a reasonable interpretation there is also an additional linking in which the hearer understands that the second occurrence of *mammoth* stands for the same mammoth situation or for a mammoth situation that is the temporal successor of the first one. Thus we could paraphrase the discourse as *Urgh sees a mammoth. He kills it*.

The default is that there is no linking of the additional kind. Distinct word uses on their own tend to present mutually distinct situations. A reason is needed to decide that two uses exhibit a tighter link. Giving the discourse more structure demands cognitive labour.

For example, *Urgh mammoth mammoth death* could very well mean: Urgh kills two mammoths and *Urgh mammoth mammoth mammoth death* could either mean: Urgh kills three mammoths, or: Urgh kills many mammoths.⁶ We note that in some situations *mammoth, mammoth, mammoth* could mean presence of the same mammoth but over an extended time.

Daniel Everett [1, p227] discusses the following conversation from father to son in (translated) Pirahã, a language that is supposed to have no (or very little) recursion.

Hey, Paitá. Bring back some nails. Dan brought those very nails. They are the same.

We would render this in English as:

*Hey Paitá. Bring back the nails that Dan brought.*⁷

We note that the example illustrates that tighter linking can take over the role of recursion. The example makes it rather plausible that language with finitely many sentences can do anything a language with infinitely sentences can do. One simply transfers work done by sentence structure to less regimented discourse structure.

The sentence in our kind of language is just a clever pre-programmed local way to convey information.

I tend to think that words being on the same tray, thus signaling a co-present situation, is also a form of tighter linking. Viewed in this way syntactic structure becomes a special case of linking.

Remark: Once a philosopher remarked to me that he considered the discourse referent to be as problematic as possible worlds. My feeling is that the discourse referent can hardly be more problematic than the notion of word. It is a tighter sameness accross uses of various uses of words. *A man ... he* and *Xixia ... she*. There is a sameness preserved from the use of *a man* to the use of *he* and similarly from *Xixia* to *she*. A discourse referent is just like a word only somewhat less faithful to a sound-type. Like the word, the discourse referent should be a respected citizen of the social ontology.

4 Eccentricity

One thing the Oowao's need in order to have language is something like Plessnerian *eccentricity*. Eccentricity is the ability to see oneself as part of the world, as located in and limited in space and time. To be able to do this we need to be able to place ourselves in a position outside ourselves.

Eccentricity is needed for language. The basic function of saying a word is *making a thing present*. We could as well say *imaging oneself as being present where the thing is*.

Language is a form of mental travel, in space, in time and even in worlds of fantasy.

In saying something we intend to bring the thing mentioned into our common space. This involves seeing the other person as something just like oneself. We understand that the other sees us just like we see the other. The eccentric position lies at the basis of the possibility of such reciprocity.⁸

Language, in its turn, strengthens our eccentric position. Language is great vehicle for space-, time- and worlds-of-fantasy-travel.

5 The Primacy of Semantics & the Nature of Syntax

The language creation myth about the Oowao stresses the idea of the primacy of semantics. In the beginning was the meaningful word. The meaning associated with the utterance of a word is not just a denotation, but rather primarily (the type of) the act of making the reference present.

Syntax develops from semantics. Moreover syntax cannot be clearly separated from semantics. I already hinted at the genesis of the argument place. Argument places are, or so I believe, derived from the constitution of total situations in reality. For example, the agens and the patiens roles are anchored in the constitutive structure of action.

But what about the embedding of sentences in sentences? Here is my speculative view about embedding. The brain supports a massively parallel process. It can easily produce several sentences at the same time. *Urgh he was very angry came in shouting.* We speak thus when we are agitated, trying to pronounce several sentences at once. When we do that, we have several trays or locations where situations are brought together in the air. Our short-term memory is good enough to keep track of them as long as we do not stack too many. Embedded sentences are a ritualized frozen way to put one sentence on top of another. The embedded location does not even have to consist of contiguous words. For example, if we say *Hob believes that a witch has made the milk sour and Nob thinks that she also made his dog sick*, then Hob believes and Nob thinks are on one level, but the witch is in the fictional world where witches exist that is introduced in the discourse. This is what makes the anaphoric reference to the non-existent witch possible.

Certain local patterns of linking and certain patterns of embedding get frozen into syntax. We can understand syntax by formulating rules. However syntax is not caused by or preprogrammed via some set of rules in the brain—supposing such an idea would be understandable at all, which I doubt.

6 The Functions of Language

What is language for? The question has a misleading reading. It suggests a user (we) and a tool (language). The user, in some sense, has to precede the tool. However, language is a constitutive element of what we are. It is just as senseless to ask what use we are for

language. One could say that Man and Language are *causae mutuae*.

If we do not interpret the *for* as *for us*, we may pinpoint some functions of language. We have:

- *Contentual communication*. Information about what is going on is essential for survival.
- *Phatic communication*. This is sometimes translated as *small talk*. But it contains much more. When Grmph says *Urgh* to acknowledge Urgh's presence, this helps consolidate relations in the group. It plays a role in the creation of a *we*. The stories told by Grmph diminish boredom and therewith mischief.

One of the important features of the growth of humanity is the ability to live together in larger groups. Language helps to make this possible.

- *Reasoning*. Reasoning in language can take place independent of the communicative functions.

It was my original hunch that the first function, to wit contentual communication, provided the first selective pressure in favor of language. However, our examples suggest that language needs reasoning. Some of that reasoning could itself employ language. Also the very idea of an assertion presupposes a lot of reciprocity. We say things with the intention to share something. Reciprocity is strengthened by phatic communication. So, perhaps, all three functions were present from the beginning and all played a role in the selective pressure, both for genetic and cultural change. Thus, language could grow both in complexity and in the size of its group of users. Thus, persons, *we*, came into being.

7 Happiness

It seems that Oowao is only suitable for wishes straight from the heart. So:

Jan friend Albert wishing happiness happiness happiness.

Notes

* . Joeri Niels comment comment comment thanking thanking thanking. ↔

1. Obviously the same string of words can be associated to different discourses. String individuation is coarser than discourse individuation. ←
2. The English expression *to present a thing* does not do quite what I want. The Dutch expression *aanwezig maken* captures the idea rather neatly. ←
3. Of course, a word group like *this is Urgh* may be our translation of *Urgh*. This at least contains our name *Urgh*. ←
4. In historical languages the semantical role of a name is never just to refer to the object. To think this would be to confuse names with constants in predicate logic. The making present function of names is preserved in historical languages. Saying a name makes the named salient in the discourse. ←
5. Of course, the story is more complicated. If we take (*) too literally, there could, for example, be no automatic production of words by a computer. However, for the purposes of this note, I will leave it at this first attempt to describe word-identity. ←
5. Of course, the story is more complicated. If we take (*) too literally, there could, for example, be no automatic production of words by a computer. However, for the purposes of this note, I will leave it at this first attempt to describe word-identity. ←
6. So, the default would be like the use of variables in the Backus-Naur form. ←
7. If you try to approximate the sentence in Oowao, you see how complex it really is. The switch from an imperative to a past tense, the plural *nails*: these are all difficult to present in Oowao. *Dan bringing nail nail nail Paitá bringing* is the closest I seem to be able to get. This discourse seems to be rather demanding on the interpretative powers of (the Oowao version of) Paitá. ←
8. The genesis of language is, according to Tomasello, part of the more encompassing genesis of human cooperation. It seems to me that eccentricity is one of the preconditions for the possibility of cooperation. ←

References

- [1] Daniel Everett. *Don't sleep, there are snakes*. Profile Books Ltd., London, 2009.

Route 65

(for Jan, on the occasion of his retirement of the CWI)

Heleen Verleur
Januari 2017
www.heleenverleur.org

♩ ± 176

Quavers played uneven, with a "jazzy" feeling

Clarinet in B \flat

mp

All quavers played uneven, with a "jazzy" feeling

Piano

mp

4

Cl.

mf

Pno.

mf

7

Cl.

f

Pno.

11

Cl.

Pno.

mf

f

tr

15

Cl.

Pno.

mf

f

tr

19

Cl.

Pno.

mf

f

tr

22

Cl.

Pno.

mf

f

tr

26

Cl. *mf*

Pno. *mf*

29

Cl.

Pno.

32

Cl.

Pno. *p*

35

Cl. *mf*

Pno. *mp*

39

Cl. *mp*

Pno. *mp*

43

Cl. *f*

Pno. *f*

LH

48

Cl. *f*

Pno. *f*

LH

52

Cl. *f*

Pno. *f*

55

Cl.

Pno.

58

Cl.

Pno.

61

Cl.

Pno.

66

Cl.

Pno.

72

Cl.

Pno.

mf

RH

78

Cl.

Pno.

f

82

Cl.

Pno.

86

Cl.

Pno.

f

90

Cl.

Pno.

94

Cl.

Pno.

98

Cl.

Pno.

102

Cl.

Pno.

105

Cl.

Pno.

2 4 3 1 4 3 2 1 4 5 4 3 2 1 3 2

p *mf*

108

Cl.

Pno.

mf *mf*

112

Cl.

Pno.

p

115

Cl.

Pno.

8va *tr* *p*

118

Cl.

Pno.

Poco acc.

(8)

122

Cl.

Pno.

125

Cl.

Pno.

128

Cl.

Pno.

140

Cl.

Pno.

mf

143

Cl.

Pno.

146

Cl.

Pno.

149

Cl.

Pno.

168

Cl.

Pno.

172

Cl.

Pno.

176

Cl.

Pno.

180

Cl.

Pno.

184

Cl.

Pno.

4 5 4 3 2 1 3 2

188

Cl.

Pno.

ROUTE 66

f *p*

191

Cl.

Pno.

194

Cl. *mp*

Pno. *mp*

197

Cl. *mf*

Pno. *mf*

tr

200

Cl. *f*

Pno. *f*

tr

204

Cl.

Pno.

208

Cl.

Pno.

tr

f

211

Cl.

Pno.

mf

mf

214

Cl.

Pno.

f

f

218

Cl.

Pno.

tr

f

222

Cl.

Pno.

226 *Poco rit.*

Cl.

Pno.

230

Cl.

Pno.

f *p*

f *p*

Ped.

Route 65

(for Jan, on the occasion of his retirement of the CWI)

$\text{♩} \pm 176$ Quavers played uneven, with a "jazzy" feeling

6 *mp* *mf*

14 *f* *mf*

20 *f*

26 *mf*

29 *mf*

34 *mp* *mf*

39 *mp*

44 *f*

47 *f*

Clarinet in B \flat

51 *f*

56 *p*

61 *p*

68 **2**

74 *mf*

78 *f* **2**

84 *f* **2**

90 **2**

97 **2**

103 **2**

110 *mf* **4**

Detailed description: This page of a musical score for Clarinet in B-flat contains ten staves of music, numbered 51 to 110. The music is written in treble clef with various time signatures: 6/8, 3/4, 6/4, 5/4, and 4/4. The score includes dynamic markings such as *f* (forte), *p* (piano), *mf* (mezzo-forte), and *mf* (mezzo-forte). It also features articulation marks like accents (>) and slurs. Rehearsal marks with the number '2' are placed above the staff at measures 68, 78, 84, 90, 97, 103, and 110. The piece concludes with a double bar line and a 5/4 time signature at the end of measure 110.

178

182

188 Φ ROUTE 66

191

194

198

202

208

212

216

221

Musical notation for measures 221-225. Measure 221: Treble clef, key signature of one sharp (F#), quarter notes G4, A4, B4, C5. Measure 222: Treble clef, quarter rest, eighth notes G4, A4, B4, C5. Measure 223: Treble clef, quarter rest, eighth notes G4, A4, B4, C5. Measure 224: Treble clef, quarter rest, eighth notes G4, A4, B4, C5. Measure 225: Treble clef, key signature change to one sharp (F#), quarter notes G4, A4, B4, C5. Performance markings include slurs and accents (>) under the eighth notes in measures 222-225.

226 *Poco rit.*

Musical notation for measures 226-229. Measure 226: Treble clef, key signature of one sharp (F#), quarter notes G4, A4, B4, C5. Measure 227: Treble clef, quarter rest, quarter notes G4, A4, B4, C5. Measure 228: Treble clef, quarter rest, quarter notes G4, A4, B4, C5. Measure 229: Treble clef, quarter rest, quarter notes G4, A4, B4, C5. Performance markings include a slur under the quarter notes in measure 226 and a long horizontal line under the quarter notes in measures 227-229.

230

Musical notation for measures 230-233. Measure 230: Treble clef, quarter notes G4, A4, B4, C5. Measure 231: Treble clef, quarter note G4. Measure 232: Treble clef, quarter rest, quarter note G4. Measure 233: Treble clef, quarter rest, quarter note G4. Performance markings include dynamics *f* under measure 231 and *p* under measure 232.

