Contents

0.1 Stuff to fit in

3-SAT

Notes on a lecture given by Cook of Cook's theorem.

n variables and $c \cdot n$ clauses. Let F be an instance. Then $P(F\text{is satisfiable}) \to 0$ as $c \to \infty$. Phase change at ~ 4.26 gets steeper as $n \to \infty$.

Cook sez: a **propositional proof system** is a polytime surjection $f : \Sigma^* \twoheadrightarrow$ set of tautologies. ("x is a proof of $f(x)$").

A proof system is **super** if there is a polynomial p such that every tautology of length n has a proof of length n. So NP = co-NP iff there is a super proof system. Generally it seems quite hard to show that a proof system is not super. (If NP = co-NP but NP $\neq$ Pthen the short proofs of tautologies cannot be found effectively (that is, in polytime)

(I find in my scribbles: Matrices $AB = I \to BA = I$ can be coded over F_2 but these tautologies do not seem to have polytime proofs . . . not certain tho' . . . What did i mean???)

2004 Paper 4 Q 9

Here is a slick way to do it. Given a stream $F = \langle f_i : i \in \mathbb{N} \rangle$ set

$$D(F) =: \lambda n. f_n(n) + 1.$$

Evidently $D(F)$ is distinct from all the functions in F.

Now let F_0 be the sequence of f_i we are presented with initially.

Then set $g_0 = D(F_0)$, and thereafter $F_{i+1} = g_i :: F_i$ (where the double colon is `cons` as in `ML`). For all $i \in \mathbb{N}$, set $g_i =: D(F_i)$.

Chapter 1

Languages and Automata

1.1 Questions

1987 p6 q3; 1988:2:3; 1988:2:11; 1989:6:12*; 1989:4:11; 1991 p4 q6; 1991 p11 q6; 1992 p4 q9*; 1992 p11 q9; 1993 p5 q12*; 1993 p6 q12; 1994 p10 q4; 1994 p4 q3 (not for 1a); 1994 p3 q3 1993 p5 q12; 1993 p6 q12; 1994:3:3; 1994:4:2; 1995:2:8; 1995 p3 q3; 1995:4:3; 1996:2:8; 1997:2:7. 1998:2:7

These are mostly 1b questions. There is actually not a great deal of material that used to be covered in 1b that is not covered in the 1a course—all i can think of is context-free languages and pushdown automata. The same goes for questions below. I have put asterisks against the numbers of tripos questions for which i have model answers.

1. Prove that $L((r|s)^*) = L((r^*s^*)^*)$ (Use induction on word length)
2. Prove that $L((rs^*)^*) \subseteq L((r^*s^*)^*)$ but that the reverse inclusion does not hold.
3. Describe[1] deterministic automata to recognise the following subsets of $\{0,1\}^*$:
   (a) The set of all strings with three consecutive 0's; provide a regular expression corresponding to this set as well;
   (b) The set of all strings w such that every set of five consecutive characters in w contains at least two 0's;
   (c) The set of all strings such that the 10th character from the right end is a '0'; provide a regular expression corresponding to this set as well. *For pedants:* This could mean one of two things. Answer both of them.
4. Let L be a regular language over an alphabet Σ. Which of the following are regular languages?
   (a) $\{w \in \Sigma^* : (\exists u \in \Sigma^*)(wu \notin L)\}$
   (b) $\{w \in L : (\forall u \in \Sigma^*)((\text{length}(u) > 0) \rightarrow wu \notin L)\}$
   (c) $\{w \in L : (\forall u, v \in \Sigma^*)((w = uv \wedge \text{length}(u) > 0) \rightarrow u \notin L)\}$
   (d) The preceding question has a typo in it. Find it.
   (e) S, an arbitrary subset of L.
   (f) $\{w \in \Sigma^* : (\exists u, v \in \Sigma^*)((w = uv) \wedge (vu \in L))\}$ (*hint*: needs a different approach ...)

[1] This word is very carefully chosen!

5. A combination lock has three 1-bit inputs and opens just when it receives the input sequence `101`, `111`, `011`, `010`. Design a finite deterministic automaton with this behaviour (with accepting state(s) corresponding to the lock being open).

6. Let Σ be an alphabet and let B and C be subsets of Σ^* such that the empty string is not in B. Let $X \subseteq \Sigma^*$ and show that if X satisfies the equation $X = BX \cup C$, then $B^*C \subseteq X$ and $X \subseteq B^*C$, i.e. the unique solution is $X = B^*C$. [Hint: use induction on number of "blocks".]

7. Show that if in the previous question we allow $\Lambda \in B$, then $X = B^*D$ is a solution for any $D \supset C$.

8. Let $A = \{b, c\}$, $B = \{b\}$, $C = \{c\}$. Find the solutions $X_1, X_2 \subset A^*$ of the following pairs of simultaneous equations: (i) $X_1 = BX_1 \cup CX_2$; $X_2 = (B \cup C)X_1 \cup CX_2 \cup \{\Lambda\}$ (ii) $X_1 = (BX_1 \cup \{\Lambda\})$; $X_2 = BC(X_1 \cup \{\Lambda\})$.

9. There is an alphabet Σ with six letters a, b, c, d, e and f that represent the six rotations through $\pi/2$ radians of each face of the Rubik cube. Everything you can do to the Rubik cube can be represented as a word in this language. Let L be the set of words in Σ^* that take the cube from its initial state back to its initial state. Is L regular? If you have had a sleepless night over this you may consult the footnote for a hint.[2]

10. Can you construct an FDA to recognise binary representations of multiples of 3? You may assume the machine starts reading the most significant bit first.

11. For which primes p can you build a FDA to recognise decimal representations of multiples of p? How many states do your machines have?

12. Let q be a number between 0 and 1. Let L be the set of sequences $s \in \{0, 1\}^*$ such that the binary number between 0 and 1 represented by s is less than or equal to q. Show that L is a regular language iff q is rational. What difference would it have made if we had defined L to be be the set of sequences $s \in \{0, 1\}^*$ such that the binary number between 0 and 1 represented by s is less than q.

13. Give regular grammars for the two following regular expressions over the alphabet $\Sigma = \{a, b\}$ and construct finite non-deterministic automata accepting the regular language denoted by them:

    (a) $ba|(a|bb)a^*b$

    (b) $((a|b)(a|b))^*|((a|b)(a|b)(a|b))^*$

14. For each of the following languages either show that the language is regular (for example by showing how it would be possible to construct a finite state machine to recognise it) or use the pumping lemma to show that it is not.

    (a) The set of all words not in a given regular language L.

    (b) The set of all palindromes over the alphabet a, b, c (a palindrome is a word that is unchanged when reversed, for example, *abcbabcba*).

    (c) If L is a regular language, the language which consists of reversals of the words in L; thus if L contains the word *abcd*, then the reversed language L^R contains *dcba*.

[2] If this is to be a regular language, there must be a FDA that recognises it. What might this FDA be?

(d) Given regular languages L and M, the set of strings that contain within them first a substring that is part of language L, then a substring from M; arbitrary characters from the alphabet a, b, c are allowed before, between and after these strings.

(e) Given regular languages L and M, the set of strings that contain within them some substring which is part of both L and M.

15. What is the language of boolean (propositional) logic? Is it regular? What about the version without infixes ("Polish notation") What about reverse polish notation?

16. A "Muller C-element" is a device which receives two streams of binary digits ($x_0x_1x_2\ldots$ and $y_0y_1y_2\ldots$) and outputs a stream of binary digits $z_0z_1z_2\ldots$ satisfying the relation

$$z_{n+1} = \begin{cases} x_{n+1} & \text{if } x_{n+1} = y_{n+1}, \\ z_n & \text{if } x_{n+1} \neq y_{n+1} \end{cases}$$

Design a Moore machine with this behaviour.

17. Give context-free grammars generating the following languages:

(a) $\{a^p b^q c^r | p \neq q \text{ or } q \neq r\}$

(b) $\{w \in \{a, b\}^* |\ w \text{ contains exactly twice as many } a\text{s as } b\text{s}\}$

18. Let M be a finite deterministic automaton with n states. Prove that $L(M)$ is an infinite set if and only if it contains a string of length l with $n \leq l < 2n$.

19. The **interleaving** of two languages L and M is the set of all words that can be obtained from a word in L and a word in M by interleaving the two words in the way that people shuffle together two halves of a pack of cards. Prove that

(a) The interleaving of two regular languages is regular

(b) The interleaving of a regular and a context free language is context free

(c) The interleaving of two context free languages is not always context-free.[3]

20. Does the set of strings in $\{a, b, c\}^*$ which have as many as as bs and cs put together make a regular language?

21. Let K, L and M be regular languages. Is $\{u \in L : (\exists v \in K)(uv \in M)\}$ a regular language?

22. Is the language of Roman numerals regular?

Fit in somewhere: overloading of juxtaposition: $abc\ uvw$ and ABC. I think this is something to do with quasiquotation ...

[3] There is a pumping lemma for context-free languages which is not in the course. With the hint that $a^n b^m c^n d^m$ is not context-free it all becomes terribly easy!

1.2 Answers

Question 1.1

Let $\Sigma = \{r, s\}$. If you sit and stare at these two expressions long enough you will come to the conclusion that they both denote Σ^*. The way to prove they are equal to each other is therefore to prove they are both equal to Σ^*. Clearly $L((r|s)^*) \subseteq \Sigma^*$ and $L((r^*s^*)^*) \subseteq \Sigma^*$ so all we have to do to prove $L((r|s)^*) = \Sigma^*$ and $L((r^*s^*)^*) = \Sigma^*$ is prove the reverse inclusions:[4] $\Sigma^* \subseteq L((r|s)^*)$ and $\Sigma^* \subseteq L((r^*s^*)^*)$. We do these two by induction on the length of strings in Σ^*. If you want to be formal about it you can properly describe what we are about to do as a proof by induction on k that $\Sigma^k \subseteq L((r|s)^*)$ and $\Sigma^k \subseteq L((r^*s^*)^*)$. This is certainly true for the empty string ($k = 0$). Suppose it is true for all strings of length k. Let wx be a string of length $k + 1$, where w is of length k and x is r or s. By induction hypothesis w belongs to both $L((r|s)^*)$ and $L((r^*s^*)^*)$. Since w is of length k and belongs to $L((r|s)^*)$ it must be k occurrences of r or s. If we stick an r or an s on the end we get $k + 1$ occurrences of r or s. If w belongs to $L((r^*s^*)^*)$ it is a number (n, say) of concatenated occurrences of (r^*s^*). Now r and s are both instances of (r^*s^*) so wx is also a number ($n + 1$ in fact) of concatenated occurrences of (r^*s^*).

Question 3

Give finite deterministic automata accepting the following languages over $\Sigma = \{0, 1\}$

1. The set of all strings with three consecutive 0s, and provide a regular expression for this language.
2. The set of all strings such that every block of five consecutive symbols contains at least two 0s.
3. The set of all strings such that the 10th symbol from the right end is 1, and provide a regular expression for this language.

Question 3 (a) Strings with three consecutive 0s

The FDA $M = (Q, i, \Sigma, F, \delta)$ would be defined as follows:

$$\begin{aligned} Q &= \{A, B, C, D\} \\ i &= A \\ \Sigma &= \{\texttt{0, 1}\} \\ F &= \{D\} \end{aligned}$$

The transition function δ would be defined such that upon receiving a zero the machine would move to the next state, but given a 1 it would move back to the initial state.

Therefore δ is given by:

	0	1
A	B	A
B	C	A
C	D	A
D	D	D

The regular expression desired is $(0|1)^*000(0|1)^*$

[4] because set inclusion is antisymmetrical. Have you forgotten this word already?

Question 3 (c)

The 10th character from the right is a zero. (1|0)*0(1|0)(1|0)(1|0)(1|0)(1|0)(1|0)(1|0)(1|0)(1|0)(1|0)

Question 4

Question 4(b)

The kiddies are liable to get confused beco's this language is always finite and usually empty. They are liable to interpret silly answers as `fail` and say 'no' when they should in fact say 'yes'.

Question 4(d)

It should be "length$(v) > 0$".

Question 4(f)

> Let $\mathcal{L}$ be an arbitrary regular language $\subseteq \Sigma^*$. Is the following also a regular subset of Σ^*?
>
> $$\{w : (\exists v, u \in \Sigma^*)(w = uv \wedge vu \in \mathcal{L})\}$$

Answer: yes. There is a very cute way of doing this with regular expressions: the regular expression corresponding to L is finite and an be chopped in two and rearranged in only finitely many ways, and we put all those finitely many topped-and-tailed versions together with slashes between them.

Question 5

> A combination lock has three 1-bit inputs and opens just when it receives the input sequence `101`, `111`, `011`, `010`. Design a finite deterministic automaton with this behaviour (with accepting state(s) corresponding to the lock being open).

The FDA $M = (Q, i, \Sigma, F, \delta)$ would be defined as follows:

$$\begin{aligned} Q &= \{A, B, C, D, E\} \\ i &= A \\ \Sigma &= \{\texttt{000, 001, 010, 011, 100, 101, 110, 111}\} \\ F &= \{E\} \end{aligned}$$

	000	001	010	011	100	101	110	111
A	A	A	A	A	A	B	A	A
B	A	A	A	A	A	B	A	C
C	A	A	A	D	A	B	A	A
D	A	A	E	A	A	B	A	A
E	E	E	E	E	E	E	E	E

Question 6

Show $B^*C \subseteq X$. $B^*C \subseteq X$ is just $(\forall w)(w \in B^*C \rightarrow w \in X)$. To prove this we prove $(\forall n \in \mathbb{N})(\forall w)(w \in B^nC \rightarrow w \in X)$. The case $n = 0$ is just $(\forall w)(w \in C \rightarrow w \in X)$, which is $C \subseteq X$ which follows from $X = BX \cup C$. Now suppose true for $n = k$. Let w be a string in $B^{k+1}C$. Then w is bw' for some $w' \in B^kC$ and some $b \in B$. Then $w' \in X$ because $B^kC \subseteq X$ by induction hypothesis. So bw' (which is w, after all) is in BX, and $BX \subseteq X$. (We were told this at the start: $X = BX \cup C$.) So $w \in X$ as desired.

Show $X \subseteq B^*C$

We classify members of X according to the number of strings from B that we can abstract from the front of them. Since $X = BX \cup C$, any string from X that has no initial segment in B must be a string from C. Any string from X that has $n + 1$ B-strings concatenated together as an initial segment can be tho'rt of as the result of sticking a B string on the front of a string with n B-strings ... etc. etc, so it is in BX. We can only do this finitely many times (n gets smaller each time) so eventually we are left with a string in C as before. The significance of B not containing the empty string is that it ensures that taking a B-string off the front results in a genuine reduction in length.

Question 8

Part (i)

$X_1 = bX_1|cX_2$ and $X_2 = (b|c)X_1|cX_2$

Joel Cartwright and Matt Cunnane and i came to the conclusion that X_1 contains all nonempty strings ending in a c, and X_2 contains the null string and all non-null strings ending in a c.

The informal way to do this is to accumulate information about what X_1 and X_2 contain. Initially X_1 contains nothing and X_2 contains the empty string. Then the two CX_c clauses tell us that X_1 and X_2 contain c^*. And so on ...

Part (ii)

$X_1 = (b|\epsilon)X_1|\epsilon$ and $X_2 = bc(X_1|\epsilon)$

We reckoned that $X_1 = b^*$ and $X_2 = bcb^*$

Question 9

The machine is the cube itself

Question 13

> Give regular grammars for the two following regular expressions over the alphabet $\Sigma = \{a, b\}$ and construct finite non-deterministic automata accepting the regular language denoted by them:
>
> 1. $ba|(a|bb)a*b$
> 2. $((a|b)(a|b))*|((a|b)(a|b)(a|b))*$

Question 13 (1): $ba|(a|bb)a*b$

An FNA constructed by the mechanical construction of Kleene's Theorem to recognise the language generated by the above regular expression has $Q = \{0, 1, 2, \ldots 12\}$, $i = 0$, $F = \{12\}$, and $\delta : (Q \times \Sigma) \rightarrow P(Q)$ as follows:

	State						
Input	0	1	2	3	4	5	6
ε	{0,4}	{}	{}	{12}	{5,7}	{}	{}
a	{}	{}	{a}	{}	{}	{6}	{}
b	{}	{2}	{}	{}	{}	{}	{}

	State					
Input	7	8	9	10	11	12
ε	{}	{}	{10}	{}	{12}	{}
a	{}	{}	{}	{}	{}	{}
b	{8}	{9}	{}	{11}	{}	{}

After removal of redundant ε-transitions, and merging of states where possible, this reduces to an FNA with $Q = \{0, 1, 2, 3\}$, $i = 0$, $F = \{3\}$ and $\delta : (Q \times \Sigma) \to P(Q)$ given by:

	State			
Input	0	1	2	3
a	{2}	{3}	{2}	{}
b	{1}	{2}	{3}	{}

Another answer: a nondeterministic automaton.
The FNA $M = (Q, \Sigma, \Delta, i, F)$ is given thus:

$$\begin{aligned} Q &= \{A, B, C, D\} \\ \Sigma &= \{a, b\} \\ i &= A \\ F &= \{D\} \end{aligned}$$

Δ, the transition relation of M is given by:

	a	b
A	C	D
B	D	C
C	C	D
D	E	E
E	E	E

And here is a grammar for it.
We define the NON-TERMINALS of G :
$N = \{\langle \texttt{start} \rangle, \langle xx \rangle\}$
the TERMINALS of G :

$$E = \{a, b\}$$

the `START` SYMBOL :
$i = \langle$ `start` $\rangle$
and the PRODUCTION of G : $\langle$ `start` $\rangle ::= ba$
$::= a\langle xx \rangle$
$::= bb\langle xx \rangle$
$\langle xx \rangle ::= a\langle xx \rangle$
$::= b$
Here is another CFG for $(ba|(a|bb)a^*b)$:

$$\begin{aligned} \Sigma &= \{a, b\} \\ N &= \langle str \rangle, \langle astarb \rangle \\ i &= \langle str \rangle \end{aligned}$$

The productions (all of the form $x \to wy$ and $x \to w$ where $w \in \Sigma^*$ is a string of terminals and $x, y \in N$ are non-terminals) are given below:

$$\begin{aligned}
\langle str \rangle &\to ba \\
\langle str \rangle &\to a\langle astarb \rangle \\
\langle str \rangle &\to bb\langle astarb \rangle \\
\langle astarb \rangle &\to a\langle astarb \rangle \\
\langle astarb \rangle &\to b
\end{aligned}$$

Question 13 (2): $((a|b)(a|b)) * |((a|b)(a|b)(a|b))*$

This language consists of all strings of a length divisible by two or three. The corresponding FDA $M = (Q, \Sigma, \Delta, i, F)$ is given thus:

$$\begin{aligned}
Q &= \{A, B, C, D, E, F\} \\
\Sigma &= \{a, b\} \\
i &= A \\
F &= \{A, C, F\}
\end{aligned}$$

Δ, the transition relation of M is given by:

	a	b
A	B	B
B	C	C
C	D	D
D	E	E
E	F	F
F	A	A

The grammar

Thinking of the ring of six states composing the equivalent FDA we define the NON-TERMINALS of G :

$$N = \{\langle S0 \rangle, \langle S2 \rangle, \langle S3 \rangle, \langle S4 \rangle\}$$

the TERMINALS of G :

$$E = \{a, b\}$$

the `START` SYMBOL : $i = \langle S0 \rangle$ and the PRODUCTION of G : $\langle S0 \rangle ::= aa\langle S2 \rangle$
$::= ab\langle S2 \rangle$
$::= ba\langle S2 \rangle$
$::= bb\langle S2 \rangle$
$::= e$
$\langle S2 \rangle ::= a\langle S3 \rangle$
$::= b\langle S3 \rangle$
$::= e$
$\langle S3 \rangle ::= a\langle S4 \rangle$
$::= b\langle S4 \rangle$
$::= e$
$\langle S4 \rangle ::= aa\langle S0 \rangle$
$::= ab\langle S0 \rangle$
$::= ba\langle S0 \rangle$

$::= bb\langle S0\rangle$
$::= e$
The CFG is...

$$\begin{aligned} \Sigma &= \{a, b\} \\ N &= \{\langle str\rangle, \langle twos\rangle, \langle threes\rangle\} \\ i &= \langle str\rangle \end{aligned}$$

The productions are...

$$\begin{aligned} \langle str\rangle &\rightarrow \langle twos\rangle \\ \langle str\rangle &\rightarrow \langle threes\rangle \\ \langle twos\rangle &\rightarrow aa\langle twos\rangle \\ \langle twos\rangle &\rightarrow ab\langle twos\rangle \\ \langle twos\rangle &\rightarrow ba\langle twos\rangle \\ \langle twos\rangle &\rightarrow bb\langle twos\rangle \\ \langle twos\rangle &\rightarrow \varepsilon \\ \langle threes\rangle &\rightarrow aaa\langle threes\rangle \\ \langle threes\rangle &\rightarrow aab\langle threes\rangle \\ \langle threes\rangle &\rightarrow aba\langle threes\rangle \\ \langle threes\rangle &\rightarrow abb\langle threes\rangle \\ \langle threes\rangle &\rightarrow baa\langle threes\rangle \\ \langle threes\rangle &\rightarrow bab\langle threes\rangle \\ \langle threes\rangle &\rightarrow bba\langle threes\rangle \\ \langle threes\rangle &\rightarrow bbb\langle threes\rangle \\ \langle threes\rangle &\rightarrow \varepsilon \end{aligned}$$

Q 14

For each of the following languages either show that the language is regular (for example by showing how it would be possible to construct a finite state machine to recognise it) or use the pumping lemma to show that it is not:

(i) the set of all words not in a given regular language L;

To make a FDA which does not accept the language L but accepts everything not in L but in Σ^* we make all the accepting states non-accepting and all the non-accepting states accepting.

Therefore, if an automaton $M = (Q, \Sigma, \delta, i, F)$ accepts the language $L = L(M)$ we can construct an automaton $M' = (Q, \Sigma, \delta, i, Q \setminus F)$ which accepts the language $L' = L(M') = \Sigma^* \setminus L(M)$.

(ii) all palindromes over the alphabet a, b, c (a palindrome is a word that is unchanged when reversed, for example, *abcbabcba*);

Consider the palindromic word $a^k ba^k$ which has length $l = 2k+1$. We can break this down into a word $w = u_1 v u_2$ thus:

$$\begin{aligned} u_1 &= a^{k-s} \\ v &= a^s \\ u_2 &= ba^k \end{aligned}$$

If we now consider the word $w = u_1 v^n u_2$ in the case $n = 0$ we get that:

$$\begin{aligned} w &= u_1 v^n u_2 \\ &= u_1 u_2 \\ &= a^{k-s} b a^k \end{aligned}$$

This cannot be palindromic since $s \geq 1$. Therefore it cannot be in L. This is in contradiction of the pumping lemma which holds for all regular languages. Therefore, L cannot be a regular language.

(iii) if L is a regular language, the language which consists of reversals of the words in L.

To every regular language there corresponds a regular expression. Reverse the regular expression corresponding to the given language L. The result corresponds to the language which consists of reversals of the words in L.

(iv) given regular languages L and M, the set of strings that contain within them first a substring that is part of language L, then a substring from M; arbitrary characters from the alphabet a, b, c are allowed before, between and after these strings;

Let l and m be regular expressions which generate the languages L and M. To generate the language requested one would use the regular expression $(a|b|c)^* l (a|b|c)^* m (a|b|c)^*$. Since we have found a regular expression to generate the language, the language must be regular by Kleene's Theorem.

(v) given regular languages L and M, the set of strings that contain within them some substring which is part of both L and M.

Since L and M are both regular languages $L \cap M$ is also regular, and corresponds to some regular expression r. We haven't been told what the alphabet is that we are working with, so—by way of illustration—assume it is $\{a, b\}$. Then the set of strings we are after corresponds to the regular expression $(a|b)^* r (a|b)^*$.

Question 16

> A "Muller C-element" is a device which receives two streams of binary digits ($x_0 x_1 x_2 \ldots$ and $y_0 y_1 y_2 \ldots$) and outputs a stream of binary digits $z_0 z_1 z_2 \ldots$ satisfying the relation.
>
> $$z_{n+1} = \begin{cases} x_{n+1} & \text{if } x_{n+1} = y_{n+1}, \\ z_n & \text{if } x_{n+1} \neq y_{n+1} \end{cases}$$
>
> *Design a Moore machine with this behaviour.*

The machine $M = (Q, i, \Sigma, F, \delta, \diamondsuit, \clubsuit)$ would be defined as follows:

$$\begin{aligned} Q &= \{A, B, C\} \\ i &= A \\ \Sigma &= \{\texttt{00, 01, 10, 11}\} \\ F &= \{\} \\ \diamondsuit &= \{\texttt{0, 1}\} \end{aligned}$$

The transition function works along the lines of: if you are in a state with output x and get a pair of $\neg x$s then change to the state without output $\neg x$ else stay in the current state. If we take into account the initial state having an undefined output we get a transition function δ given by:

	00	01	10	11
A	B	A	A	C
B	B	B	B	C
C	B	C	C	C

Each state in the set Q an output from the set $\diamondsuit$ has associated with it, given by the following function ♣:

State	Output
A	Undefined
B	0
C	1

Another answer:

Let $Q = \{0, 1, 2\}$, $\Sigma = \{0, 1\}$, $i = 0$. Let $\delta : (Q \times \Sigma^2) \to Q$ be given by:

	(x_n, y_n)			
State	(0,0)	(0,1)	(1,0)	(1,1)
0	1	0	0	2
1	1	1	1	2
2	1	2	2	2

Let $z = \{x, 0, 1\}$, and $Z : Q \to z$ be given by:

q	$Z(q)$
0	x
1	0
2	1

Then $(Q, \Sigma^2, z, \delta, Z, i)$ is a Moore machine which implements the Muller C-element.

Question 17

Give context-free grammars generating the following languages:

1. $\{a^p b^q c^r | p \neq q \text{ or } q \neq r\}$

2. $\{w \in \{a, b\}^* |\ w \text{ contains exactly twice as many } a\text{s as } b\text{s}\}$

First one: The CFG would be as below. A large number of non-terminals are needed because the problem breaks up into subproblems quite readily. There are those with $p \neq q$ and those with $q \neq r$. In the first case we can reduce to $p > q$ and $p < q$ and in the second $q < r$ and $q > r$.

$$\begin{aligned} \Sigma &= \{a, b, c\} \\ N &= \{\langle str\rangle, \langle ab\rangle, \langle c\rangle, \langle pgq\rangle, \langle plq\rangle\} \\ i &= \langle str\rangle \end{aligned}$$

The productions are given by the following (given in Backus-Naur Form for brevity).

$$
\begin{aligned}
\langle str\rangle &::= \langle ab\rangle\langle c\rangle|\langle a\rangle\langle bc\rangle \\
\langle ab\rangle &::= a\langle pgq\rangle|\langle plq\rangle b \\
\langle c\rangle &::= c\langle c\rangle|\varepsilon \\
\langle a\rangle &::= a\langle a\rangle|\varepsilon \\
\langle bc\rangle &::= b\langle qgr\rangle|\langle qlr\rangle c \\
\langle pgq\rangle &::= a\langle pgq\rangle|a\langle pgq\rangle b|\varepsilon \\
\langle plq\rangle &::= \langle plq\rangle b|a\langle plq\rangle b|\varepsilon \\
\langle qgr\rangle &::= b\langle qgr\rangle|b\langle qgr\rangle c|\varepsilon \\
\langle qlr\rangle &::= \langle qlr\rangle c|b\langle qlr\rangle c|\varepsilon
\end{aligned}
$$

Second one: The CFG would be defined as follows:

$$
\begin{aligned}
\Sigma &= \{a, b\} \\
N &= \{\langle str\rangle\} \\
i &= \langle str\rangle
\end{aligned}
$$

The productions would work as follows. On each application two as and one b are added to the string. This can be done with the string at any position relative to the new characters and with the new characters in any order.

$$
\begin{aligned}
\langle str\rangle &\rightarrow \varepsilon \\
\langle str\rangle &\rightarrow aab\langle str\rangle \\
\langle str\rangle &\rightarrow aba\langle str\rangle \\
\langle str\rangle &\rightarrow baa\langle str\rangle \\
\langle str\rangle &\rightarrow aa\langle str\rangle b \\
\langle str\rangle &\rightarrow ab\langle str\rangle a \\
\langle str\rangle &\rightarrow ba\langle str\rangle a \\
\langle str\rangle &\rightarrow a\langle str\rangle ab \\
\langle str\rangle &\rightarrow a\langle str\rangle ba \\
\langle str\rangle &\rightarrow b\langle str\rangle aa \\
\langle str\rangle &\rightarrow \langle str\rangle aab \\
\langle str\rangle &\rightarrow \langle str\rangle aba \\
\langle str\rangle &\rightarrow \langle str\rangle baa
\end{aligned}
$$

Question 18

> Let M be a finite deterministic automaton with n states. Prove that $L(M)$ is an infinite set if and only if it contains a string of length l with $n \leq l < 2n$

First consider the simpler of the two implications, that *if $L(M)$ contains a string of length l (where $n \leq l < 2n$) then L(M) must be infinite.*

Since the automaton has only n states, if $l \geq n$ the string must pass through some states more than once (since a length of n requires n transitions or $n + 1$ states). This requires that a cycle exists in the automaton. If a cycle exists then

one could pass through it an arbitrarily large number of times, thus ensuring that $L(M)$ is an infinite set. This proves the implication in the case where $l \neq n$.

The reverse implication is that *if $L(M)$ is infinite then the language must contain a string of length l (where $n \leq l < 2n$).*

If $L(M)$ is infinite then M must contain some cycles (since it does not have an infinite number of states). The length of any one cycle from beginning, around and back to beginning can be no longer than n states since this would be the whole of the automaton. Passing twice around any of these cycles would produce a string of length in the desired range.

⋆ **9** ⋆ *Use the pumping lemma to show that $\{a^k b^k | k \geq 0\}$ is not a regular language over $\Sigma = \{a, b\}$.*

Here is a mistaken answer.

> If the word w is decomposed as $w = u_1 v u_2$ thus:
>
> $$\begin{aligned} u_1 &= a^{r-s} \\ v &= a^s \\ u_2 &= a^{k-r} b^k \end{aligned}$$
>
> The pumping lemma states that for all $n \geq 0$, $u_1 v^n u_2 \in L$. Consider our example in the case $n = 0$:
>
> $$\begin{aligned} w &= u_1 v^0 u_2 \\ &= u_1 u_2 \\ &= a^{r-s} a^{k-r} b^k \\ &= a^{k-s} b^k \end{aligned}$$
>
> which would only be a member of the language if s were zero. The pumping lemma states that $s \geq 1$. Therefore, this language is not regular.

The mistake is as follows. Suppose there were a machine that recognises $\{a^k b^k | k \geq 0\}$ and that it has n states, and w is a word in $\{a^k b^k | k \geq 0\}$ of length at least n, then certainly there is a decomposition of w $w = u_1 v u_2$. The trouble is that we do not know if v consists entirely of a's, or entirely of b's, or of some a's followed by some b's. The author of this answer didn't consider the other two possibilities. I think this is because he didn't understand why the pumping lemma is true!

Another pumping lemma question

Let M be a finite state machine that accepts all strings in $\{a^k b a^k | k \geq 0\}$. We will show that it will accept some strings that aren't palindromes, and therefore doesn't recognise $\{a^k b a^k | k \geq 0\}$.

Pick n larger than the number of states of the alleged machine. Consider what happens when we give the machine $a^n b a^n$. The idea is that in reading this string the machine must have visited one of its states twice, and must have gone through a loop. (Think of this as an application of the pigeonhole principle). That is to say, this word $a^n b a^n$ has a substring within it which corresponds to the machine's passage through the loop. Call this string w. In fact, since we have force-fed the machine n a's, and it has fewer than n states, we can take it that w consists entirely of a's. This idea is that any string which is like $a^n b a^n$ but differs from it in having the substring w within it replaced by any number of consecutive copies of w will still be accepted by the machine. The poor machine will go thru' the loop umpteen

times of course, but it has no memory of such events (it "knows" only the state it is in, not how often it has visited that state). Thus we can compel the machine to accept a string $a^k b a^n$ where $k \neq n$, demonstrating—as desired—that is doesn't recognise $\{a^k b^k | k \geq 0\}$.

$\star$ **4** $\star$ *Construct a finite deterministic automaton accepting the same language as the following finite non-deterministic automaton over* $\Sigma = \{0, 1\}$*: (Not reproduced here.)*

The FDA $M = (Q, i, \Sigma, F, \delta)$ would be defined as follows:

$$\begin{aligned} Q &= \{\phi, \{p\}, \{q\}, \{r\}, \{s\}, \{q,r\}, \{q,s\}, \{r,s\}, \{p,q,r\}, \{q,r,s\}\} \\ i &= \{p\} \\ \Sigma &= \{0, 1\} \\ F &= \{\{s\}, \{q,s\}, \{r,s\}, \{q,r,s\}\} \end{aligned}$$

The transition function δ would be defined thus:

	0	1
ϕ	ϕ	ϕ
$\{p\}$	$\{q,s\}$	$\{q\}$
$\{q\}$	$\{r\}$	$\{q,r\}$
$\{r\}$	$\{s\}$	$\{p\}$
$\{s\}$	ϕ	$\{p\}$
$\{q,r\}$	$\{r,s\}$	$\{p,q,r\}$
$\{q,s\}$	$\{r\}$	$\{p,q,r\}$
$\{r,s\}$	$\{s\}$	$\{p\}$
$\{p,q,r\}$	$\{q,r,s\}$	$\{p,q,r\}$
$\{q,r,s\}$	$\{r,s\}$	$\{p,q,r\}$

Odd number of 0s and 1s

This one is a real bugger. Here follows some edited highlights of the attempts of four Churchillians (Mssrs Tonge, Ewald, Scott and Walker) to crack it. One thing everybody agrees on is that the automaton accepts strings containing an odd number of 0s and an odd number of 1s.

One Churchillian says:

> A regular expression for the language it accepts can be constructed as follows. Let x be the regular expression 00|11|0101|1010. Then the required regular expression is $x^*((0x^*1)|(1x^*0))x^*$, since recognising x^* does not change the state.
>
> At any point $(00|11)^*$ and $(0101|1010)^*$ have the effect of sitting still. We can simplify this to $(00|11|0101|1010)^*$.

Another chap says:

> To get around clockwise from p to s we need to sit still at p, get a 0, sit still at q, get a 1, sit still at s. Alternatively, we can go anticlockwise and swap the 0s and 1s in the previous sentence and mention r rather than q.
>
> This becomes:

$$\begin{aligned} s &= 00|11|0101|1010 \\ r &= s^*(0s^*1|1s^*0)s^* \\ &= (00|11|0101|1010)^*(0(00|11|0101|1010)^*1| \\ &\quad 1(00|11|0101|1010)^*0)(00|11|0101|1010)^* \end{aligned}$$

Karl Ewald says:

> The shortest ones I can come up with at first glance are
>
> both even: $((00|11)|(01|10)(00|11)*(01|10))*$
>
> others: $(00|11)*R$ both-even
>
> where R is (01|10) for both-odd
>
> (1|010) for odd number of 1's - even number of 0's (0|101) for odd number of 0's - even number of 1's
>
> I think they are right, whether they are the shortest possible I don't know.
>
> Karl

For that machine which accepts strings containing even numbers of ones and zeros -

$$\{00|11|(01|10)[(00)^*(11)^*]^*(01|10)\}^*$$

This seems to be the simplest and most intuitive solution, the others I have found are:

$$\{0(11)^*0|[(01(11)^*0)(0(11)^*0)^*(01(11)^*0)]\}^*$$

and the same where all zeros and ones are swapped.

James

From rjt26@thor.cam.ac.uk Thu May 1 22:19:06 1997
Is this a correct answer for the DFA example you gave in the supervision?

$$(00|11|(01|10)(00|11)^*(01|10))^*$$

Rob Walker says:

> Here is what I think the regular expression for the FDA that accepts strings with an odd number of 0's and 1's in them:
>
> The (reformatted) answer is:

```
   0(00)*1
|
  (1  | 0(00)*01)
  (11 | 10(00)*01)*
  (0  | 10(00)*1)
|
  (0(00)*1 | (1  | 0(00)*01)
             (11 | 10(00)*01)*
             (0  | 10(00)*1) )
  (1(00)*1 | (0  | 1(00)*01)
             (11 | 10(00)*01)*
             (0  | 10(00)*1) )+
```

> Examples of strings this generates:

```
01
0001
10
001101

01101101
000011110111
01010011000010
```

> etc ...
>
> The (somewhat messy) program I used to generate this is:

```
/* A messy C program which takes the
   FDA to recognise strings over {0,1}
   with a odd number of both 0's & 1's
   and works out the regular expression
   that represents it using Kleene's
   theorem.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>

/*-------------------------------------------------------*/
```

```
/* delta : states x lang --> states */

int delta[5][2] = { -1, -1,     /* ignore these */
                     2,  3,
                     1,  4,
                     4,  1,
                     3,  2 };

static int initial = 1;
static int accept[5] = {0, 0, 0, 0, 1};

/* use dynamic programming techniques to reduce amount
   of work done by a very large amount
*/
static char *dynamic[5][5][5];

/*-----------------------------------------------------------*
 * R --> a | bc*d
 *-----------------------------------------------------------*/

char *R(int i, int j, int k)
{
 /* have I worked this out before ? */
 if (dynamic[i][j][k]) return(dynamic[i][j][k]);

  if (k == 0) {
    char *answer = malloc(8);          /* longest = (0|1) */
    int c=0;

    if (delta[i][0] == j) c = 1;
    if (delta[i][1] == j) c += 2;

    /* so c = 0, 1, 2, 3 */
    switch (c) {
       case 0: *answer = '\0'; break;
       case 1: sprintf(answer,"0"); break;
       case 2: sprintf(answer,"1"); break;
       case 3: sprintf(answer,"(0|1)"); break;
       }

    /* store result for use later */
    dynamic[i][j][k] = answer;
    return(answer);
    }
 else {
   char *answer, *pt1, *pt2, *pt3, *pt4, *second;
   int c=0;

   /* now it gets messy - work out what the resulting
      string is ...
   */

   pt1 = R(i, j, k-1); if (*pt1) c |= 1;

   pt2 = R(i, k, k-1); if (*pt2) c |= 2;
   pt3 = R(k, k, k-1); if (*pt3) c |= 4;
   pt4 = R(k, j, k-1); if (*pt4) c |= 8;
```

```
   answer = malloc(strlen(pt1)+strlen(pt2)+strlen(pt3)
                              +strlen(pt4)+12);
   second = malloc(strlen(pt2)+strlen(pt3)+strlen(pt4)+12);
   *second = '\0';

   /* if no components then go home! (second == "") */
   if (!c) { dynamic[i][j][k] = second; return(second); }

   /* if either p2 OR p4 is blank then NO second bit! */
   if (!*pt2 || !*pt4)
      { dynamic[i][j][k] = pt1; return(pt1); }

   /* is there a first part to the second part? */
   if (*pt2) strcat(second, pt2);

   /* what about the starred bit? */
   if (*pt3) {
      /* are the last two parts the same --> use + instead of * */
      if (!strcmp(pt3, pt4)) {

        /* try to reduce number of brackets */
        if (*pt3 != '(') strcat(second, "(");
        strcat(second, pt3);
        if (*pt3 != '(') strcat(second, ")");

        strcat(second, "+");
        }
      else {
        if (*pt3 != '(') strcat(second, "(");
        strcat(second, pt3);
        if (*pt3 != '(') strcat(second, ")");
        strcat(second, "*");
        }
      }

   /* is there an end bit? */
   if (*pt4 && strcmp(pt3, pt4)) strcat(second, pt4);

   /* if no first bit ... */
   if (!*pt1) { dynamic[i][j][k] = second; return(second); }

   /* is no second bit */
   if (!*second) { dynamic[i][j][k] = pt1; return(pt1); }

   /* if 1st bit = 2nd part then just return one */
   if (!strcmp(pt1, second)) { dynamic[i][j][k] = pt1; return(pt1); }

   sprintf(answer, "(%s|(%s))", pt1, second);

   dynamic[i][j][k] = answer;
   return(answer);
 }
}

/*--------------------------------------------------------*/

int main()
{
```

```
 printf("answer is %s\n", R(1,4,4));
 return(1);
}

%>>--------------------------------------------------------------------
%>>--------------------------------------------------------------------

%Rob

%-----------------------------------------------------------------------------
%| Rob Walker                      | Pt 1B Computer Science Tripos          |
%| Churchill College               | Telephone 0223 69509                   |
%| Cambridge. CB3 0DS              | e-mail:   rnw1000@cus.cam.ac.uk        |
%-----------------------------------------------------------------------------
```

An answer from Lewis Brown, but i'm not sure what the questions were

Let $Q = \{p, q, r, s\}$, $\Sigma = \{0, 1\}$, $i = \{p\}$, and $F = \{S \mid S \in P(Q) \wedge s \in S\}$. Let $\delta : (P(Q) \times \Sigma) \rightarrow P(Q)$ be as follows:

	State									
Input	{}	{p}	{q}	{r}	{s}	{q,r}	{q,s}	{r,s}	{p,q,r}	{q,r,s}
0	{}	{q,s}	{r}	{s}	{}	{r,s}	{r}	{s}	{q,r,s}	{r,s}
1	{}	{q}	{q,r}	{p}	{p}	{p,q,r}	{p,q,r}	{p}	{p,q,r}	{p,q,r}

1.3 Model Answers to Tripos Questions

1989:6:12

Explain what is meant by
(a) a regular expression;
(b) a (deterministic) finite state machine.

(a) Regular expressions are used to describe the language of an automaton. They describe how the language $L \subseteq \Sigma^*$ is constructed using a standard set of operations for making new languages from old ones. These operations are called union, concatenation and Kleene closure.

Union of two regular expressions r and s gives the regular expression $r|s$ which denotes the language $L(R) \cup L(S)$.

Concatenation of two regular expression r and s gives the regular expression rs which denotes the language $L(R)L(S)$.

Kleene closure of a regular expression r gives the regular expression $r^* = r|rr|rrr|rrrr|rrrrr\ldots$ which denotes the language $L(R)^*$.

The language $L(r)$ denoted by the regular expression r is the set of strings $L \subseteq \Sigma*$ which can be constructed by application of the regular expression r.

(b) A finite state machine takes has a set of possible states Q which it can be in at any time. According to its inputs from a finite set Σ and a transition function δ it moves between these states. An initial state i must be specified as must the set of accepting states F. When the machine is in an accepting state it can be said to have accepted the string of symbols fed it up to that point. The set of all such strings is the language $L(M)$ of the machine.

A deterministic machine can only be in one state for any input sequence, whereas a non-deterministic machine can be any of a number of states for the same input sequence.

Another tripos question

Assuming Kleene's theorem (which states that the regular expressions and finite state machines are closely related), describe what is meant by a regular language by relating such languages to both regular expressions and to machines.

A regular language L is one for which $L = L(r)$ for some regular expression r over the same alphabet Σ as that of the language L. In the context of machines, L is regular if $L = L(M)$ for some Finite Deterministic Automaton M.

State the Pumping Lemma for regular languages

For every regular language L, there is a number $l \geq 1$ such that all strings $w \in L$ of length l or more can be expressed as $w = u_1 v u_2$ where u_1, v and u_2 satisfy

- $length(v) \geq 1$

- $length(u_1 v) \leq l$
- for all $n \geq 0$, $u_1 v^n u_2 \in L$.

ie: For long enough strings in the language, more strings in the alphabet can be got by repeating the middle bit.

1992:4:9

(a)

Let L be the set of words over the alphabet $\{a, b\}$ that end in b and do not contain the word aa. Describe, with justification, a finite deterministic automaton accepting L.

Which, if any, of the following regular expressions denotes L? Justify your answer in each case.

(i) $(ab|bb|ba)^*b$

(ii) $(b|ab)^*b$

(iii) $(b|ab)^+$ [6 marks]

(b)

L and F are languages over an alphabet Σ, and F is finite. Prove that L is regular iff $L \cup F$ is regular. You may use any well-known results provided you state them clearly. [8 marks]

Answer to (a):

Our FDA $M = (Q, \Sigma, \delta, i, F)$ is defined thus:

$$\begin{aligned} Q &= \{\texttt{initial}, \texttt{got a}, \texttt{error}, \texttt{ok}\} \\ \Sigma &= \{a, b\} \\ i &= \{\texttt{initial}\} \\ F &= \{\texttt{ok}\} \end{aligned}$$

The state transitions δ are defined thus:

	initial	got a	error	ok
ε	initial	got a	error	accept
a	got a	error	error	got a
b	accept	accept	error	accept

Which of the three expressions denotes L? (i) cannot be the answer because it allows consecutive as. (ii) cannot be the answer because it cannot deliver ab. (iii) is OK however. It is clear that everything in $(b|ab)^+$ ends in b and does not contain two consecutive as. Conversely we want to show that any string ending in b that does not contain two consecutive as is in $(b|ab)^+$. We prove this by induction on the relation of end-extension on finite strings. Let w be a word ending in b that does not contain two consecutive as which does not belong to $(b|ab)^n$ for any n, and let it be minimal with this property. w must be either uab or ub where u is another word ending in b that does not contain two consecutive as. Because w is minimal with this bad property we know that u belongs to $(b|ab)^n$ for some n. But then w belongs to $(b|ab)^{n+1}$ contradicting assumption on w.

Answer to (b)

One direction is easy: the union of two regular languages is regular, and every finite language is regular so if L is regular so is $L \cup F$.

For the other direction notice that

$$L = (L \cup F) \setminus (F \setminus L)$$

which is

$$(L \cup F) \cap \overline{(F \setminus L)}$$

which is

$$(L \cup F) \cap \overline{(F \cap \overline{L})}$$

Now $(L \cup F)$ is regular by assumption; $F \cap \overline{L}$ is regular because it is a subset of F, and so is finite, and $(L \cup F) \cap \overline{(F \cap \overline{L})}$ is regular because it's the intersection of two regular languages.

1993:5:12

Part 2:

Given that $L(t|sr) \subseteq L(r)$. We want to show

$L(s^*t) \subseteq L(t|sr)$.

We prove by induction on the number n of occurrences of 's' on the front of a word w that $w \in L(s^*t) \rightarrow w \in L(t|sr)$. If $n = 0$ then $w \in L(t)$ so $w \in L(t|sr)$. Suppose true for k occurrences of 's' at the beginning of w. (That is to say, if there are k occurrences of 's' at the beginning of w and $w \in L(s^*t)$ then $w \in L(t|sr)$). So let sw be a word in $L(s^*t)$ with $k+1$ occurrences of 's' at the beginning. By induction hypothesis we can infer that $w \in L(t|sr)$. Now we were told at the outset that $L(t|sr) \subseteq L(r)$ so $w \in L(r)$ whence $sw \in L(sr)$ and $L(sr) \subseteq L(t|sr)$ so $sw \in L(t|sr)$ as desired.

1994:3:3

$$(1|01^*0)(0^*|(1^+0)^*)^*$$

1996:2:8

> *Show that if L is a regular language then the set of strings in L of odd length is also a regular language. Is the same true of strings of even length? Justify your answer* *[8 marks]*

The set of strings of even length is a regular language and the set of strings of odd length likewise a regular language. (the machine to do it needs only two states!) The intersection of two regular languages is regular, so the set of L-strings of even length is a regular language if L is.

To be a bit more formal about it, make two copies of a machine that recognises L, and change the transition rules so that each rule like "If in state s a letter p is received go to state s'" is replaced by two rules: (i) "If in state $\langle s, 0\rangle$ a letter p is received go to state $\langle s'1\rangle$" and (ii) "If in state $\langle s, 1\rangle$ a letter p is received go to state $\langle s'0\rangle$".

> *If L is a regular language let L' be the set of strings in L that are palindromes. Is it possible that L' is regular? Will L' necessarily be regular? Explain your answer with suitable examples and proofs* *[6 marks]*

Well, it's certainly possible that L' should be regular, because if L is finite L' will be finite too, and therefore regular. L' needn't be regular because if $L = \Sigma^*$ then L' is the language of all palindromes which famously is not regular.

It is known that the language Pal of all palindromes is not a regular language. If possible find a regular language L such that L is a subset of pal, or if this is not possible explain why. Similarly either find a regular language L' so that Pal is a subset of L' or again explain why this cannot be done. [6 marks]

Finally any finite set of palindromes is regular, so the answer to the first part is 'yes'. For the second part *Pal* is clearly a subset of Σ^*, and Σ^* is regular.

1998:2:7

The problem can be solved if we have a way of telling whether or not two given machines recognise the same language. After all, we can get a machine from a grammar and we can get a machine from a regular expression, so the Janet-and-John question can be expressed in this form.

So, given M_1 and M_2, do they recognise the same language? We seek a bound on the length of the shortest string on which they disagree. We can build a machine that accepts things accepted by M_1 but not by M_2, and a machine that accepts strings accepted by M_2 but not by M_1. (These two machines will even have the same number of states, namely $|M_1| \times |M_2|$). There is a version of the pumpkin (sorry, pumping) lemma that tells us that if either of these new machines accepts any strings at all, it will accept at least one string of length at most $|M_1| \times |M_2|$, which gives us the bound we need.

1.3.1 Regular expressions and pattern matching

```
(*  R E G U L A R  E X P R E S S I O N S  &  P A T T E R N  M A T C H I N G
 *
 *                                          by Inge Norum (in206) May '98
 *
 *  String interface functions are provided in 'RegExpStringInterface.ml'
 *)


datatype 'a RegExp = CONCAT  of ('a RegExp * 'a RegExp)
                   | UNION   of ('a RegExp * 'a RegExp)
                   | STAR of 'a RegExp
                   | VAL  of 'a
                   | NULL
                   | EMPTY;

(********** Sequence/Lazy list functions: **********)

datatype 'a seq = Cons of 'a * (unit -> 'a seq) | Nil;

fun head (Cons(x,_)) = x;

fun tail (Cons(_,f)) = f()  (* force evaluation *)
  | tail Nil = Nil;

fun Append (Nil,B) = B
  | Append (A,B) = Cons(head A, fn()=>Append(tail A, B));

fun Map f A = case A of Nil       => Nil
                      | Cons(x,t) => Append(f(x), ((Map f) (t())) );
(* not analogous to the list map function, uses append, not cons *)
```

```
(*********************************
  The pattern matching function.
  Returns a sequence of solutions, in the form of input 'left over'.
  => no input left over is a match  (but 'Nil' is not)
*)

fun REmatch (VAL(a),[]   ) = Nil
  | REmatch (VAL(a),x::xs) = if (a=x) then Cons(xs, fn()=>Nil)
                                      else Nil
  | REmatch (CONCAT(A,B),l) =
    Map (fn (inputleft_A) => REmatch(B, inputleft_A)) (REmatch(A, l) )

  | REmatch (UNION(A,B), l) = Append( REmatch(A,l), REmatch(B,l) )

  | REmatch (STAR(A), l) =
(*   = REmatch(UNION(NULL,CONCAT(A,STAR(A))), l )  can cause an infinite loop *)
    let
        fun StarFun(Nil) = Nil
          | StarFun(Match_A) =
          if ((head Match_A)=l) (* A has evalueted to NULL, stop recursion *)
             then StarFun(tail(Match_A))
             else Append( REmatch(STAR(A),head(Match_A))
                        , StarFun(tail(Match_A)) )
    in
        Cons( l , fn() => StarFun(REmatch(A,l)) )
    end

  | REmatch (NULL, l) = Cons( l, fn()=>Nil )

  | REmatch (EMPTY, l) = Nil
  ;

fun curry f a b = f(a,b);

val REmatchCurry = curry REmatch;

(* MatchFinder converts a sequence (from the function above)
   into a sequence of the solutions (i.e. an empty list).
   CountMatches can then be used to count these.             *)

fun MatchFinder (Cons([],f)) = (* no input left => successful *)
                               Cons(true, fn()=>MatchFinder(f()) )
  | MatchFinder (Nil)        = Nil (* no (more) matches *)
  | MatchFinder (S) = MatchFinder (tail(S));  (* try next *)


(* A curried 'interface' function for REmatch, using MatchFinder. *)

fun REmatches (R) input = MatchFinder( REmatch(R, input) );


(* Counts the number of solutions/matches in a bool solution sequence
   generated by MatchFinder/RegExpMatch.
   e.g. the number of ways "11" matches ( 1* )* is 2.
   i.e. CountMatches (REmatches ( STAR(STAR(VAL(1))) ) [1,1] ) = 2
*)

fun CountMatches (Cons(b,f)) = if b then 1+CountMatches(f())
```

```
                                    else   CountMatches(f())
  | CountMatches (Nil) = 0
  | CountMatches (l) = CountMatches(tail(l));

(*      S T R I N G   I N T E R F A C E   F U N C T I O N S
 *
 *    for the RegExp pattern matching functions (in RegExp.ml)
 *
 *                                        by Inge Norum (in206) May '98
 *)

(* Returns a list of the n first elements in list l *)
fun nFirst (l,n) = if n=0 then [] else hd(l)::nFirst(tl(l),n-1);

(* Returns list l with the n first elements removed *)
fun nLess  (l,n) = if n=0 then l else nLess(tl(l),n-1);

(* Returns the nth elemts of list l *)
fun nth ((x::xs),n) = if n=1 then x else nth(xs, n-1);

exception ErrorBrackets;
exception ErrorSyntax;

fun ScanCorrespondingRBracket (l, bracketdepth) =
    let fun ScanCRB([],_,_) = raise ErrorBrackets
          | ScanCRB(x::xs, pos, b) =
             if x="(" then ScanCRB(xs,pos+1,b+1)
        else if x=")" then if b=1 then pos else ScanCRB(xs,pos+1,b-1)
                      else ScanCRB(xs,pos+1,b)
    in  ScanCRB(l,0,bracketdepth)
    end;

fun ScanOuterUnion (l) =
    let fun ScanOU([],_,_) = ~1      (* no outer | found *)
          | ScanOU(x::xs, pos, b) =  (* b is the bracketdepth *)
                if x="(" then ScanOU(xs,pos+1,b+1)
          else if x=")" then ScanOU(xs,pos+1,b-1)
          else if x="|" andalso b=0 then pos
                        else ScanOU(xs,pos+1,b)
    in  ScanOU(l,0,0)
    end;


(*************************************************************)
(* Main Function: Generates a string RegExp from string input   *)
(*                                                           *)
(* Order of precedence:  (-)  >  -*  >  --  >  -|-           *)
(*                                                           *)
(* Note: "()" is converted to a null character               *)
(*************************************************************)

fun L2RE ([],_) = NULL
  | L2RE (x::xs,l) =

    if ScanOuterUnion(x::xs)>0 then
       let
           val n = ScanOuterUnion(x::xs)
       in
```

```
           UNION(L2RE(nFirst(x::xs,n),n), L2RE(nLess(xs,n),l-n-1))
       end

    else if x="(" then
      let val n = ScanCorrespondingRBracket(xs,1)
      in  if l=n+2 then L2RE(nFirst(xs,n),n)
          else if (nth(xs,n+2))="*" then
                  if l=n+3 then STAR(L2RE(nFirst(xs,n),n))
                  else CONCAT(STAR(L2RE(nFirst(xs,n),n))
                             ,L2RE(nLess(xs,n+2),l-n-3) )
          else CONCAT(L2RE(nFirst(xs,n),n), L2RE(nLess(xs,n+1),l-n-2) )
      end

    else if x=")" then raise ErrorBrackets
    else if x="*" then raise ErrorSyntax
    else if x="|" then raise ErrorSyntax

    else if l=1 then VAL(x)

    else if (hd(xs))="*" then if l=2 then STAR(L2RE([x],1))
         else CONCAT(STAR(L2RE([x],1)), L2RE(tl(xs),l-2))

    else CONCAT(L2RE([x],1), L2RE(xs,l-1))
    ;

(* SRE2IRE converts a string RegExp to an int RegExp one.
   Characters are changed to ints using ord(), relative to "0". *)

fun SRE2IRE (VAL(c)) = VAL(ord(c) - ord("0"))
  | SRE2IRE (NULL)   = NULL
  | SRE2IRE (EMPTY)  = EMPTY
  | SRE2IRE (STAR(A)) = STAR(SRE2IRE (A))
  | SRE2IRE (CONCAT(A,B)) = CONCAT(SRE2IRE (A), SRE2IRE (B))
  | SRE2IRE (UNION(A,B))  =  UNION(SRE2IRE (A), SRE2IRE (B));

(* Removes spaces from an expanded string (string list) *)

fun RemoveSpaces [] = []
  | RemoveSpaces (x::xs) = if x=" " then      RemoveSpaces(xs)
                                    else x :: RemoveSpaces(xs);

(* Calls L2RE to generate a string RegExp from a string.
   All spaces in the input are removed!                  *)

fun S2SRE (S) = let val ProperString = RemoveSpaces (explode S)
                in  L2RE (ProperString, length(ProperString))
                end;

(* S2IRE generates an integer RegExp, using S2SRE and SRE2IRE.
   (Int RegExps use less memory and are faster than string RegExps.)
*)
fun S2IRE (S) = SRE2IRE (S2SRE(S));


(* S2Sinput converts a string to a list of input characters for use
   in the matching function (REmatch).  i.e. "ab" -> ["a","b"]     *)

fun S2Sinput (S) = explode (S);
```

```
(* S2Iinput converts a string to an integer input list
   i.e. "10120" -> [1,0,1,2,0]
   Alphabetical characters will also be uniquely mapped to an integer
   equivalent. (Using integer internal structures is much faster than
   using the string ones in the pattern matching function.)          *)

fun S2Iinput (S) = (map (fn(c)=>ord(c)-ord("0")) (explode S));


(* SMatch and IMatch test string input for matching in
   string and int RegExps respectively.
   They return a sequence of bools, corresponing to each match. *)

fun SMatch R StringInput = (REmatches (R)) (S2Sinput(StringInput));

fun IMatch R StringInput = REmatches (R) (S2Iinput(StringInput));


(* BoolAnswer can be used to turn the output of the preceding two functions
   into a single true or false.  i.e. accepted/not accepted by the RegExp *)

fun BoolAnswer (Cons(b,f)) = if b then true else BoolAnswer(f())
  | BoolAnswer (Nil) = false;

(* A 'total' string matching function, give it two strings and
   it returns a bool. (it is curried)                          *)

fun match StringRegExp StringInput =
    BoolAnswer(SMatch (S2SRE(StringRegExp)) (StringInput) );


(* Same as match, but returns the total number of matches.
   Uses integer RegExps for lower memory requirement.      *)

fun matchCount StringRegExp StringInput =
    CountMatches (IMatch (S2IRE(StringRegExp)) (StringInput) );


Dear Dr. Forster,

I have made some Language code.  (Using sequences here as well!)  It isn't
working reliable yet, but just thought I would email it anyway.

I didn't use lexicographic ordering for the Language, because then expressions
like (a|b)* would just evaluate to "", "a", "aa", "aaa", ...  I used a mixture
of length and alphabetic ordering:I.e. "", "a", "b", "ab", "ba", ...

The function locks into an infinite loop if the regular expression
contains too many stars.  I guess it is inserting infinitely many null
strings...  I will try to fix that later.

Inge

RegExpLanguage.ml:

(*  L A N G U A G E   G E N E R A T O R
 *                                        by Inge Norum (in206) May '98
```

```
 *
 *  (Uses RegExp/sequence datatypes/functions defined in RegExp.ml)
 *)



fun MapCons f A = case A of Nil       => Nil
                          | Cons(x,t) => Cons((f x), fn()=>((MapCons f) (t())))
);
(* analogous to the list map function *)


fun Merge (Cons((i:int,a:string),fa), Cons((j,b),fb)) =
    if i<j orelse (i=j andalso a<b)
       then Cons((i,a), fn()=>Merge(fa(),Cons((j,b),fb)) )
    else if j<i orelse b<a
       then Cons((j,b), fn()=>Merge(Cons((i,a),fa),fb()) )
       else Cons((j,a), fn()=>Merge(fa(),fb()) )
  | Merge (A,Nil) = A
  | Merge (Nil,B) = B;


fun ConCat ((i:int,a:string), (j,b)) = (i+j, a^b);



(* Generates a sequence of all the strings accepted by a regular expression,
   in 'length order' (& lexicographic order)                                  *)

fun Language (VAL(x:string)) = Cons((1,x), fn()=>Nil)

  | Language (UNION(R,S)) = Merge(Language(R),Language(S))

  | Language (CONCAT(R,S)) =
    let
        val Language_S = Language(S)
        fun ConCatFun (Nil) = Nil
          | ConCatFun (Cons(head_R, fr)) =
            Merge(
            (MapCons (fn (head_S) => ConCat(head_R, head_S))  Language_S )
            , (ConCatFun(fr())) )
    in
        ConCatFun(Language(R))
    end

  | Language (STAR(R)) = Cons( (0,""), fn()=>Language(CONCAT(R,STAR(R))) )

  | Language (NULL)    = Cons( (0,""), fn()=>Nil)

  | Language (EMPTY)   = Nil
  ;


(* The n first words in the language: *)

fun LanguageList (Cons((i,s),f), n) = if n=0 then []
                                      else s::LanguageList(f(), n-1)
  | LanguageList (Nil,_) = ["End of language."];
```

```
(* All the words with length not longer than l: *)

fun LanguageNotLongerThan (Cons((i,s),f), l:int) =
    if i<=l then s::LanguageNotLongerThan(f(), l)
            else []
  | LanguageNotLongerThan (Nil,_) = ["End of language."];
```

Chapter 2

Foundations of Functional Programming

Some suitable tripos questions.

1991:5:9, 1993:12:9, 1994:5:10, 1994:10:12, 1995:5:10, 1995:12:12, 1996:5:9, 1996:12:11, 1997:5:11, 1998:13:10

A Question from John Harrison's notes

```
Moscow ML version 1.40 (1 July 1996)
Enter ‘quit();’ to quit.
- fun itlist f [] b = b
    | itlist f (h::t) b = f h (itlist f t b);
> val itlist = fn : (’a -> (’b -> ’b)) -> ’a list -> ’b -> ’b
- fun map f [] = []
    | map f (h::t) = (f h)::(map f t);
> val map = fn : (’a -> ’b) -> ’a list -> ’b list
- fun C f x y = f y x;
> val C = fn : (’a -> (’b -> ’c)) -> ’b -> ’a -> ’c
- local fun mem x [] = false
          | mem x (h::t) = x = h orelse mem x t;
        fun insert x l = if mem x l then l else x::l
     in fun union l1 l2 = itlist insert l1 l2
    end;
> val union = fn : ’’a list -> ’’a list -> ’’a list
- fn f => fn l1 => fn l2 => itlist (union o C map l2 o f) l1 [];
> val it = fn : (’a -> (’b -> ’’c)) -> (’a list -> (’b list -> ’’c list))
- it (fn a => fn b => (a,b)) [1,2,3] [4,5,6];
> val it =
    [(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]
    : (int * int) list
```

2.1 Answers to Tripos questions

1991:5:9

We declare some λ-terms as follows.

$\underline{\text{true}} = \lambda xy.x$; $\underline{\text{false}} = \lambda xy.y$; $\underline{0} = \lambda x.x$; $\underline{\text{succ}} = \lambda zf.f\ \underline{\text{false}}\ z$

- a We want $\underline{\text{succ}}\ (\underline{\text{succ}}\ \underline{0})$

First we compute succ $\underline{0} = \lambda z f.f$ false $z \;\; \lambda x.x$
$= \lambda f.f$ false $(\lambda x.x)$. Do succ again
succ $(\lambda f.f$ false $(\lambda x.x))$
$= \lambda y g.g$ false y $(\lambda f.f$ false $(\lambda x.x))$
$= \lambda g.g$ false $(\lambda f.f$ false $(\lambda x.x))$

- b Evaluate at false
- c Evaluate at true
- d Distinct normal forms

1993:12:9

```
- fun N f x = x;;
val N = fn : 'a -> 'b -> 'b

-  fun P a k f x = f a (k f x);;
val P = fn
  : 'a -> (('a -> 'b -> 'c) -> 'd -> 'b) -> ('a -> 'b -> 'c) -> 'd -> 'c

- fun Q k l f x = k f (l f x);;
val Q = fn : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c


-  fun W a k = Q k (P a k);;
val W = fn
  : 'a -> (('a -> 'b -> 'c) -> 'c -> 'b) -> ('a -> 'b -> 'c) -> 'c -> 'b

-  fun R k = k W N;;
val R = fn
  : ((('a -> (('a -> 'b -> 'c) -> 'c -> 'b) -> ('a -> 'b -> 'c) -> 'c -> 'b)
     -> ('d -> 'e -> 'e) -> 'f)
    -> 'f
```

Suppose further that `K` and `L` have `ML` definitions of the form:

```
val K = P a1 (P a2 (P a3 N ...))

val K = P b1 (P b2 (P b3 N ...))
```

1. *State the* `ML` *types of* `N` *and* `P`.

   `N` has type $\alpha \to (\beta \to \beta)$. `P` seems to have the type:

   $\alpha \to (((\alpha \to (\beta \to \gamma)) \to (\delta \to \beta)) \to ((\alpha \to (\beta \to \gamma)) \to (\delta \to \gamma)))$. God knows why.

2. *What does the expression* `K f x` *evaluate to?*

   Let us start by noting that `K` of the form `P a1 T` for some term `T`. Then

   `K f x` becomes

   `P a1 T f x` by expanding `K`. Then expand `P` to get

   `f a1 (T f x)`

   But now we have a task just like the one we started with, since `T` is an expression of the same form as `K`. We repeat this step until we have `T` that is in fact `N`. But `N f x` is just `x`, so the result is going to be `K` with all the `P`s replaced by `f`s and the final `N` replaced by `x`.

3. *What does the expression* `Q K L f x` *evaluate to?*

   `Q K L f x` First expand `Q` to get

   `K f (L f x)` Next use the fact that `K` is of the form `P a1 T` as before.

   `P a1 T f (L f x)` then expand `P` getting

   `f a1 (T f (L f x))`

But by now the embedded expression `T f (L f x)` is clearly of the same form as the term `K f (L f x)` we started with, so we can keep repeating this until we get down to a `T` that will in fact be `N`. Then `f a1 (N f (L f x))` will clearly β-reduce to `f a1 (L f x)` which means that the result is going to be a concatenation of the two lists!

4. *What does the expression* `R K f x` *evaluate to?* Given that the earlier ISWIM code in this question relates to an ISWIM implementation of lists, it's an obvious guess that `R` means `R`everse. Let's see.

`R k f x`	expand `R k`
`k W N f x`	expand `K` into `P a1 k'`
`P a1 k' W N f x`	expand `P a1 k' W N` using definition of `P`
`W a1 (k' W N) f x`	expand `W`
`Q (k' W N) (P a1 N) f x`	`W` now has 4 arguments, so expand it
`k' W N f (P a1 N f x)`	

So by unpeeling the top level of the structure of `k`, into `P a1 k'`, we have turned

`k W N f x` into

`k' W N f (P a1 N f x)`

These two look suspiciously alike. We match `k` to `k'`, `W` and `N` to themselves, and `P a1 N f x` to `x`. This tells us what will happen if we repeat what we have just done, this time on `k'` instead of on `k`. `R` is indeed `R`everse!

1994:5:10

'=' between λ-terms is not (syntactic) equality, but *convertibility* and it is defined by the following recursion

- $(\lambda x.M)N \ = \ M[x := N]$ (β-conversion)
- $M = M$
- $M = N \rightarrow N = M$
- $M = N \wedge N = L \rightarrow M = L$
- $M = N \rightarrow MZ = NZ$
- $M = N \rightarrow ZM = ZN$
- $M = N \rightarrow \lambda x.M = \lambda x.N$

Set **true** $= \lambda xy.x$ and **false** $= \lambda xy.y$. Then we can define the desired terms as follows.

cons Define **cons** $a\ l = \lambda fx.fa(lfx)$

null I am assuming that the null list is $\lambda fx.x$, which is **false** (i can't think what else the given definition of list could intend it to be). So **null** l must be $\lambda l.(lt)$**true** for some term t. Consideration of the null case doesn't tell us what t is to be.

Check this:

$((\lambda l.lt)$**true**) **false**

β-reduces to (**false** t)**true**

which β-reduces to **true** as desired.

On the other hand a non-null list is $\lambda fx.faM$ for some term M containing an occurrence of 'x'. This will tell us what t has to be. So let's apply

$\lambda l.l\ t$ **true** to $\lambda fx.faM$.

$\lambda l.l\ t$ **true** $\lambda fx.faM$.

$(\lambda fx.faM)\ t$ **true**.

$(\lambda x.taM)$ **true**.

This has got to β-reduce to **false**, and we can achieve this only by tweaking t. Let's take t to be $K(K($**false**$))$. Then

$(\lambda x.taM)$ **true** is

$(\lambda x.K(K($**false**$))aM)$ **true**.

$(\lambda x.$**false**) **true**.

false.

So **null** must be $\lambda l.l\ (K(K$**false**$))$ **true**.

append Define **append** $l_1\ l_2 = \lambda fx.(l_1 f)(l_2 fx)$

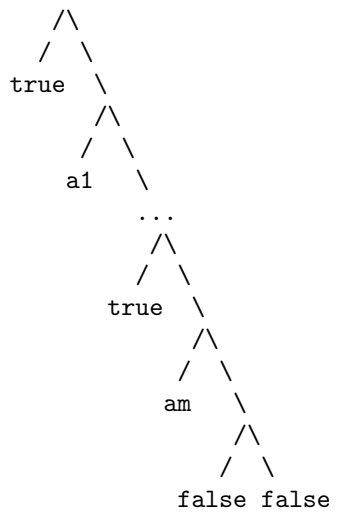
hd can be $\lambda l.l$ **true** t for any λ-term t.

tl (Matt Grimwade says:)

The clue is in the question: "assume λ-encodings of ordered pairs". So first use:

$$\texttt{mk\_pairs} = \lambda l.l(\lambda ax.\texttt{pair}\ true\ (\texttt{pair}\ ax))(\texttt{pair}\ \textbf{false}\ \textbf{false})$$

to turn the list into a lot of internesting pairs, flagging the end so we'll know when we get there:



get rid of the unwanted head using `snd`; and convert back with:

$$\texttt{mk\_list} = Y(\lambda gpfx.\texttt{if}(\texttt{fst}\ p)(f(\texttt{fst}(\texttt{snd}\ p))(g(\texttt{snd}(\texttt{snd}\ p))fx)x).$$

In total we have:

$$\texttt{tl} = \lambda lfx.\texttt{mk\_list}(\texttt{snd}(\texttt{snd}(\texttt{mk\_pairs}\ l)))fx.$$

1994:10:12

Recall that `f o g` is the function that sends x to $f(g(x))$. Consider the `ML` definitions: `fun I` $x = x$; `fun pair` $(f,g)(x,y) = ((fx),(gy))$; `fun pup` $(f,g)z = ((fz),(gz))$; `fun fst`$(x,y) = x$; `fun snd`$(x,y) = y$;

Describe the effect of the following functions:

- `pair`$(\texttt{I},\texttt{I})$ is the identity relation of type $\alpha \times \beta \to \alpha \times \beta$.
- `pair`$((f1 \circ f2),(g1 \circ g2))$ sends (x,y) to $(f1(f2(x)),g1(g2(y)))$
- `pup`(`fst`, `snd`) is the identity relation of type $\alpha \times \beta \to \alpha \times \beta$.
- `pup(f o fst, g o snd)` sends x to $((f(\texttt{fst}(x)),g(\texttt{snd}(x)))$. That is, it behaves like `pair` (f,g) x.

Infinite lists can be represented in a functional language by triples. The triple (a,h,t) represents the infinite list whose nth element is $h(t^n(a))$ (for $n \geq 0$).

(a) Give a representation for the infinite list $n, n+1, n+2, \ldots$

This is $(n, \texttt{I}, succ)$

(b) Code in `ML` a map functional for this representation; given a function f and the infinite list $x_0, x_1, \ldots x_n \ldots$, it should yield a representation of $f(x_0), f(x_1), \ldots f(x_n) \ldots$,

`map` f $(a,h,t) =: (a,(f \circ h),t)$;

(c) Code in `ML` a *zip* function

`zip` (a,h,t) $(a',h',t') =: ((a,a'),$ `pair`$(h,h'),$ `pair`$(t,t'))$;

(d) Code in `ML` an *interleave* function

[*HOLE !!*]

- (e) Discuss

1995:5:10

Let

$A = \lambda xy.y(xxy)$

$\Theta = AA$

$\texttt{suc} = \lambda nfx.f(nfx)$

$\texttt{true} = \lambda xy.x$

$\texttt{false} = \lambda xy.y$

The first thing to do is to demonstrate that Θ is a fix-point combinator. Let us β-reduce $\Theta\ f$. This is $A\ A\ f$, or
$((\lambda xy.y(xxy))\ A)\ f$
The first occurrence of A (the one i have written out in full) has two variables at the front, 'x' and 'y'. We replace 'x' by A and 'y' by 'f', and lop off the '$\lambda xy.$' getting $f(\Theta f)$.
This will save a lot of trouble later on.

- Θ `suc` will β-reduce to `suc`$(\Theta$ `suc`$)$, which is $\lambda fx.f((\Theta$ `suc`$)\ f\ x)$ which is in head normal form, tho' clearly not in normal form.
- $\Theta\ (\lambda x.xx)$ β-reduces to $(\Theta(\lambda x.xx))(\Theta(\lambda x.xx))$
  This is going to go on getting bigger and will have no normal form.
- $\Theta($`suc` $n)$ β-reduces (once one has relettered 'n' for 'x') to (`suc` $n)(\Theta$ (`suc` $n))$. Notice that `suc` of anything has a head normal form. But this thing is not going to have a normal form.
- Θ `true` β-reduces to
  `true`$(\Theta$ `true`$)$. But `true` x is always $\lambda y.x$. So this is $\lambda y.(\Theta$ `true`$)$ which has our original formula as a subformula, so this will go on for ever. It doesn't even have a head normal form, beco's it never returns anything except itself.
- Θ `false` β-reduces to
  `false`$(\Theta$ `false`$)$. `false` of anything is I.
- $\Theta(\lambda x.fxx)$ β-reduces to $(\lambda x.fxx)(\Theta(\lambda x.fxx))$ and one more β-reduction will give
  $((f(\Theta(\lambda x.fxx))))(\Theta(\lambda x.fxx))$
  which is in head normal form.

If M has no hnf then $M[N/x]$ has no hnf, for any N. Use this fact to prove the following:

If M has no hnf then MN has no hnf for any N.

1995:12:12

The type of `curry` is $((\alpha \times \beta) \to \gamma)) \to (\alpha \to (\beta \to \gamma))$ and the type of `uncurry` is $(\alpha \to (\beta \to \gamma)) \to ((\alpha \times \beta) \to \gamma))$.

`curry(fn(x,y) => x)` is $\lambda xy.x$.

`uncurry o curry` is the identity function of type $((\alpha \times \beta) \to \gamma)) \to ((\alpha \times \beta) \to \gamma))$

`curry I` is $\lambda xy.\texttt{pair}(x, y)$. To deduce this we have to specialise `I` to a thing of type $(\alpha \times \beta) \to (\alpha \times \beta)$, and `curry` of a thing of this type must be of type $\alpha \to (\beta \to (\alpha \times \beta))$.

To determine what `uncurry I` evaluates to we must note that `uncurry` expects an argument of type $\alpha \to (\beta \to \gamma)$. So we must specialise `I` to identity of type $(\beta \to \gamma) \to (\beta \to \gamma)$. Applying `uncurry` to this will give a result of type $((\beta \to \gamma) \times \beta) \to \gamma$ and this must be

$\lambda x.((\texttt{fst}(x))\texttt{snd}(x))$.

(a) `fun` n `=>` $n * 2$;

(b) If `g` codes the list in question we want `f o g`.

(c) `drop` $f\ i\ n = f\ (n + i)$;

(d) `interleave` $f\ g\ n =$ `if even` n `then` f $(n$ `div` $2)$ `else` g $(n$ `div` $2)$;

(e)
```
fun ifilter(f,p,x,0,0) = if p(f(x)) then f(x) else
                            ifilter(f,p,x+1,0,0)
  | ifilter(f,p,x,n,l) = if n=l then f(x-1) else
                         if p(f(x)) then ifilter(f,p,x+1,n+1,l)
                                    else ifilter(f,p,x+1,n,l);;
fun filter f p x = ifilter(f,p,0,0,x);;
```

(Thanks to David Bradshaw)

1996:5:9

Consider binary trees that are either empty (written '`leaf`') or have the form `Br` $x\ t_1\ t_2$ where t_1 and t_2 are themselves binary trees. Give an encoding of binary trees in the λ-calculus, including functions `isleaf`, `label`, `left` and **right** satisfying

`isleaf leaf` $\rightarrow$ **false**

`isleaf`(`Br` $x\ t_1\ t_2$) $\rightarrow$ **true**

`label`(`Br` $x\ t_1\ t_2$) $\rightarrow x$

`left`(`Br` $x\ t_1\ t_2$) $\rightarrow t_1$

right(`Br` $x\ t_1\ t_2$) $\rightarrow t_2$

If you use encodings of other data structures, state the properties assumed.

[6 marks]

Consider the `ML` functions `f` and `g` defined to satisfy

```
f([],ys) = ys

f(x::xs,ys) = f(xs,x::ys)

g([],ys) = ys

g(x::xs,ys) = x :: g(xs,ys)
```

Using list induction, prove `f(f(xs,[]), []) = xs`. [14 marks]

A model answer from Larry Paulson

Here are the definitions:

`leaf` $= \lambda z.z$

`Br` $= \lambda x t_1 t_2$.`pair` **false** (`pair` x (`pair` $t_1\ t_2$))

`isleaf` $=$ `fst`

`label` $= \lambda t$.`fst` (`snd` t)

`left` $= \lambda t$.`fst`(`snd` (`snd` t))

right $= \lambda t$.`snd`(`snd` (`snd` t))

This assumes `fst`(`pair` $x\ y$) $\rightarrow x$ and `snd`(`pair` $x\ y$) $\rightarrow y$; the definition of `leaf` given above actually relies on `fst` $= \lambda p.p$ **true**. Thus `isleaf leaf` β-reduces successively to $(\lambda z.z)$**true** then to **true**. The other laws hold trivially.

Now for the second part.

Obviously, g is the append function. The given formula requires generalization before induction. Perform induction on xs in

$\forall ys.f(f(xs, ys), zs) = f(ys, g(xs, zs))$.

The result will then follow by the definition of f and by $g(xs, []) = xs$, itself provable by a trivial induction.

The base case of the induction holds by definition of f and g:

$f(f([], ys), zs) = f(ys, zs) = f(ys, g([], zs))$.

For the inductive step we have (mostly by definition)

$f(f(x :: xs, ys), zs) = f(f(xs, x :: ys), zs)$

$= f(x :: ys, g(xs, zs))$ (induction hypothesis)

$= f(ys, x :: g(xs, zs))$

$= f(ys, g(x :: xs, zs))$

The main difficulty lies in choosing the right generalization.

1996:12:11

Any recursive function f :`int -> int` declared like

$f\,n =$: `if` $n = 0$ `then` a `else` $g(n, f(n-1))$

is a fixed point for the function F:`(int -> int) -> (int -> int)` declared as

$F\ f\ n =$: `if` $n = 0$ `then` a `else` $g(n, f(n-1))$

If $\mathcal{F}$ is a λ-term for F then $Y\mathcal{F}$ is a λ-term for f.

The displayed attempt to declare Y in `ML` won't work because it won't typecheck. Instead we should declare

```
fun Y f x = f (Y f) x;

val Y = fn : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
- -

val fac = Y ( fn g => fn x => if x=0 then 1 else x*(g (x-1)));
val fac = fn : int -> int
```

Notice that `fun Y f x => f (Y f) x`

doesn't cause non-termination because `ML` evaluates functions only when they have been given all their arguments (I think; I don't believe it does any partial evaluation). So if you evaluate `fac`, it goes something like this:

```
val fac => Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)));

fac 1 => Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1))) 1
```

(now Y has enough arguments, so gets evaluated)

```
=> ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))
    (Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))) 1

=> if 1= 0 then 1
   else 1*(Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))) (1-1))

=> 1*((Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))) 0))

=> 1*( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))
    (Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))) 0

=> 1*(if 0 = 0 then 1
      else x*((Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))) (0-1))))

=> 1*1

=> 1
```

Of course, we're just pretending that it uses substitution rather than keeping variables in store locations, but that would make what's already rather messy too much worse.

Anyway, the crucial things are that

- it does look at the definition of `Y`
- it works because the applicative order evaluation strategy doesn't apply to function arguments.

(Thanks to Jon Fairbairn)

Grant Warrell sez the answer that was wanted was one not using recursion, to wit:

```
- val Y = (fn t=>t(T t));
val Y = fn : ('a t -> 'a) -> 'a
```

[*HOLE ML detects failure of typechecking in cases like 'xx' by doing an occurs check. What happens if we write it in PROLOG which famously doesn't do an occurs check? What is the class of things that suddenly become OK?*]

1997:5:11

(a) [bookwork]

(b) (i) Y_M is $\lambda f.WWM$. This is

$$\lambda f.[\lambda x \lambda z.f(xxz)\ W\ M]$$

Now do two β-reductions inside the square bracket (W/x and M/z) getting

$$\lambda f.f(WWM)$$

which is in head normal form.

(ii) $Y_M(KI)$ is $(\lambda f.WWM)\ (KI)$
Using (i) this becomes $\lambda f.f(WWM))\ (KI)$ (then do a β-reduction) $(KI)(WWM)$ which of course is just I.

(iii) We follow the same track as (ii) to get $K(WWM)$. If that has a HNF i'll eat my hat.

(c) If a λ-term M is in HNF, then the body is of the form fN for some vbl f and some term N. If we now ensure that f gets instantiated to KI, we will find that $(KI)N$ β-reduces to I as desired. As far as i can see it doesn't matter what the other variables get instantiated to.

The fact that $Y_M(KI)$ β-reduces to I shows that Y_M is solvable. II is also I so $Y_M(KI)$ is solvable too.

The third one is not solvable. Y_M is a fixpoint combinator so $Y_M K$ is a fixpoint for K. But no fixpoint for K can possibly be solvable: whatever you apply it to you just get back what you started with and that can't be I—beco's I is not a fixed point for K!

1998:13:10

```
fun update(s, x, i) = if x = y then i else s(y)
   fun interpret  Assign(a, Expr(e)) s1 = fun update(s1, x, e(s1)))
    — interpret  Sequence(c1, c2) s1 = interpret(c2,interpret(c2, s1))
    — interpret  while_do(Expr(e), c) s1 = if not e(s1)<> 0 then interpret(c, s1)
elses1
```

Chapter 3

Semantics

Some tripos questions: 1991:13:9 1992:7:9 1992:9:10 1996:9:10

Three induction questions

Andy's semantics exercise 2.5.2

The subset D of $\mathbb{N} \times \mathbb{N}$ defined by the two rules

$$\langle n, 0\rangle \in D$$

and

$$\frac{\langle n, n'\rangle \in D}{\langle n, n + n'\rangle \in D}$$

Is the intersection of all subsets of $\mathbb{N} \times \mathbb{N}$ which contains all ordered pairs of the form $\langle n, 0\rangle$ and contains $\langle n, n + n'\rangle$ whenever it contains $\langle n, n'\rangle$.

The set $\{\langle n, n'\rangle : (\exists k \in \mathbb{N})(n' = k * n)\}$ contains all ordered pairs of the form $\langle n, 0\rangle$ and contains $\langle n, n + n'\rangle$ whenever it contains $\langle n, n'\rangle$. Let us demonstrate this.

For every $n \in \mathbb{N}$, the tuple $\langle n, 0\rangle$ satisfies $(\exists k \in \mathbb{N})(0 = k * n)$, since the desired k is 0. Now suppose that for the ordered pair $\langle n, n'\rangle$ $(\exists k \in \mathbb{N})(n' = k * n)$. We seek k' such that $n + n' = k' * n$. Clearly the desired k' is $k + 1$.

Therefore $\{\langle n, n'\rangle : (\exists k \in \mathbb{N})(n' = k * n)\}$ contains every pair $\langle n.0\rangle$ and is closed under the operation in the rule so it is a superset of D. In other words it contains every member of D as desired.

For the other direction ("Use Mathematical induction on k to show conversely that for all $n, n' \in \mathbb{N}$, if $n' = k * n$ then $\langle n, n'\rangle \in D$") we note that the rule $\langle n, 0\rangle \in D$ disposes of the case where $k = 0$. For the induction suppose k is such that for all $n, n' \in \mathbb{N}$ if $n' = k * n$ then $\langle n, n'\rangle \in D$; then the rule $\frac{\langle n, n'\rangle \in D}{\langle n, n + n'\rangle \in D}$ ensures that the same is true for $k + 1$.

1992:9:10

R is a wellfounded relation on X if for all nonempty $X' \subseteq X$ $(\exists y \in X')(\forall x)(R(x, y) \rightarrow x \notin X')$.

The effect of the $\ll$ relation is to ensure that if $w \ll u$ then at the leftmost place where they disagree w has a lower number than u. If we fix the length of words we are looking at, and think of a word as a numeral to base 3 (pretending that the alphabet was $\{0, 1, 2\}$ instead of $\{1, 2, 3\}$) then $\ll$ is a subset of the relation "is less than" on numbers whose base-3 representations are of that length (or less, beco's

of leading zeroes) and is thus obviously wellfounded. On the whole of Σ^*, $\ll$ is actually the lexicographic order induced on Σ^*, by taking $1 < 2 < 3$ and this order is well known to be illfounded, because of the sequence 2, 12, 112, 1112, 11112, ...

The recursive declaration of the relation $\rightarrow$ ensures that if $w \rightarrow u$ then the number corresponding to the numeral w is strictly bigger than the number corresponding to the numeral u. We do this by structural induction. Finally to show that $\ll$ is wellfounded it is sufficient to note that it is a subset of $<$ on $\mathbb{N}$, since any subset of a wellfounded relation is wellfounded and $<$ is wellfounded.

Finally we note that we can prove by structural induction that if $w \rightarrow u$ then w and u are the same length. So the words which are $\rightarrow$-maximal are those consisting entirely of 3s.

1991:13:9

The principle of complete induction for $<$ over $\mathbb{N}$ is the following inference scheme:

$$\frac{(\forall n)((\forall m)(m < n \rightarrow \phi(m)) \rightarrow \phi(n))}{(\forall n)(\phi(n))}$$

The assertion we are asked to prove has two number variables in it. Are we to fix m and prove it by induction on n? Or the other way round? Or what? The fact that the declaration of `divide` involves a recursion on m but not on n is a dead give-away that we should be trying to prove the allegation by induction on m.

So let us assume that it is true for all $m' < m$ that

$$\texttt{divide}(m', n) = m' \texttt{ div } n$$

In particular this is true for $m' = m - n$, so $\texttt{divide}(m - n, n) = (m - n) \texttt{ div } n$. But $m \texttt{ div } n = ((m - n) \texttt{ div } n) + 1$, giving us the desired result.

These trees are binary trees with natural numbers at the endpoints. The obvious induction principle is this: if ϕ is something that is true of all singleton trees (leaves) and is true of the result of gluing together two trees t_1 and t_2 by the `fork` constructor, then it is true of *all* `intree`s.

`rv` fixes all singleton trees (leaves) and so $\texttt{sum}(\texttt{rv } t) = \texttt{sum}(t)$ certainly holds for such trees. To verify the induction we need to show that if $\texttt{sum}(\texttt{rv } t) = \texttt{sum}(t)$ holds for both parents of a tree t', then it holds for t' as well. We reason thus:

$\texttt{sum}(\texttt{rv } t) =$ (expanding t)

$\texttt{sum}(\texttt{rv}(\texttt{Fork}(t_1, t_2))) =$ (expanding the recursive definition of `rv`)

$\texttt{sum}(\texttt{Fork}(\texttt{rv } t_1, \texttt{rv } t_2)) =$ (expanding the recursive definition of `sum`)

$\texttt{sum}(\texttt{rv } t_1) + \texttt{sum}(\texttt{rv } t_2) =$ (applying induction hypothesis)

$\texttt{sum}(t_1) + \texttt{sum}(t_2) =$ (contracting t)

$\texttt{sum}(t)$

1993:8:10

Fixed point induction

$$\frac{P(\bot) \quad (\forall d \in \mathcal{D})(P(d) \rightarrow P(f`d))}{P(fix(f))}$$

where $fix(f)$ is the least fixed point of f and P is an inclusive subset of $\mathcal{D}$.

$$\delta\ f\ d\ = \texttt{in}\langle \texttt{hd}(d), f(\texttt{tl}\ d)\rangle$$

... or $\bot$ if $d = \bot$.

We will need the fact that the only fixed point for δ is the identity. Let $f = \delta f$. For each $d \in \mathcal{D}$ we prove by induction on n that the nth member of fd = nth member of d.

maps is that function which increases each coordinate by 1.

To show that there is at most one fixed point for $\lambda d.\texttt{in}\langle$0,maps(d)$\rangle$ follow the hint. Suppose d_1 and d_2 are both fixed points for $\lambda d.\texttt{in}\langle$0,maps(d)$\rangle$. Match the '$\mathcal{D}$' above to `integer_streams -> integer_streams` and let $P(d)$ say that (i) d commutes with *maps* and (ii) $d(d_1) \sqsubseteq d_2$. f has to be δ.

Claim: δ satisifes (i). *i.e.* a commutes with *maps* implies that $\delta(a)$ commutes with *maps*.

$(\delta a)(maps\ x)$ =$\texttt{in}\langle \texttt{hd}\ (maps\ x), a\texttt{tl}\ (maps\ x)\rangle$

=$\texttt{in}\ \langle \texttt{hd}x + 1, a(maps\ \texttt{tl}\ x)\rangle$

but a commutes with *maps* to give

=$\texttt{in}\ \langle(\texttt{hd}\ x) + 1, maps\ (a\ \texttt{tl}\ x)\rangle$

= $maps\ \ \texttt{in}\ \langle \texttt{hd}\ x, a\ (\texttt{tl}\ x)\rangle$

= $maps\ ((\delta\ a)x)$

We also need to know that δ satisfies (ii). *i.e.* if $a(d_1) \sqsubseteq d_2$ then $\delta(a)(d_1) \sqsubseteq d_2$

⋮

We check easily that $\bot$ (the empty function) satisfies (i) and (ii) and that the class of functions satisfying (i) and (ii) is an ω-closed subset of $\mathcal{D}$.

1993:9:10

Definition of wellfounded relation is bookwork.

Suppose $(\forall y)(y \trianglelefteq x \rightarrow f(y) = g(y))$. We wish to infer that $f(x) = g(x)$. If $x > 100$ this is true irrespective of the induction hypothesis so we'll go straight to the hard part where $x \leq 100$. We want to prove that $f(f(x + 11)) = 91$.

A good rule in life is that if a stranger gives you something for nothing then it isn't worth having, but exams are not life so perhaps the idea of considering the case $x = 100$ separately isn't necessarily a bad one.

If $x = 100$ then $f(x) =: f(f(111))$. $f(111) = 101$ and $f(101) = 91$, which is what we wanted.

Notice that we haven't used the induction hypothesis yet. It might be a good idea to get a feel for what the relation $\trianglelefteq$ does. It is the reverse of the usual ordering on numbers ≤ 100. We've got $f(100)$ sorted out, so let's orient ourselves by thinking about $f(99)$, since 99 is $\trianglelefteq$-minimal among the things we haven't yet considered. This must be $f(f(110))$. $f(110)$ is 100 and $f(100)$ we have just shown to be 91.

So let's try the case $90 \leq x < 100$. Naturally $f(x) =: f(f(x + 11)$. But $x + 11 > 100$ so $f(x + 11) = (x + 11) - 10 = x + 1$. But $x + 1 \trianglelefteq x$ and so, by induction hypothesis $f(x + 1) = 91$. This tells us that $f(x) = 91$ as desired.

Finally we have to deal with the case $x < 90$. As before $f(x) =: f(f(x+11))$. $x+11$ is now 100 at most, so f of it is 91, as we showed (without using the induction hypothesis!). $f(91) = 91$, as we showed in the previous paragraph. This completes the proof.

1994:8:12

An **inclusive subset** of a cpo simply one that is closed under taking least upper bounds of (countable) chains — aka "chain-closed subset".

The principle of fixed point induction is:

$$\frac{P(\bot) \quad (\forall d \in \mathcal{D})(P(d) \rightarrow P(f`d))}{P(fix(f))}$$

where $fix(f)$ is the least fixed point of f and P is an inclusive subset of $\mathcal{D}$.

1996:6:12

(a) *An arbitrary intersection of closed subsets is a closed subset.*

Let C be an intersection of a family $\mathcal{C}$ of closed subsets, and suppose $x \in C$. Then x belongs to every member of $\mathcal{C}$. Then any y below x likewise belongs to every member of $\mathcal{C}$ and therefore to C. That takes care of the first condition.

Now let $\langle x_i : i \in \mathbb{N}\rangle$ be a $\sqsubseteq$-increasing sequence of elements of C. Each x_i belongs to every element of $\mathcal{C}$ and therefore, since each such element is closed, the sup of this sequence belongs to every member of $\mathcal{C}$ and therefore to C.

(b) *A union of finitely many closed subsets is a closed subset*

Let C be a union of a finite family $\mathcal{C}$ of closed subsets, If $y \in C$ and $x \sqsubseteq y$ then y belongs to some member of $\mathcal{C}$ and this (together with the fact that $x \sqsubseteq y$) implies that x belongs to that member of $\mathcal{C}$, and therefore to C. That takes care of the first condition. (Notice that we haven't yet used the fact that $\mathcal{C}$ is finite.)

Now let x_∞ be the sup of a $\sqsubseteq$-increasing sequence $\langle x_i : i \in \mathbb{N}\rangle$ of elements of C. We want x to be in C. Each x_i belongs to a member of $\mathcal{C}$ but there are infinitely many x_i and only finitely many things in $\mathcal{C}$, so one of the things in $\mathcal{C}$ must have infinitely many x_i in it. But then that thing in $\mathcal{C}$ must contain x_∞, being closed, So x_∞ is in C as desired.

(c) This set trivially satisfies the downward-closed condition. For the other condition notice that an sup of an ω-sequence (indeed any sequence) of things all $\sqsubseteq x$ is itself $\sqsubseteq x$ and therefore in $\downarrow x$.

(d) *The preimage of a closed set in a cts function is closed*

Let C be a closed subset of $\mathcal{D} = \langle D, \sqsubseteq_D\rangle$ and let f be a continuous function $\mathcal{E} \to \mathcal{D}$ whose range is C.

Suppose $f(y) \in C$ and $x \sqsubseteq_E y$. Then $f(x) \sqsubseteq_D f(y)$ by monotonicity of f. This takes care of the first condition.

Finally suppose that x_∞ is the ($\sqsubseteq_E$ -) sup of a $\sqsubseteq_E$-increasing sequence $\langle x_i : i \in \mathbb{N}\rangle$ of elements of $\mathcal{E}$ such that for each $i \in \mathbb{N}$ $f(x_i) \in C$. Then, by continuity of f, $f(x_\infty)$ is the ($\sqsubseteq_D$-) sup of the sequence $\langle f(x_i) : i \in \mathbb{N}\rangle$. All these elements are in C, so their sup is in C, so x_∞ is in the preimage as desired.

Question 6 on Andy's example sheet

$\mathcal{D}$,$\mathcal{E}$ and $\mathcal{F}$ are cpo's and $f : \mathcal{D} \times \mathcal{E} \to \mathcal{F}$ has cts projections. i.e., for all $d \in \mathcal{D}$ and all $e \in \mathcal{E}$

$$\lambda y \in \mathcal{E}.f(d, y)$$

and

$$\lambda y \in \mathcal{D}.f(y, e)$$

are continuous.

Prove that f is cts. First we show that f is monotone. Suppose $x \leq_{\mathcal{D}} x'$ and $y \leq_{\mathcal{E}} y'$. Then $f`\langle x, y\rangle \leq f`\langle x', y\rangle$ by continuity of $\lambda y \in \mathcal{D}.f(y, e)$ and then $f`\langle x', y\rangle \leq f`\langle x', y'\rangle$ by continuity of $\lambda y \in \mathcal{D}.f(y, e)$.

Now it remains to show that f is cts at limits. We are given a sequence

$$\langle d_0, e_0\rangle, \langle d_1, e_1\rangle, \langle d_2, e_2\rangle \ldots$$

Its sup is $\langle d_\infty, e_\infty\rangle$, where d_∞, e_∞ are the two sups of the two sequences. We have to show that $f`\langle d_\infty, e_\infty\rangle = sup_{n\in\mathbb{N}} f`\langle d_n, e_n\rangle$

Consider the two sequences

$$(1): \quad \langle d_0, e_\infty\rangle, \langle d_1, e_\infty\rangle, \langle d_2, e_\infty\rangle \ldots$$

$$(2): \quad \langle d_\infty, e_0\rangle, \langle d_\infty, e_1\rangle, \langle d_\infty, e_2\rangle \ldots$$

and fs of them

$$(3): \quad f`\langle d_0, e_\infty\rangle, f`\langle d_1, e_\infty\rangle, f`\langle d_2, e_\infty\rangle \ldots$$

$$(4): \quad f`\langle d_\infty, e_0\rangle, f`\langle d_\infty, e_1\rangle, f`\langle d_\infty, e_2\rangle \ldots$$

By continuity of $\lambda y \in \mathcal{D}.f(y, e)$ the sup of (3) must be $f`\langle d_\infty, e_\infty\rangle$ and ditto (4). Since $f$ is monotone we know at least that $f`\langle d_n, e_n\rangle \leq f`\langle d_\infty, e_\infty\rangle$ for all $n \in \mathbb{N}$, so $sup_{n\in\mathbb{N}} f`\langle d_n, e_n\rangle \leq f`\langle d_\infty, e_\infty\rangle$.

Now fix some $k \in \mathbb{N}$ and think about $sup_{n\in\mathbb{N}} f`\langle d_n, e_k\rangle$. By continuity of $\lambda y \in \mathcal{D}.f(y, e)$ this must be $f\langle d_\infty, e_k\rangle$. By continuity of $\lambda y \in \mathcal{E}.f(d, y)$, $f`\langle d_\infty, e_\infty\rangle$ is the sup of all the $f\langle d_\infty, e_k\rangle$. But each $f\langle d_\infty, e_k\rangle$ is below $sup_{n\in\mathbb{N}} f`\langle d_n, e_n\rangle$ because the $k$ is constant and the $e_n$ are eventually bigger than $e_k$. So $f`\langle d_\infty, e_\infty\rangle$ is the sup of a lot of things all $\leq sup_{n\in\mathbb{N}} f`\langle d_n, e_n\rangle$ so it is $\leq sup_{n\in\mathbb{N}} f`\langle d_n, e_n\rangle$ itself, and we've got the other half of the inequality.

1995:5:12

Tim Waugh writes: Continuation semantics of IMP (Cont = States → $A_\perp$, [[-]]: Comm → (consts → (states → $A_\perp$)))

[[`skip`]] k S = k(S)

[[x:= ie]] k S = k (S[[[ie]]S/x])

[[C_1;C_2]] k S = [[C_1]]([[C_2]] k) S

[[if be then C_1 else C_2]] k S = [[be]] S `=>` [[C_1]] k S — [[C_1]] k S

[[`while` be do C]] k S = [[be]] S `=>` [[C]] ([[`while` be do C]] k) S — k(S)

[[`while` be do C]] k S fix Φ; Φ = $\lambda f \in \{$`states` → $A_\perp\}, \lambda k \in \{$`states` → $A_\perp\}$. [[be]] S `=>` f*

[[abort]] k S = `Err`

To add `exit` and `orelse` [[-]] is redefined as

[[-]] Comm → (cont → (cont → (`states` → $A_\perp$)))

and the commands are redefined as

[[`skip`]] k e S = k(S)

[[x:= ie]] k e S = k (S[[[ie]]S/x])
[[C_1;C_2]] k e S = [[C_1]]([[C_2]] k e) e S
[[if be then C_1 else C_2]] k e S = [[be]] S `=>` [[C_1]] k e S — [[C_1]] k e S
[[`while` be do C]] k e S = [[be]] S `=>` [[C]] ([[`while` be do C]] k e) e S — k(S)
[[abort]] k e S = `Err`
[[`exit`]] k e S = e(S)
[[C_1 `orelse` C_2]] k e S = [[C_1]] k ([[C_2]] k e) S

For `exit` to behave as abort outside an '`orelse`' clause, e should initially be $\lambda S \in$ `states`. `Err`.

The function b `=>` $C_1|C_2$ is a conditional expression defined as b `=>` $C_1|C_2 = C_1$ if $b =$ `true`; $= C_2$ if $b =$ `false`.

Chapter 4

Part II types

A Lambda-2 question

We want a lambda-2 term of type

$$\forall\alpha\forall\beta[(\forall\gamma)((\alpha\to(\beta\to\gamma))\to\gamma)\to\alpha]$$

Specialise the γ to α and apply to a suitable version of the K combinator. So we want

$$\Lambda\alpha.\Lambda\beta.\lambda x^{(\forall\gamma)((\alpha\to(\beta\to\gamma))\to\gamma)}.x_\alpha K^{\alpha\to(\beta\to\alpha)}$$

... where K is of course $\lambda uv.u$ as usual.

(Here the superscript on a variable denotes its type. The subscript on a variable (as in "X_α") is an instruction to specialise that object by substituting—in this case α—for the bound variable (in this case γ)).

Why not specialise the λx to $\alpha\to(\beta\to\alpha)\to\alpha$ to start with? The idea is that then you are eating something that can never give rise to a β, but only an α.

1991:9:11

Define `int` to be the type $(\forall\alpha)((\alpha\to\alpha)\to(\alpha\to\alpha))$. This is a halfway-sensible name for this type, 'cos it's obvious that all Church numerals are polymorphic chaps of type $((\alpha\to\alpha)\to(\alpha\to\alpha))$ for all α. (What is less obvious, and nobody will ask you to prove it, is that Church numerals are the *only* polymorphic things of type $(\alpha\to\alpha)\to(\alpha\to\alpha)$ for all α.)

Now, what are Church numerals for? Answer: take a thing x of any type you wish, like α, take a function f of type $\alpha\to\alpha$, and a church numeral n, and do f n times to x. This activity is described by the higher-order lambda term

$$\Lambda\alpha.\lambda x_\alpha\lambda f_{\alpha\to\alpha}\lambda n_{\texttt{int}}.nfx$$

Next we have to check that it does what it is supposed to do, and that its type is what it is supposed to be. The $\Lambda\alpha$ at the beginning ensures that it is of type $(\forall\alpha)(\ldots)$ and the rest is easy.

1992:8:11

The only remotely hard bit is the second part (12 marks!) Do it by ("structural") induction on `e`.

- If `e` is an identifier and $\Gamma\vdash$ `e` $:\sigma$ this can only be because Γ contains `e` $:\sigma$. So by the definition of v on contexts (v of a context is the least value that v assigns to a type mentioned in that context) $v(\Gamma)\le v(\sigma)$ as desired.

- Suppose `e` is $\lambda x.$`e'`, and that Γ thinks that `e` is of type $\sigma \to \tau$. (I suppose it is ok to assume that the type of `e` is a function type, but it might be an idea to put out some patter on this point). By the construction of proof trees for type assignments, if we have $\Gamma \vdash$ `e'` $: \sigma \to \tau$ then this must have come from $\Gamma \vdash x : \sigma$ and $\Gamma \vdash$ `e'` $: \tau$. The induction hypothesis tells us that these can happen only if $v(\Gamma) \leq v(\sigma)$ and $v(\Gamma) \leq v(\tau)$. But, by the inductive way in which v is defined on bigger types in terms of its definition on smaller ones, we know that $v(\Gamma) \leq v(\sigma \to \tau)$ as desired.

- Suppose `e` is $\texttt{e}_1\texttt{e}_2$. If Γ thinks that `e` is of type σ then for some τ we have $\Gamma \vdash \texttt{e}_1 : \tau \to \sigma$ and $\Gamma \vdash \texttt{e}_2 : \tau$. Then, by induction hypothesis, we have $v(\Gamma) \leq v(\tau \to \sigma)$ and $v(\Gamma) \leq v(\tau)$. We now have to consider two possibilities—whether or not $v(\tau) \leq v(\sigma)$. If $v(\tau) \leq v(\sigma)$ then the fact that $v(\Gamma) \leq v(\tau)$ is enough to ensure that $v(\Gamma) \leq v(\sigma)$. Contrariwise, if $v(\tau) > v(\sigma)$ then $v(\tau \to \sigma) = v(\sigma)$ and we already know that $v(\Gamma) \leq v(\tau \to \sigma)$ so as before we infer $v(\Gamma) \leq v(\sigma)$ as desired.

For the next part ("Deduce that if [ ]$\vdash$ `e` $: \sigma \ldots$") notice that v of the empty context is 1.

Next ("for four marks!!") deduce that there is no `ML` expression `e` such [ ]$\vdash$ `e` $: ((\alpha \to \alpha') \to \alpha) \to \alpha$.

Well, if there were such an `e` then for *any* v we would have $v(((\alpha \to \alpha') \to \alpha) \to \alpha) = 1$. We could only have drawn the conclusion that [ ] $\vdash$ `e` $: ((\alpha \to \alpha') \to \alpha) \to \alpha$ by means of the rules that assigns function types to lambda terms. That is, `e` must have been $\lambda x.$`e'` and we had deduced [ ] $\vdash$ `e` $: ((\alpha \to \alpha') \to \alpha) \to \alpha$ from $x : (\alpha \to \alpha') \to \alpha \vdash$

stuff missing

we must have had

By the way v is defined on function types we would have had to have either

> *(This reveals someone's hidden agenda, and i suspect Larry Paulson. This expression,* $(((\alpha \to \alpha') \to \alpha) \to \alpha)$*, if thought of as a propositional formula, is a truth-table tautology, but it is not provable intuitionistically. This is an important fact. The formula itself is called* **Peirce's Law**)

[*HOLE I have done this in rather a hurry and i may have missed out some details to do with deleting*] $x : \sigma$ from Γ *in the case where* `e` *is* $\lambda x.$`e'`*. I can't remember the small print of these recursions. Please check it yourselves: it won't kill you, and let me know if this answer looks halfway-sensible.*

Question 6 on the types sheet

I still don't really know what is going on, since it turns out that it's all category theory at heart. However i am clinging on to the following pieces of wreckage in the hope that they will eventually bear fruit:

It probably also helps to consider a few examples:

(i) If $\tau(\alpha)$ is $1 + (\kappa \times \alpha)$ (+ is disjoint union then ι turns out to be κ list. (ii)If $\tau(\alpha)$ is $1 + (\alpha \times \alpha)$ then ι turns out to be binary trees. If you want binary trees with αs at each node you want to start with $1 + (\alpha \times \kappa \times \alpha \times \kappa)$ (iii) I think (tho' i don't remember this bit very clearly) that if you start with $1 + \alpha$ you get Nats.

Chapter 5

Program correctness

"Is the following specification true?

```
|- {X=x /\ Y=y} X:=X+Y; Y:=X-Y; X:=X-Y {Y=x /\ X=y}
```

Of so, prove it. If not, give the circumstances in which it fails."

It looks to me that the spec is true, with VC:

```
X = x /\ Y = y ==> (X+Y)-Y = x /\ (X+Y)-((X+Y)-Y) = y
```

My man says: if X and Y are the same variable then after

```
X := X + Y
```

we have

```
X = x + x = 2x
```

then after

```
Y := X - Y   we have Y = X - X = 0
```

because the two variables are the same!

Ah ha! I see the point. Quite subtle! I guess what you said was right, namely that there is a lurking assumption that variables with distinct names were distinct.

Mike

Paul Curzon has a short list of exercises which aren't quite right:

There is an ambiguity in the notation used with While loops. It is not clear whether

```
WHILE S DO C1; C2
```

means

```
WHILE S DO (C1;C2)
```

or

```
(WHILE S DO C1); C2
```

especially since the indentation often suggests the former when the latter is meant (eg see Exercise 21)

Also in exercise 21 either y should be Y or the precondition should include Y = y

In exercise 23, $N > 0$ is given as an invariant when it is not. I think it needs to be $N \geq 0$ in which case its also needed in the precondition (or possibly its not needed at all).

There is confusion throughout over whether the variables range over natural numbers or integers (or even reals). Some of the exercises assume naturals I think.

1992:6:12

Write down a condition $P(k)$, which holds for infinitely many k, such that the following specification becomes true:

$\{X = x \wedge x \geq 0 \wedge K = k \wedge P(k)\}$ `WHILE` $K > 1$ `DO` $X := X * X; K :=$ K `DIV` 2 $\{X = x^k \wedge K = 1\}$

A bit of ingenuity reveals that $P(k)$ has to be $(\exists n)(k = 2^n)$. This P is not only sufficient but is necessary.

The `WHILE` rule on p 45 of the notes allows us to claim we have proved something of the form

$\{P\}$ `WHILE` S `DO` $\{R\}$ `C` $\{Q\}$

as long as we have
$\vdash P \rightarrow R$
$\vdash (R \wedge \neg S) \rightarrow Q$
$\{R \wedge S\}$ `C` $\{R\}$
Now we have to do some pattern-matching. We know that

P is $X = x \wedge x \geq 0 \wedge K = k \wedge (\exists n)(k = 2^n)$

and that

Q is $X = x^k \wedge K = 1$

and S is $K > 1$ but we have to come up with an R to decorate the `WHILE` loop with. This has to be something that is true each time we go through the loop. A good candidate for R would appear to be $K \geq 1 \wedge X = x^{k \texttt{ DIV } K} \wedge x \geq 0 \wedge (\exists n)(k = 2^n)$

$P \rightarrow R$

is

$X = x \wedge x \geq 0 \wedge K = k \wedge (\exists n)(k = 2^n) \rightarrow K \geq 1 \wedge X = x^{k \texttt{ DIV } K} \wedge x \geq 0 \wedge (\exists n)(k = 2^n)$

which certainly follows by arithmetic. (We are assuming that all our variables take values in $\mathbb{N}$, so that $K \geq 1$ follows from $K = k \wedge (\exists n)(k = 2^n)$)

We also want $(R \wedge \neg S) \rightarrow Q$. This is

$(K \geq 1 \wedge X = x^{k \texttt{ DIV } K} \wedge x \geq 0 \wedge (\exists n)(k = 2^n) \wedge \neg(K > 1)) \rightarrow (X = x^k \wedge K = 1)$

and this too will follow by arithmetic. (If $\neg(K > 1)$ then the only possibility is $K = 1$ and then $X = x^{k \texttt{ DIV } K}$ simplifies to $X = x^k$ as desired.)

It remains to deal with

$\{R \wedge S\}$ `C` $\{R\}$

... which is

$\{K \geq 1 \wedge X = x^{k \texttt{ DIV } K} \wedge x \geq 0 \wedge (\exists n)(k = 2^n) \wedge K > 1\} X := X * X; K :=$ K `DIV` 2 $\{K \geq 1 \wedge X = x^{k \texttt{ DIV } K} \wedge x \geq 0 \wedge (\exists n)(k = 2^n)\}$

Now we do not need *both* $K \geq 1$ and $K > 1$ in the precondition and we will delete $K \geq 1$ since it follows from the other, to get

$$\{X = x^{k\ \mathtt{DIV}\ K} \wedge x \geq 0 \wedge (\exists n)(k = 2^n) \wedge K > 1\}\ X := X * X; K := K\ \mathtt{DIV}\ 2\ \{K \geq 1 \wedge X = x^{k\ \mathtt{DIV}\ K} \wedge x \geq 0 \wedge (\exists n)(k = 2^n)\}$$

Now we have to invoke the sequencing rule to find something that is true after we have reset X but before we have reset Y, or a sequence of things to put in between such that we can infer along the chain by arithmetic (precondition strengthening and postcondition weakening). It's pretty obvious that this has to be something like

$$\{X = x^{k\ \mathtt{DIV}\ (K\ \mathtt{DIV}\ 2)} \wedge x \geq 0 \wedge (\exists n)(k = 2^n) \wedge K > 1\}$$

. . . so let's try it. This means we have to prove

1. $\{X = x^{k\ \mathtt{DIV}\ K} \wedge x \geq 0 \wedge (\exists n)(k = 2^n) \wedge K > 1\}\ \ X := X * X\ \ \{X = x^{k\ \mathtt{DIV}\ (K\ \mathtt{DIV}\ 2)} \wedge x \geq 0 \wedge (\exists n)(k = 2^n) \wedge K > 1\}$
2. $\{X = x^{k\ \mathtt{DIV}\ (K\ \mathtt{DIV}\ 2)} \wedge x \geq 0 \wedge (\exists n)(k = 2^n) \wedge K > 1\}\ \ K := K\ \mathtt{DIV}\ 2\ \{K \geq 1 \wedge X = x^{k\ \mathtt{DIV}\ K} \wedge x \geq 0 \wedge (\exists n)(k = 2^n)\}$

To prove (1) we need worry only about those bits of the precondition and postcondition that concern X, namely

$$\{X = x^{k\ \mathtt{DIV}\ K}\}\ \ X := X * X\ \ \{X = x^{k\ \mathtt{DIV}\ (K\ \mathtt{DIV}\ 2)}\}$$

and for (2) we can ignore things that don't mention K:

$$\{X = x^{k\ \mathtt{DIV}\ (K\ \mathtt{DIV}\ 2)} \wedge K > 1\}\ K := K\ \mathtt{DIV}\ 2\ \{K \geq 1 \wedge X = x^{k\ \mathtt{DIV}\ K}\}$$

The second follows easily from the assignment rule whereas the first needs the arithmetic theorem that if $a = b^{c\ \mathtt{DIV}\ e}$ then $a^2 = b^{c\ \mathtt{DIV}\ (e\ \mathtt{DIV}\ 2)}$ which follows from the rules for exponentiation.

1990:3:10

There are presumably various rules one could set up. The following serves our purposes pretty well:

$$\frac{\bigwedge_{i \in I}(\{P \wedge S_i\} C_i \{P\})}{\{P\}\ {}^*[S_1 \to C_1 | \ldots | S_k \to C_k]\{\bigwedge_{i \in I} \neg S_i \wedge P\}}$$

(If none of the S_i are true then nothing happens and they remain all false. If one of them is true the loop is executed and carries on being executed until they are all false)

To apply this to the case in hand we make the following identifications:

S_1 is $X < Y$

S_2 is $Y < X$

P is $GCD(X, Y) = GCD(x, y)$

The index set I in this case has only two members. (There is only S_1 and S_2.) We must verify that the antecedents of the correctness rule are true in this case. The fact that P is preserved by S_1 and S_2 is conveniently supplied by the management. Accordingly we can deduce the conclusion of the rule, namely

$\{GCD(X, Y) = GCD(x, y)\}$

$$^*[X < Y \to Y := Y - X | Y < X \to X := X - Y]$$

$\{\neg(X < Y) \wedge \neg(Y < X) \wedge GCD(X, Y) = GCD(x, y)\}$

(I have written this on three lines beco's of shortage of space) Now $GCD(X, Y) = GCD(x, y)$ is certainly true when we start. If $X < Y$ and $Y < X$ are both false when we begin then we are done. If at least one of them is true we have satisfied the precondition and we know that the postcondition will hold if the program ever terminates. The postcondition tells us that $X < Y$ and $Y < X$ are both false. It is true that we haven't been told that $<$ is a strict total order (which is what we need to deduce that $X = Y$) but we are conventionally allowed as much arithmetic as we need. $X = GCD(x, y)$ then follows easily.

The point about the last part of the question is that termination is guaranteed only if x and y are both nonzero naturals.

1987:3:6

We are going to treat

```
This REPEATED that TIMES
```

as a `WHILE` loop with an incrementer/decrementer in the obvious way. Thus the middle line of

$$\{\texttt{X} = \texttt{x},\ \texttt{Y} = \texttt{y},\ \texttt{X} \geq 0\}$$
```
Y := Y + 1 REPEATED X TIMES
```
$$\{\texttt{Y} = \texttt{x} + \texttt{y}\}$$

becomes

```
BEGIN VAR X count
     count := 0; WHILE count < x DO
     (Y := Y + 1; count := count + 1)
END
```

We have to find two things to decorate this with. (i) Something that is going to be true after we have initialised `count` and (ii) something to put after `DO` that will be true each time through the loop. Since initialising `count` does nothing to any of the quantities mentioned in the precondition there is—on the face of it—no reason why our candidate for (i) shouldn't be the original precondition itself with "`count` $= 0$" adjoined. It's pretty obvious that the thing we want for (ii) has to be $\texttt{Y} = \texttt{y} + \texttt{count} \wedge \texttt{count} \leq \texttt{x}$. This means that we have generated the correctness task:

$\{\texttt{X} = \texttt{x},\ \texttt{Y} = \texttt{y},\ \texttt{X} \geq 0\}$

```
BEGIN
          count = 0; { X = x ∧ Y = y ∧ X ≥ 0 ∧ count = 0}
  WHILE count < x DO {Y = y + count ∧ count ≤ x }
    (Y := Y+1; count := count+1)
END
```

$\{\texttt{Y} = \texttt{x} + \texttt{y}\}$

This is a sequence of two commands so first we use the sequence rule to split this up into two correctness tasks. We reckon we have already decided what ought to be true after the first command (the assignment) is executed, namely

$\texttt{X} = \texttt{x} \wedge \texttt{Y} = \texttt{y} \wedge \texttt{X} \geq 0 \wedge \texttt{count} = 0$

and this is easy, so we can concentrate on the second, which is:

$\{P\}$ `WHILE` S `DO` $\{R\}$ `C` $\{Q\}$

where

P is $\texttt{X} = \texttt{x} \wedge \texttt{Y} = \texttt{y} \wedge \texttt{X} \geq 0 \wedge \texttt{count} = 0$

R is $\texttt{Y} = \texttt{y}+\texttt{count} \wedge \texttt{count} \leq \texttt{x}$

S is $\texttt{count} < \texttt{x}$

C is `Y := Y + 1; count := count + 1`

Q is $\{\texttt{Y} = \texttt{x} + \texttt{y}\}$

(Look at the wee magic box—in my copy it's on page 45 of the handout—that tells you how to generate subtasks from a decorated `WHILE`-loop.) According to the wee magic box this generates the following things to prove:

1. $\texttt{X} = \texttt{x} \wedge \texttt{Y} = \texttt{y} \wedge \texttt{X} \geq 0 \wedge \texttt{count} = 0 \rightarrow \texttt{Y} = \texttt{y} + \texttt{count}$
2. $(\texttt{Y} = \texttt{y} + \texttt{count} \wedge \texttt{count} \leq \texttt{x}\ ) \wedge \neg(\texttt{count} < \texttt{x}) \rightarrow \texttt{Y} = \texttt{x} + \texttt{y}$
3. $\{\texttt{Y} = \texttt{y}+\texttt{count} \wedge \texttt{count} < \texttt{x}\}$
   `Y := Y + 1; count = count + 1`
   $\{\texttt{Y} = \texttt{y} + \texttt{count}\}$

(1) and (2) are verification conditions and we are supposed to be able to wave our arms over them. Fortunately we can. (3) is another correctness task. (I have dropped the clause '$\texttt{count} \leq \texttt{x}$' since it follows from '$\texttt{count} < \texttt{x}$': we can do this by precondition strengthening.) Here again we have to be clever and find something that is true after the first command has been executed and is such that if we subsequently execute the second command then $\{\texttt{Y} = \texttt{y}+\texttt{count}\}$ is true again. The obvious thing is

$$\texttt{Y} = \texttt{y} + \texttt{count} + 1 \wedge \texttt{count} < \texttt{x}$$

Then we can infer that

$$\texttt{Y} = \texttt{y} + \texttt{count}$$

holds after the assignment statement by the assignment rule.
This leaves us with two further correctness tasks

$\{\texttt{Y} = \texttt{y}+\texttt{count} \wedge \texttt{count} < \texttt{x}\}$
`Y := Y + 1`
$\{\ \texttt{Y} = \texttt{y} + \texttt{count} + 1 \wedge \texttt{count} < \texttt{x}\ \}$

and

$\{\ \texttt{Y} = \texttt{y} + \texttt{count} + 1 \wedge \texttt{count} < \texttt{x}\ \}$
$\texttt{count} := \texttt{count}+1$
$\{\texttt{Y} = \texttt{y} + \texttt{count}\}$

both of which we can attack with the assignment rule.

1991:5:12

We are looking at the code

```
while K < N do
  begin
    K := K+1;
    if A(K) < A(0) then (T:=A(0); A(0):=A(K); A(K):=T)
  end
```

We want to prove that if K was 0 before we started then when we've finished we have

$$(\forall i)(0 \leq i \leq \mathtt{N} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))$$

Consider the result of substituting 'K' for 'N' in this formula:

$$(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))$$

Notice that this is a trivial consequence of $\mathtt{K} = 0$. Therefore we can prove what we wanted—namely that if K was 0 before we started then when we've finished we have $(\forall i)(0 \leq i \leq \mathtt{N} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))$—by proving that if

$$(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))$$

was true before we started than

$$(\forall i)(0 \leq i \leq \mathtt{N} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))$$

will be true after we have finished. This changes the task to

$$\{(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))\}$$

```
while K < N do
  begin
    K := K+1;
    if A(K) < A(0) then (T:=A(0); A(0):=A(K); A(K):=T)
  end
```

$$\{(\forall i)(0 \leq i \leq \mathtt{N} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))\}$$

We have to annotate this thing, and this involves writing after the '`do`' something inside curly brackets which will be true whenever control reaches that point. A good candidate seems to be $(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))$ which we have just seen.

Now we invoke the correctness rule for annotated `while` commands, which says that to prove

$$\{P\} \text{ while } S \text{ do } \{R\}\ C\{Q\}$$

you have to prove

1. $P \rightarrow R$
2. $(R \wedge \neg S) \rightarrow Q$
3. $\{R \wedge S\}C\{R\}$

To invoke this we do some pattern matching:

- S is of course $\mathtt{K} < \mathtt{N}$;

- P is $(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(0))$ and
- Q is $(\forall i)(0 \leq i \leq \mathtt{N} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))$.
- R is $(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))$.
- C is `begin K := K+1; if A(K) < A(0) then (T:=A(0); A(0):=A(K); A(K):=T) end`

C has a `begin` and and `end`. These oblige us to respect certain conditions on the local nature of variables, which are going to be respected anyway (it's all to do with `T`) so we'll forget them. Items 1 and 2 are new verification conditions. 1 is trivial. 2 is

$$(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i)) \wedge \neg(\mathtt{K} < \mathtt{N}) \rightarrow (\forall i)(0 \leq i \leq \mathtt{N} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))$$

We note *en passant* that this is going to be provable.

It also generates a new correctness problem (from "$\{R \wedge S\}C\{R\}$"): we have to prove

$\{(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i)) \wedge \mathtt{K} < \mathtt{N}\}$

```
begin
  K := K+1;
  if A(K) < A(0) then (T:=A(0); A(0):=A(K); A(K):=T)
end
```

$\{(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))\}$

Now we have to use the correctness rule for sequencing, which is pretty obvious really. If you want to prove—given that P was true and you have done C_1 followed by C_2—that Q is now true, you have to find something else (call it P' to give it a name) such that $\{P\}C_1\{P'\}$ and $\{P'\}C_2\{Q\}$. In this case it means we have to find a P so that we are in with a chance of proving both

$\{(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i)) \wedge \mathtt{K} < \mathtt{N}\}$ `K := K+1` $\{P\}$

and

$\{P\}$
`if A(K) < A(0) then (T:=A(0); A(0):=A(K); A(K):=T)`
$\{(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))\}$

A suitable candidate seems to be

$(\forall i)(0 \leq i <\mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i)) \wedge \mathtt{K} \leq \mathtt{N}$

To deal with the first hurdle we use an array assignment rule, and that should be straightforward. The next is a one-armed conditional. We have to prove

$\{(\forall i)(0 \leq i <\mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i)) \wedge \mathtt{K} \leq \mathtt{N}\}$
`if A(K) < A(0) then (T:=A(0); A(0):=A(K); A(K):=T)`
$\{(\forall i)(0 \leq i \leq \mathtt{K} \rightarrow \mathtt{A}(0) \leq \mathtt{A}(i))\}$

The rule for one-armed conditionals is

$$\frac{\{P \wedge S\}\ C\ \{Q\} \quad (P \wedge \neg S) \rightarrow Q}{\{P\}\ \text{If } S \text{ then } C\ \{Q\}}$$

Again we have to do some pattern-matching:

- P is $(\forall i)(0 \leq i <$`K`$\rightarrow$ `A`$(0) \leq$`A`$(i)) \wedge$ `K`$\leq$`N`
- S is `A(K) < A(0)`
- Q is $(\forall i)(0 \leq i \leq$ `K` $\rightarrow$ `A`$(0) \leq$`A`$(i))$

The verification condition $(P \wedge \neg S) \rightarrow Q$ is therefore
$\{(\forall i)(0 \leq i <$`K`$\rightarrow$ `A`$(0) \leq$`A`$(i)) \wedge$`K`$\leq$`N`$\} \wedge \neg$(`A(K) < A(0)`$) \rightarrow \{(\forall i)(0 \leq i \leq$ `K` $\rightarrow$ `A`$(0) \leq$`A`$(i))\}$
This is certainly true. So we have the new correctness problem:

$\{(\forall i)(0 \leq i <$`K`$\rightarrow$ `A`$(0) \leq$`A`$(i)) \wedge$ `K`$\leq$`N`$\wedge$ `A(K) < A(0)`$\}$

```
T:=A(0); A(0):=A(K); A(K):=T
```

$\{(\forall i)(0 \leq i \leq$ `K` $\rightarrow$ `A`$(0) \leq$`A`$(i))\}$

Now we invoke the rule for sequencing again. As before, we have to find things that will be true at intermediate stages in the computation. (In what follows keep your eye on the last clause in the displayed formula). Given the precondition, it seems a fair bet that after the first assignment has been executed the following is true:

$(\forall i)(0 \leq i <$`K`$\rightarrow$ `A`$(0) \leq$`A`$(i)) \wedge$ `K`$\leq$`N`$\wedge$ `A(K) < T`

and after the second assignment has been executed the following will hold:

$(\forall i)(0 \leq i <$`K`$\rightarrow$ `A`$(0) \leq$`A`$(i)) \wedge$ `K`$\leq$`N`$\wedge$ `A(0) < T`

and after the third ...

$(\forall i)(0 \leq i <$`K`$\rightarrow$ `A`$(0) \leq$`A`$(i)) \wedge$ `K`$\leq$`N`$\wedge$ `A(0) < A(K)`

Finally we use postcondition weakening: If `A(0) < A(K)` then certainly `A(0)` $\leq$ `A(K)`.

Chapter 6

Logic and Proof

Notice that one can do it more shortly by complicating the logic $(\exists x_1 \dots x_n)(\forall y)(\bigvee_{i \leq n} y = x_i)$

Programme: sequent calculus is natural deduction with control structures. So a sequent should be read as saying "there is a proof of something on the R using assumptions on the L". This explains why its conjunction on the left but disjunction on the right. One can be quite specific about it. For example, $\wedge$-L. Why does this operation preserve goodness of a sequent? A: Beco's of $\wedge$-elimination: if i can deduce something from the two premises ϕ and ψ, then by two applications of $\wedge$-elimination i can deduce it from $\psi \wedge \phi$.

Similarly we can tell a story about $\rightarrow$-R in terms of $\rightarrow$-int. Perhaps we can tell a story about $\rightarrow$-L in terms of $\rightarrow$-elim.

Notice that Gödel-style proofs suffer from not having the subformula property. Make a meal of it.

6.0.1 2002 p2 q11

One of Larry's exercise
$(\forall x)(P(x) \rightarrow P(f(x))) \vdash (\forall x)(P(x) \rightarrow P(f(f(x))))$
clearly came by a $\forall$-R from
$(\forall x)(P(x) \rightarrow P(f(x))) \vdash P(a) \rightarrow P(f(f(a))))$
which in turn came from a $\rightarrow$-R from
$(\forall x)(P(x) \rightarrow P(f(x))), P(a) \vdash P(f(f(a))))$
Now we have to be clever. A $\forall$-L
$(\forall x)(P(x) \rightarrow P(f(x))), P(a) \rightarrow P(f(a)), P(a) \vdash P(f(f(a))))$
and another
$P(f(a)) \rightarrow P(f(f(a))), P(a) \rightarrow P(f(a)), P(a) \vdash P(f(f(a))))$
after which two $\rightarrow$-L will do it.

6.1 Some Exercises

1. A graph is a set of vertices with undirected edges. It is connected if one can get from any vertex to any other vertex by following edges. The complement of a graph is what you think it is.

   Use resolution to show that a graph and its complement cannot both be disconnected.

2. 1993:3:3

3. The following text is a celebrated argument by Bishop Berkeley which purports to show that nothing exists unconceived. It's a fairly delicate exercise in formalisation.

   > HYLAS : What more easy than to conceive of a tree or house existing by itself, independent of, and unperceived by any mind whatsoever. I do at present time conceive them existing after this manner.
   >
   > PHILONOUS : How say you, Hylas, can you see a thing that is at the same time unseen?
   >
   > HYLAS : No, that were a contradiction.
   >
   > PHILONOUS : Is it not as great a contradiction to talk of *conceiving* a thing which is *unconceived*?
   >
   > HYLAS : It is
   >
   > PHILONOUS : This tree or house therefore, which you think of, is conceived by you?
   >
   > HYLAS : How should it be otherwise?
   >
   > PHILONOUS : And what is conceived is surely in the mind?
   >
   > HYLAS : Without question, that which is conceived exists in the mind.
   >
   > PHILONOUS How then came you to say, you conceived a house or a tree existing independent and out of all mind whatever?
   >
   > HYLAS That was I own an oversight ...

   The exercise here is to formalise this and construct a natural deduction proof that everything is conceived (as Berkeley wants) and perhaps even a sequent calculus proof. This has been discussed in print by my friend Graham Priest, and this treatment draws heavily on his.

   You may, if you wish to think through this exercise very hard, try to work out what new syntactic gadgets one needs to formalise this argument, but i don't recommend it. The best thing is to use the gadgetry Priest introduced.

   Priest starts off by distinguishing, very properly, between **conceiving objects** and **conceiving propositions**. Accordingly in his formalisation he will have *two* devices. One is a sentence operator T which is syntactically a modal operator and a predicate τ whose intended interpretation is that $\tau(x)$ iff x is conceived. $T\phi$ means that the proposition ϕ is being entertained. (*By whom* is good question: is the point of the argument that for every object there is someone who conceives it? or that everybody thinks about every object?)

   At this point you could, if you like, work out your own natural deduction rules. Here are the rules Priest came up with.

   $$\frac{\phi \rightarrow \psi}{T(\phi) \rightarrow T(\psi)}$$

   which says something to the effect that T distributes over conditionals. Priest calls this "affixing". The other rule is one that tells us that if we conceive an object to have some property ϕ then we conceive it.

   $$\frac{T(\phi(x))}{\tau(x)}$$

   Let us call it the **mixed rule**.

(a) Devise a natural deduction proof of $(\forall x)(\tau(x))$, or of $(\forall x)((\tau(x) \to \bot) \to \bot)$. You are allowed to use the undischarged premiss Tp where p is an arbitrary propositional letter. You may wish to use a natural deduction version of the law of excluded middle. My model answer doesn't, and accordingly i prove only that $(\forall x)((\tau(x) \to \bot) \to \bot)$. You might try to prove $(\exists x)(\tau(x) \to \bot) \to \bot$ as well.

At this point you could, if you like, work out your own sequent calculus rules. Here are the rules i came up with.

$$\frac{\Gamma, A \vdash \Delta, B}{\Gamma, TA \vdash \Delta, TB}$$

and

$$\frac{\Gamma \vdash \Delta, T(\phi(x))}{\Gamma \vdash \Delta, \tau(x)}$$

(b) Prove the sequent $Tp \vdash (\forall x)(\tau(x))$

(c) Prove that a premiss of the form Tp really is needed.

1993:3:3

1. $\{\neg P(x), Q(x)\}$
2. $\{\neg P(x), \neg Q(x), P(fx)\}$.
3. $\{P(b)\}$
4. $\{\neg P(f^4x)\}$.
   1 and 2 resolve to give
5. $\{\neg P(x), P(fx)\}$. First reletter this to get
6. $\{\neg P(w), P(fw)\}$
   Resolve 5 and 6 by unifying $w \to fx$, cut against $P(fx)$ to get
7. $\{\neg P(x), P(f^2x)\}$. First reletter this to get
8. $\{\neg P(w), P(f^2w)\}$
   Resolve 7 and 8 by unifying $w \to f^2x$ cut against $P(f^2x)$ to get
9. $\{\neg P(x), P(f^4x)\}$. Resolve with 3 to get
10. $\{P(f^4b)\}$. Resolve with 4 to get the empty clause
11. $\{\}$.

That's the clever way to do it. I think what PROLOG does is something more like this. It cuts 4 against 2 to get $\{\neg P(f^3x), \neg Q(f^3x)\}$ and cuts against 1 to get $\{\neg P(f^3x)\}$.

Then repeat until you get $\{\neg P(x)\}$ which you can cut against 3. The point is that at each stage PROLOG only ever cuts the current goal clause against something it was given to start with. That way it has only a linear search for a cut at each stage instead of a quadratic one. I'm not sure what sort of relettering PROLOG does, and whether it can make copies of clauses, and reletter one to cut against the other as above. It certainly only ever does linear resolution.

(a) How long does it take?
(b)
(c)
(d)

$$\neg[(\forall y \exists x)\neg(p(x,y) \longleftrightarrow \neg(\exists z)(p(x,z) \wedge p(z,x)))]$$

Rewrite to get rid of the biconditional

$$\neg[(\forall y \exists x)\neg(p(x,y) \vee \neg(\exists z)(p(x,z) \wedge p(z,x)) \wedge p(x,y) \vee \neg(\exists z)(p(x,z) \wedge p(z,x))]$$

push in $\neg$

$$[(\exists y \forall x)\neg(p(x,y) \vee \neg(\exists z)(p(x,z) \wedge p(z,x)) \wedge p(x,y) \vee \neg(\exists z)(p(x,z) \wedge p(z,x))]$$

1996:5:10

$$(\forall z)(\exists x)(\forall y)((P(y) \to Q(z)) \to (P(x) \to Q(x)))$$

Given that the decision problem for first-order logic is undecidable, you haven't much chance of finding a proof of something or a convincing refutation of it unless you postpone work on it until you have a feel for what it is saying.

First we notice that as long as there is an x s.t. $Q(x)$ we can take that element to be a witness to the '$\exists x$' no matter what z is. This is because the truth of '$Q(x)$' ensures the truth of the whole conditional. On the other hand even if *nothing* is Q we are still OK as long as nothing is P—because the falsity of '$P(x)$' ensures the truth of the consequent of the main conditional. There remains the case where $(\forall x)(\neg Q(x))$ and $(\exists x)(P(x))$. But it's easy to check that in that case the whole conditional comes out true too.

So we can approach the search for a sequent calculus proof confident that there is one to be found.

Clearly the only thing we can do with

$$(\forall z)(\exists x)(\forall y)((P(y) \to Q(z)) \to (P(x) \to Q(x)))$$

is a $\forall$-R getting

$$\vdash (\exists x)(\forall y)((P(y) \to Q(a)) \to (P(x) \to Q(x)))$$

(I have relettered 'z' to 'a' for no particular reason). We could have got this by $\exists$-R by replacing 'a' by 'x' so that it came from

$$\vdash (\forall y)((P(y) \to Q(a)) \to (P(a) \to Q(a)))$$

but this doesn't appear to be valid. So we presumably have to keep an extra copy of '$\vdash (\exists x)(\forall y)((P(y) \to Q(a)) \to (P(x) \to Q(x)))$' and we got it from

$$\vdash (\exists x)(\forall y)((P(y) \to Q(a)) \to (P(x) \to Q(x))), \quad (\forall y)((P(y) \to Q(a)) \to (P(a) \to Q(a)))$$

which came by $\forall$-R from

$$\vdash (\exists x)(\forall y)((P(y) \to Q(a)) \to (P(x) \to Q(x))), \quad ((P(b) \to Q(a)) \to (P(a) \to Q(a)))$$

This obviously came from an $\exists$-R:

$$\vdash (\forall y)((P(y) \to Q(a)) \to (P(b) \to Q(b))), \quad ((P(b) \to Q(a)) \to (P(a) \to Q(a)))$$

. . . where i'm assuming the 'x' came from the 'b' we've already seen.

and this must've come from a ∀-R with a new variable:

$$\vdash (P(c) \to Q(a)) \to (P(b) \to Q(b))), \quad ((P(b) \to Q(a)) \to (P(a) \to Q(a)))$$

and now we've got all the quantifiers out of the way and have only the propositional rules to worry about: pretty straightforward from here. Four applications of →-R take us to

$$P(c) \to Q(a),\ P(b) \to Q(a),\ P(b),\ P(a) \vdash Q(b),\ Q(a)$$

and if we break up the '$P(b) \to Q(a)$' on the left we get the two initial sequents:

$$P(c) \to Q(a),\ P(b),\ P(a),\ \underline{Q(a)} \vdash Q(b),\ Q(a)$$

and

$$P(c) \to Q(a),\ P(b),\ P(a) \vdash \underline{P(b)},\ Q(b),\ Q(a)$$

. . . where i have underlined the two formulæ that get glued together by the →-L rule.

1996:6:10

Davis-Putnam: This procedure has three main steps:

1. Delete tautological clauses;
2. Delete unit clauses $\{A\}$ and remove $\neg A$ from all clauses. This is safe because a unit clause $\{A\}$ can be satisfied only if $A \mapsto$ **true** and once that is done A does not need to be considered further.
3. Delete any formula containing pure literals. (If a literal appears always positively or always negatively we can send it to **true** or to **false** without compromising later efforts to find an interpretation of the formula).

If a point is reached where none of the rules above can be applied, a variable is selected arbitrarily for a **case split** and the proof proceeds along both resulting clause sets. We will be happy if *either* resolves to the empty clause. This algorithm terminates because each case split removes a literal.

In this example, we have no tautological clauses or pure literals, so we start with a case split, arbitrarily selecting P to split. If P is true, our clauses are $\{R\}, \{\neg R\}$. We delete unit clause $\{R\}$, and then delete $\neg R$ from all clauses; we are left with the empty clause, which constitutes a refutation of the clause set (the empty disjunction), so the formula is valid. The P false case proceeds similarly, with Q for R. Resolution: There is only one rule of inference in resolution:

$$\frac{\{B, A\}\{\neg B, C\}}{\{A, C\}}$$

The algorithm terminates because as soon as a point is reached where the rule cannot be applied, the clause set is established as satisfiable. Repeatedly applying this rule to the given clause set:

$$\frac{\{\neg P, R\}\{P, \neg Q\}}{\{R, \neg Q\}}$$

$$\frac{\{\neg P, \neg R\}\{P, Q\}}{\{\neg R, Q\}}$$

$$\frac{\{R, \neg Q\}\{\neg R, \neg Q\}}{\square}$$

The empty clause ($\square$) is a contradiction: we have refuted the clause set and so proved the original formula.

Part of an answer to another question

Let A^* represent the formula A, converted into polynomial representation. First we note that in arithmetic mod 2, $x^2 \equiv x$, as $0^2 = 0 \equiv 0$ and $1^2 = 1 \equiv 1$, and all integers are congruent to 0 or 1 modulo 2. Now $(\neg A)^*$ is $1 + A^*$, $(A \wedge B)^*$ is $A^* \cdot B^*$, $(A \vee B)^*$ becomes $A^* + B^* + A^*B^*$, $A \to B$ is $1 + A^* + A^*B^*$, and $A \leftrightarrow B$ is $((1 - A^*) + B^*) \cdot ((1 - B^*) + A^*)$, which simplifies to $1 + 2A^*B^* - A^* - B^*$ and thence to $1 + A^* + B^*$. Recursively applying these rules to any formula will convert it to equivalent polynomial form. $(A \wedge B) \leftrightarrow (B \wedge A)$ translates into $1 + 2(A^*B^*)^2 - A^*B^* - B^*A^* = 1$, hence the formula is a tautology. $A \leftrightarrow A$ translates into 1. $1 \leftrightarrow A$ translates into $1 + 2A^* - A^* - 1 = A^*$. So if we adopt the notation $(A \leftrightarrow A)^n$, to represent formulae of the given type where $\leftrightarrow$ appears n times, we get: $(A \leftrightarrow A)^n = 1$ (n odd) or A (n even), $n \geq 0$. This works for $n = 0$, which is just the formula A.

1998:6:10

Clause 1 tells us that if x pees on itself it pees on a. Clause 2 tells us that if x does *not* pee on itself then it pees on fa. This drops a broad hint that perhaps a is $\{x : x \in x\}$ and fa is $\{x : x \notin x\}$. Clause 3 tells us that nothing pees on both a and fa which is starting to look good. Now ask whether or not $P(fa, fa)$? Well, $P(fa, fa) \rightarrow P(fa, a)$ by clause 1. Then use clause 3 to infer $\neg P(fa, fa)$ whence $\neg P(fa, fa)$. But then clause 2 tells us that $P(fa, fa)$ after all.

The final part. Three clauses:

$$\{\neg P(x, x), P(x, a)\}, \{P(x, x), \neg P(x, f(a))\}\{\neg P(y, f(x)), \neg P(y, x)\}$$

I think this is Russell's paradox. $P(x, y)$ is $x \in y$; a is the complement of the Russell class, and f is complementation.

In the third clause make the substitutions a/x and $f(a)/y$ to get $\{\neg P(f(a), f(a)), \neg P(f(a), a)\}$

In the first clause make the substitution $f(a)/x$ to get $\{\neg P(f(a), f(a)), P(f(a), a)\}$ and resolve on $P(f(a), a)$ to get $\{\neg P(f(a), f(a))\}$.

In the second clause make the substitution $f(a)/x$ to get $\{\neg P(f(a), f(a)), P(f(a), f(a))\}$ which is of course $\{\neg P(f(a), f(a)), P(f(a), f(a))\}$ resolves with the current goal clause to get

damn

Some ML code for unification

```
let apply_subst l t = rev_itlist (\pair term.subst[pair]term) l t;;

% Find a substitution to unify two terms (lambda-terms not dealt with) %

letrec find_unifying_subst t1 t2 =
 if t1=t2
  then []
 if is_var t1
  then if not(mem t1 (frees t2)) then [t2,t1] else fail
 if is_var t2
  then if not(mem t2 (frees t1)) then [t1,t2] else fail
 if is_comb t1 & is_comb t2
  then
   (let rat1,rnd1 = dest_comb t1
    and rat2,rnd2 = dest_comb t2
    in
    let s = find_unifying_subst rat1 rat2
    in s@find_unifying_subst(apply_subst s rnd1)(apply_subst s rnd2)
   )else fail;;
```

This language (HOL) curries n-place predicates. This corresponds to a determination—when unifying (for example)—'$f(a, b, f(x))$' with '$f(x, y, w)$'—to detect $x \mapsto a$ and then do that to the third argument of the first occurrence of 'f' so that it becomes '$f(a)$' before we get there. This finesses questions about simultaneous vs consecutive execution of substitution. It might be a good idea (or perhaps a very bad one!) to think about unification in $L_{\omega_1,\omega}$. How does it work?

An illustration

In the two axioms.

1. $(\forall xy)(x > y \rightarrow Sx > Sy)$
2. $(\forall w)(Sw > 0)$

'S' is the successor function: $S(x) = x + 1$. (Remember that $\mathbb{N}$ is the recursive datatype built up from 0 by means of the successor function.)

Now suppose we want to use PROLOG-style proof with resolution and unification to find a z such that $z > S0$. We turn 1 and 2 into clauses getting $\{\neg(x > y), Sx > Sy\}$ and $\{Sw > 0\}$, and the (negated) goal clause $\{\neg(z > S0)\}$.

The idea now is to refute this negated goal clause. Of course we can't refute it, beco's there are indeed some z of which this clause holds, but we might be able to refute some instances of it, and this is where unification comes in.

$z > S0$ will unify with $Sx > Sy$ generating the bindings $z \mapsto Sx$ and $y \mapsto 0$. We apply these bindings to the two clauses clauses $\{\neg(x > y), Sx > Sy\}$ and $\{\neg(z > S0)\}$, obtaining $\{\neg(x > S0), Sx > S0\}$ and $\{\neg(Sx > S0)\}$. These two resolve to give $\{\neg(x > 0)\}$. Clearly the substitution $x \mapsto Sw$ will enable us to resolve $\{\neg(x > 0)\}$ (which has become $\{\neg(Sw > 0)\}$) with $\{Sw > 0\}$ to resolve to give the empty clause. *En route* we have generated the bindings $z \mapsto Sx$ and $x \mapsto Sw$, which compose to give $z \mapsto SSw$, which tells us that the successor of the successor of any number is bigger than the successor of 0 as desired.

The idea is this: We are trying to find a witness to $(\exists x)(A(x))$. Assume the negation of this, and try to refute it. In the course of refuting it we generate bindings that tell us what the witnesses are.

Higher-order Unification

Unification in first-order logic is well-behaved. For any two complex terms t_1 and t_2 if there is any unifier at all there is a most general unifier which is unique up to relettering. This doesn't hold for higher-order logic where there are function variables. It's pretty clear what you have to do if you want to unify $f(3)$ and 6: you replace f by something like

`if` $x = 3$ `then` 6 `else don't care`

(which one might perhaps write $(\epsilon f)(f(3) = 6)$).

However what happens if you are trying to unify $f(3)$ and $g(6)$? You want to bind 'f' to

`if` $x = 3$ `then` $g(6)$ `else don't care` (A)

but then you also want to bind 'g' to

`if` $x = 6$ `then` $f(3)$ `else don't care` (B)

and you have a vicious loop of substitutions. There are restricted versions that work, and there was even a product called `Q-PROLOG` ('Q' for Queensland) that did something clever. I've long ago forgotten.

I find in my notes various ways of coping with this, one using ϵ terms. One can have an epsilon term which is is a pair of things satisfying (A) and (B):

$$(\epsilon p)(\exists h_1, h_2)(p = \langle h_1, h_2\rangle \wedge h_1(3) = h_2(6))$$

so that we bind 'f' to '`fst`(p)' and 'g' to '`snd`(p)'.

Question 63

Question 633a

The natural deduction proof i favour looks like this:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[p]\quad[\tau(x)\to\bot]}{p\wedge(\tau(x)\to\bot)}}{(\tau(x)\to\bot)}}{p\to(\tau(x)\to\bot)}}{Tp\to T(\tau(x)\to\bot)^{a}}\qquad\dfrac{}{Tp}}{T(\tau(x)\to\bot)}}{\tau(x)^{m}}\qquad\dfrac{}{[(\tau(x)\to\bot)]}$$
$$\dfrac{\dfrac{\dfrac{\bot}{(\tau(x)\to\bot)\to\bot}}{(\forall x)((\tau(x)\to\bot)\to\bot)}}{}$$

. . . where m superscript betrays a use of the mixed rule; a a use of the affixing rule. Cancelled assumptions are enclosed in [square brackets] as usual. (There may be other proofs as well.)

633b

The shortest sequent calculus proof i can find is the following.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p,\tau(x)\vdash\tau(x)}{p\vdash\tau(x),\neg\tau(x)}}{Tp,\vdash\tau(x),T(\neg\tau(x))}}{Tp,\vdash\tau(x),\tau(x)}}{Tp\vdash\tau(x)}}{Tp\vdash\forall x\tau(x)}$$

633c

Prove that a premiss of the form Tp really is needed.

Notice that since neither the affixing rule nor the mixed rule have anything like $T\phi$ as a conclusion, we can obtain models of this calculus in which Tp is always false (The modal logicians express this by saying that T is a *falsum* operator) and $\tau(x)$ is always false. Accordingly we cannot expect to be able to prove that even one thing is τ without some extra premisses.

61

Use resolution to prove that a graph and its complement cannot both be disconnected.

Robert Thatcher's model answer (edited by me)

Suppose G is a graph such that G and $\overline{G}$ are both disconnected. We will derive a contradiction by resolution.

If G is disconnected then there are vertices a and b which are not connected in G. If $\overline{G}$ is disconnected then there are vertices c and d which are disconnected in $\overline{G}$.

Let us have six propositional letters: ab, ac, ad, bc, bd, cd. The intended interpretation is that ab (for example) means that the edge ab belongs to the edge set of G.

First consider G. The first thing we know is that the edge ab is **not** in the edge set of G, hence we have the clause $\neg ab$. Since we know that a and b are disconnected in G, we cannot allow any indirect paths from a to b. There are two possible lengths of indirect path involving 1 or 2 indirect vertices. (It will turn out that we can get our desired contradiction without considering indirect paths that are longer, but we don't know that yet, and are just hoping for the best!) This tells us that $(\neg ac \vee \neg bc)$, or, in resolution jargon, $\{\neg ac, \neg bc\}$. Similarly we may add $\{\neg ad, \neg bd\}$. The paths involving 2 vertices are $acdb$ and $adcb$. We already know that the path cd must be present (since it cannot be in $\overline{G}$) hence we may add the clauses $\{\neg ac, \neg bd\}$ and $\{\neg ad, \neg cb\}$.

Now consider $\overline{G}$. This cannot have cd, so we can add $\{cd\}$ (this is not negated, since we are now considering edges that are not in $\overline{G}$, and hence must be in G). Similarly, we cannot have any indirect connections from c to d, so we cannot have the paths cad, cbd, $cabd$ and $cbad$. Since we know that ab cannot be in the graph, we can write these as the clauses: $\{ac, ad\}$, $\{bc, bd\}$, $\{ac, bd\}$ and $\{bc, ad\}$. Note that the last two do not contain ab since we know that we **must** have $\neg ab$ by choice of a and b.

So now we have a set of clauses representing the conditions that need to be satisfied if both G and $\overline{G}$ are to be disconnected. To recapitulate, these are:

$$\{\neg ab\}\ \{\neg ac, \neg bc\}\ \{\neg ad, \neg bd\}\ \{\neg ad, \neg bc\}\ \{\neg ac, \neg bd\}$$
$$\{cd\}\ \{ac, ad\}\ \{bc, bd\}\ \{ac, bd\}\ \{ad, bc\}$$

Now we may combine these clauses (carefully) using resolution – the choice of clauses to resolve is crucial, since it is very easy to end up with many useless clauses of the form $\{A, \neg A\}$.

$$\dfrac{\dfrac{\{\neg ac, \neg bc\} \qquad \{bc, bd\}}{\{\neg ac, bd\}} \qquad \{\neg ac, \neg bd\}}{\{\neg ac\}} \tag{6.1}$$

$$\dfrac{\dfrac{\{\neg ad, \neg bd\} \qquad \{bd, bc\}}{\{\neg ad, bc\}} \qquad \{\neg ad, \neg bc\}}{\{\neg ad\}} \tag{6.2}$$

Now we have two literal clauses, we can use them to derive a contradiction:

$$\dfrac{\dfrac{\{ac, ad\} \qquad \{\neg ac\}}{\{ad\}} \qquad \{\neg ad\}}{\bot} \tag{6.3}$$

We have derived the empty clause.

Note that only six of the original 10 clauses were required for the resolution – partly because the two simple clauses are disjoint from the remaining eight (due to the way we initially wrote down the clauses – in a sense they have already been used in a resolution). The remaining two clauses are superfluous: they do not provide any information beyond the original eight. The clauses found when considering the 3-stage path in $\overline{G}$ are complementary to those found when considering similar paths in G, and hence if resolved in any way with them, give useless clauses of the form $\{A, \neg A\}$.

6.2 Answers to Larry's exercises

$P(a), P(b) \vdash P(a), P(b), P(a), P(b)$
$P(a), P(b) \vdash P(a) \wedge P(b), P(a) \wedge P(b)$
$\vdash P(a) \rightarrow P(a) \wedge P(b), P(b) \rightarrow P(a) \wedge P(b)$ ($\wedge$-R twice)
$\vdash (\exists z)(P(z) \rightarrow P(a) \wedge P(b)), (\exists z)(P(z) \rightarrow P(a) \wedge P(b))$ $\exists$-R twice
$\vdash (\exists z)(P(z) \rightarrow P(a) \wedge P(b))$ (contraction)
(This needs to be properly set using the stylefile bussproofs....
Most of these answer are by Dave Tonge. Not all, and some of his have been mutilated by me.

$\star$ **1** $\star$ *Is the formula $A \rightarrow \neg A$ satisfiable? Is it valid?*
The case where A is false satisfies. The case where A is true does not satisfy. Therefore the expression is satisfiable but not valid.

$\star$ **3** $\star$ *Work out the details above.*
Negate and convert $(A_1 \wedge \cdots \wedge A_k) \rightarrow B$ to CNF
Negate to give $\neg((A_1 \wedge \cdots \wedge A_k) \rightarrow B)$
Eliminate $\rightarrow$ to give $\neg(\neg(A_1 \wedge \cdots \wedge A_k) \vee B)$
Push in negations $(A_1 \wedge \cdots \wedge A_k) \wedge \neg B$
Remove parentheses to give $A_1 \wedge \cdots \wedge A_k \wedge \neg B$

Convert $M \rightarrow K \wedge P$ to clausal form.
Split into two formulae, $M \rightarrow K$ and $M \rightarrow P$.
Eliminate $\rightarrow$s to give $\neg M \vee K$ and $\neg M \vee P$.
Convert to clauses $\{\neg M, K\}$ and $\{\neg M, P\}$.

$\star$ **5** $\star$ *Write down a formula that is true in every domain that contains at least m elements. Write down a formula that is true in every domain that contains at most most m elements.*
At least m:

$$(\exists x_1 \ldots x_m)(\bigwedge_{k \neq j} (a_j \neq a_k)^1$$

An answer for the next is obviously obtainable by increasing m by one and negating!
At most m:

$$(\forall x_1 \ldots x_{m+1})(\bigvee_{i \neq j < m} x_j = x_i)$$

Many readers find the following more natural

$$(\exists x_1 \ldots x_m)(\forall y)(\bigvee_{1 \leq i \leq m} y = x_i)$$

This formula is logically more complicated (it has an alternation of quantifiers) but is shorter.
A brief question to ask yourself: how rapidly does the formula grow with n?

$\star$ **6** $\star$ *Verify these equivalences by appealing to the truth definition for first order logic.*
There are too many of these, so I'll just do the infinitary de Morgan law $\mathcal{M}_V \models \neg((\forall x)A) = ((\exists x)\neg A)$.

[1] The temptation to write this as: $(\exists a_1 \ldots a_m)(\forall j, k < m)(k \neq j \rightarrow a_j \neq a_k)$ must be resisted. This is *not* correct, since the subscripts on the variables are not themselves variables and cannot be bound.

To show this we have to show that $\mathcal{M}_V \models \neg((\forall x)A)$ is equivalent to $\mathcal{M}_V \models ((\exists x)\neg A)$.

The first half becomes: for all $m \in M$ such that $\mathcal{M}_{V\{m/x\}} \models A$ does not hold. The second half becomes there exists an $m \in M$ for which $\mathcal{M}_{V\{m/x\}} \models A$ does not hold.

These two are plainly equivalent for if the first one does not hold then there there will not exist an m for which the second holds. Similarly, if the second is true then the first will not hold for all ms (for it won't hold for the m given by the first).

⋆ **7** ⋆ *Explain why the following are not equivalences. Are they implications? In which direction?*

$((\forall x)A) \vee ((\forall x)B)? = ?(\forall x)(A \vee B)$

$((\exists x)A) \wedge ((\exists x)B)? = ?(\exists x)(A \wedge B)$

First one: The RHS could be true if A were true and B were false for a particular x. Thus, B would not be true for all x. There might be another x for which A were false and B true. Thus A is not true for all x. Thus the LHS can be false although the RHS is true. Because this case exists the two statements are not equivalent. However, there is a left-to-right implication.

Second: The x for which A might not be the same x for which B. Therefore the RHS will be false in this case. The two statements are note equivalent. However, the RHS implies the LHS.

⋆ **9** ⋆ *Verify that* $\circ$ *is associative and has* [] *for an identity.*

To show associativity we need to show that $(\phi \circ \theta) \circ \sigma = \phi \circ (\phi \circ \theta)$.

If we consider ϕ, θ and σ as functions $f(x)$, $g(x)$ and $h(x)$ which map literals to their substituted values then we get the composition

$$\begin{aligned} \lambda x.((f \circ g) \circ h) &= \lambda x.(f \circ (g \circ h)) \\ \lambda x.((\lambda y.f(g(x)))h) &= \lambda x.(f(g \circ h)) \\ \lambda x.(f(g(h(x)))) &= \lambda x.f(g(h(x)))p \end{aligned}$$

Which says that they are the same. This relies on our functions returning the literal given as an argument in cases where no substitution has been defined.

To show that [] is the identity we need to consider it as a function g which maps all the argument literals to themselves, without substitution. ϕ remains the function f as before.

$$\begin{aligned} f \circ g &= f \\ g(f) &= f \\ f &= f \end{aligned}$$

⋆ **11** ⋆ *Each of the following formulae is satisfiable but not valid. Exhibit a truth assignment that makes the formula true and another truth assignment that makes the formula false.*

$P \rightarrow Q$

True for $P = true$ and $Q = true$. False for $P = true$ and $Q = false$.

$P \vee Q \rightarrow P \wedge Q$

True for $P = Q = true$. False for $P = true$ and $Q = false$.

$\neg(P \vee Q \vee R)$

True for $P = Q = R = false$. False otherwise.

$\neg(P \wedge Q) \wedge \neg(Q \vee R) \wedge (P \vee R)$

True for $P = true$ and $Q = R = false$. False for $P = Q = R = true$.

⋆ **12** ⋆ *Convert of the following propositional formulæ into Conjunctive Normal Form and also into Disjunctive Normal Form. For each formula, state whether it is valid, satisfiable, or unsatisfiable; justify each answer.*
$(P \to Q) \wedge (Q \to P)$

To obtain CNF we first eliminate $\to$ to get $(\neg P \vee Q) \wedge (\neg Q \vee P)$.

To obtain DNF we first eliminate $\to$ to get $(\neg P \vee Q) \wedge (\neg Q \vee P)$. Push in conjunctions to get $(\neg P \wedge (\neg Q \vee P)) \vee (Q \wedge (\neg Q \vee P))$. And again to get $(\neg P \wedge P) \vee (\neg P \wedge \neg Q) \vee (Q \wedge \neg Q) \vee (Q \wedge P)$. Remove those which are obviously false to get $(\neg P \wedge \neg Q) \vee (P \wedge Q)$.

This formula is satisfiable—it is satisfied when $P = Q$.

$$((P \wedge Q) \vee R) \wedge (\neg((P \vee R) \wedge (Q \vee R)))$$

Both CNF and DNF require one to push in negations to get

$$((P \wedge Q) \vee R) \wedge (\neg(P \vee R) \vee \neg(Q \vee R))$$

and then

$$((P \wedge Q) \vee R) \wedge ((\neg P \wedge \neg R) \vee (\neg Q \wedge \neg R))$$

.

TO get CNF push in disjunctions to get

$$(P \vee R) \wedge (Q \vee R) \wedge (\neg P \vee \neg Q) \wedge (\neg P \vee \neg R) \wedge (\neg R \vee \neg Q) \wedge (\neg R \vee \neg R)$$

which is

$$(P \vee R) \wedge (Q \vee R) \wedge (\neg P \vee \neg Q) \wedge (\neg R \vee \neg Q) \wedge \neg P \wedge \neg R$$

.

To get DNF push in conjunctions to get $(P \wedge Q \wedge \neg R \wedge \neg R) \vee (P \wedge Q \wedge \neg Q \wedge \neg R) \vee (R \wedge \neg P \wedge \neg R) \vee (R \wedge \neg Q \wedge \neg R)$.

The formula is unsatisfiable—if you look at it in DNF each conjunct has an atom in both negated and unnegated form so all conjuncts must be false so the whole disjunction is always false.

$\neg(P \vee Q \vee R) \vee ((P \wedge Q) \vee R)$

Both CNF and DNF require one to push in negations to get $(\neg P \wedge \neg Q \wedge \neg R) \vee ((P \wedge Q) \vee R)$.

To get CNF we need to push in disjunctions to get $(\neg P \wedge \neg Q \wedge \neg R) \vee ((P \vee R) \wedge (Q \vee R))$ then $((\neg P \wedge \neg Q \wedge \neg R) \vee (P \vee R)) \wedge ((\neg P \wedge \neg Q \wedge \neg R) \vee (Q \vee R))$ and then $(\neg P \vee P \vee R)) \wedge (\neg Q \vee P \vee R) \wedge (\neg R \vee P \vee R) \wedge (\neg P \vee Q \vee R) \wedge (\neg Q \vee Q \vee R) \wedge (\neg R \vee Q \vee R)$ which might as well be $(\neg Q \vee P \vee R) \wedge (\neg P \vee Q \vee R)$.

To get DNF we don't have to do much except expand brackets to $(\neg P \wedge \neg Q \wedge \neg R) \vee (P \wedge Q) \vee R$.

This is satisfiable—it is only false for $P = R = false$, $Q = true$ and $P = true$, $Q = R = false$.

$\neg(P \vee Q \to R) \wedge (P \to R) \wedge (Q \to R)$

Both CNF and DNF need one to get rid of $\to$s to give $\neg(\neg(P \vee Q) \vee R) \wedge (\neg P \vee R) \wedge (\neg Q \vee R)$. Push in negations to get $((P \vee Q) \wedge \neg R) \wedge (\neg P \vee R) \wedge (\neg Q \vee R)$.

We would appear to have the CNF already—$(P \vee Q \vee \neg R) \wedge (\neg P \vee R) \wedge (\neg Q \vee R)$.

To get DNF we ned to push in conjunctions to get $(P \vee Q \vee \neg R) \wedge (\neg P \vee \neg Q) \wedge (\neg P \vee R) \wedge (R \vee \neg Q) \wedge R$. Again to give $(P \vee Q \vee R) \wedge (P \vee R) \wedge (\neg P \vee Q \vee R) \wedge (Q \vee R) \wedge (\neg P \vee \neg Q \vee \neg R)$.

This is satisfiable - for example in the case $P = Q = false, R = true$ but it can be false as it is when $P = Q = R = false$.

⋆ **14** ⋆ *Prove* $(P \wedge Q \to R) \wedge (P \vee Q \vee R) \to ((P \leftrightarrow Q) \to R)$ *and* $((P \to Q) \to P) \to P$ *by resolution. Show the steps of converting the formula into clauses.*

We have to negate and remove →s first.

$$\neg((P \wedge Q \to R) \wedge (P \vee Q \vee R) \to ((P \leftrightarrow Q) \to R))$$
$$\neg(\neg((\neg(P \wedge Q) \vee R) \wedge (P \vee Q \vee R)) \vee (\neg((\neg P \vee Q) \wedge (\neg Q \vee P)) \vee R))$$
$$((\neg P \vee \neg Q \vee R) \wedge (P \vee Q \vee R)) \wedge \neg((P \wedge \neg Q) \vee (\neg P \wedge Q) \vee R)$$
$$(\neg P \vee \neg Q \vee R) \wedge (P \vee Q \vee R) \wedge (\neg(P \wedge \neg Q) \wedge \neg(\neg P \wedge Q) \wedge \neg R)$$
$$(\neg P \vee \neg Q \vee R) \wedge (P \vee Q \vee R) \wedge (\neg P \vee Q) \wedge (P \vee \neg Q) \wedge \neg R$$

This gives us the clauses $\{\neg P, \neg Q, R\}$, $\{P, Q, R\}$, $\{\neg P, Q\}$, $\{\neg Q, P\}$, $\{\neg R\}$.

If we resolve the last one with the first two we get $\{\neg P, \neg Q\}$ and $\{P, Q\}$. This gives us all four possible clauses starring P and Q so we will certainly get the empty clause. ($\{\}$). We have assumed the negation of the theorem and derived a contradiction. This proves the theorem.

We have to negate and eliminate →s first.

$$\neg(\neg(\neg(\neg P \vee Q) \vee P) \vee P)$$
$$\neg(\neg((P \wedge \neg Q) \vee P) \vee P)$$
$$\neg((\neg(P \wedge \neg Q) \wedge \neg P) \vee P)$$
$$\neg(((\neg P \vee Q) \wedge \neg P) \vee P)$$
$$(\neg((\neg P \vee Q) \wedge \neg P) \wedge \neg P)$$
$$((\neg(\neg P \vee Q) \wedge P) \wedge \neg P)$$
$$(((P \wedge \neg Q) \wedge P) \wedge \neg P)$$
$$P \wedge P \wedge \neg P \wedge \neg Q$$
$$P \wedge \neg P \wedge \neg Q$$

This gives us the clauses $\{P\}$, $\{\neg P\}$ and $\{\neg Q\}$. Resolving $\{P\}$ with $\{\neg P\}$ gives a contradiction ($\{\}$). We have assumed the negation of Peirce's law and derived a contradiction, thus proving the law.

⋆ **15** ⋆ *Using linear resolution, prove that* $(P \wedge Q) \to (R \wedge S)$ *follows from* $(P \to R) \wedge (Q \to S)$ *and* $R \wedge P \to S$.

The two assumed formulæ and the negated conclusion give us the clauses $\{\neg P, R\}$, $\{\neg Q, S\}$ and $\{\neg R, \neg P, S\}$. We need to resolve these with the clauses given by the negation of the formula we are trying to prove. These clauses are $\{P\}$, $\{Q\}$ and $\{\neg R, \neg S\}$.

Take $\{\neg R, \neg S\}$ and resolve with $\{\neg R, \neg P, S\}$ to get $\{\neg R, \neg P\}$.

Resolve the result with $\{\neg P, R\}$ to get $\{\neg P\}$.

Resolve the result with $\{P\}$ to get a contradiction ($\{\}$). This proves the formula.

⋆ **16** ⋆ *Convert these axioms to clauses, showing all steps. Then prove* $Winterstorm \rightarrow Miserable$ *by resolution:* $Rain \wedge (Windy \vee \neg Umbrella) \rightarrow Wet$, $Winterstorm \rightarrow Storm \wedge Cold$, $Wet \wedge Cold \rightarrow Miserable$ *and* $Storm \rightarrow Rain \wedge Windy$.

First we need to construct all our definite clauses.

$$
\begin{array}{rl}
 & (Rain \wedge (Windy \vee \neg Umbrella) \wedge Wet \\
expand\wedge & ((Rain \wedge Windy) \vee (Rain \wedge \neg Umbrella)) \rightarrow Wet \\
remove \rightarrow & \neg((Rain \wedge Windy) \vee (Rain \wedge \neg Umbrella)) \vee Wet \\
 & (\neg(Rain \wedge Windy) \wedge \neg(Rain \wedge \neg Umbrella)) \vee Wet \\
 & ((\neg Rain \vee \neg Windy) \wedge (\neg Rain \vee \neg Umbrella)) \vee Wet \\
 & (\neg Rain \vee \neg Windy \vee Wet) \wedge (\neg Rain \vee \neg Umbrella \vee Wet) \\
clauses\ are & \{\neg Rain, \neg Windy, Wet\}\ and\ \{\neg Rain, \neg Umbrella, Wet\}
\end{array}
$$

$$
\begin{array}{rl}
 & Winterstorm \rightarrow Storm \wedge Cold \\
remove \rightarrow & \neg Winterstorm \vee (Storm \wedge Cold) \\
expand\wedge & (\neg Winterstorm \wedge Storm) \wedge (\neg Winterstorm \wedge Cold) \\
clauses\ are & \{\neg Winterstorm, Storm\} and \{\neg Winterstorm, Cold\}
\end{array}
$$

$$
\begin{array}{rl}
 & Wet \wedge Cold \rightarrow Miserable \\
remove \rightarrow & \neg(Wet \wedge Cold) \vee Miserable \\
push\ in\ \neg & \neg Wet \vee \neg Cold \vee Miserable \\
clauses\ are & \{\neg Wet, \neg Cold, Miserable\}
\end{array}
$$

$$
\begin{array}{rl}
 & Storm \rightarrow Rain \wedge Windy \\
remove \rightarrow & \neg Storm \vee (Rain \wedge Windy) \\
expand\wedge & (\neg Storm \vee Rain) \wedge (\neg Storm \vee Windy) \\
clauses\ are & \{\neg Storm, Rain\} and \{\neg Storm, Windy\}
\end{array}
$$

In order to prove $Winterstorm \rightarrow Miserable$ we have to assume its negation and derive a contradiction. So, let's find out the clauses that would give us.

$$
\begin{array}{rl}
 & Winterstorm \rightarrow Miserable \\
negate & \neg(Winterstorm \rightarrow Miserable) \\
remove \rightarrow & \neg(\neg Winterstorm \vee Miserable) \\
push\ in\neg & Winterstorm \wedge \neg Miserable \\
clauses\ are & \{Winterstorm\} and \{\neg Miserable\}
\end{array}
$$

Now, using all the clauses we have gathered we need to use resolution to get a contradiction.

Resolve $\{Winterstorm\}$ with $\{\neg Winterstorm, Storm\}$ to give $\{Storm\}$.
Resolve $\{Winterstorm\}$ with $\{\neg Winterstorm, Cold\}$ to give $\{Cold\}$.
Resolve $\{Storm\}$ with $\{\neg Storm, Windy\}$ to give $\{Windy\}$.
Resolve $\{Storm\}$ with $\{\neg Storm, Rain\}$ to give $\{Rain\}$.
Resolve $\{Rain\}$ with $\{\neg Rain, \neg Windy, Wet\}$ to give $\{\neg Windy, Wet\}$.

Resolve $\{Windy\}$ with $\{\neg Windy, Wet\}$ to give $\{Wet\}$.
Resolve $\{Wet\}$ with $\{\neg Wet, \neg Cold, Miserable\}$ to give $\{\neg Cold, Miserable\}$.
Resolve $\{Cold\}$ with $\{\neg Cold, Miserable\}$ to give $\{Miserable\}$.
Resolve $\{Miserable\}$ with $\{\neg Miserable\}$ to give a contradiction ($\{\}$). This proves the theorem by contradiction of the negated theorem.

⋆ **17** ⋆ *Let $=$ be a 2-place predicate symbol, which we write using infix notation: for instance, $x = y$ rather than $=(x, y)$. Consider the following axioms:*

$$(\forall x) \quad x = x \tag{6.4}$$
$$(\forall xy) \quad (x = y \rightarrow y = x) \tag{6.5}$$
$$(\forall xyz) \quad (x = y \wedge y = z \rightarrow x = z) \tag{6.6}$$

Let the universe be the set of natural numbers, $N = \{0, 1, 2, \cdots\}$. Which axioms hold if the interpretation of $=$ is...
the empty relation, ϕ?
(1) does not hold. (2), (3) hold.
Notice that the empty relation on an empty set **is** *reflexive!!*
the universal relation, $\{(x, y)|x, y \in N\}$?
(1), (2), (3) all hold.
the relation $\{(x, x)|x \in N\}$?
(1), (2), (3) all hold.
the relation $\{(x, y)|x, y \in N \wedge x + y$ is even$\}$?
(1), (2), (3) all hold.
the relation $\{(x, y)|x, y \in N \wedge x + y = 100\}$?
(1), (3) do not hold. (2) holds.
the relation $\{(x, y)|x, y \in N \wedge x = y \,(mod\, 16)\}$?
(1), (2), (3) all hold.

⋆ **18** ⋆ *Taking $\sim$ and R as 2-place relation symbols, consider the following axioms:*

$$(\forall x) \quad \neg R(x, x)$$
$$(\forall xy) \quad \neg(R(x, y) \wedge R(y, x))$$
$$(\forall xyz) \quad (R(x, y) \wedge R(y, z) \rightarrow R(x, z))$$
$$(\forall xy) \quad (R(x, y) \vee x = y \vee R(y, x))$$
$$(\forall xz) \quad (R(x, z) \rightarrow (\exists y)(R(x, y) \wedge R(y, z)))$$

Exhibit two interpretations that satisfy axioms 1-3 and falsify axioms 4 and 5. Exhibit two interpretations that satisfy axioms 1-4 and falsify axiom 5. Exhibit two interpretations that satisfy axioms 1-5. Consider only interpretations that make $=$ denote the equality relation.

1-3 true, 4 and 5 false.
Domain is sets of natural numbers. $\hat{R}$ is $\subseteq$ or $\subset$ (strict subset) or $\supseteq$ or $\supset$ (strict superset).

1-4 true, 5 false.
Domain is natural numbers. $\hat{R}$ is $<$ or $>$ or $\leq$ or $\geq$.

1-5 true.
Domain is real numbers. $\hat{R}$ is $<$ or $>$ or $\leq$ or $\geq$.

⋆ **19** ⋆ *Consider a first-order language with 0 and 1 as constant symbols, with - as a 1-place function symbol and + as a 2-place function symbol, and with $=$ as a 2 place predicate symbol.*

(a) Describe the Herbrand Universe for this language.

$$
\begin{aligned}
\mathcal{C} &= \{0,1\} \\
\mathcal{F}_1 &= \{-\} \\
\mathcal{F}_2 &= \{+\} \\
\mathcal{F}_n(n>2) &= \phi \\
\mathcal{P}_1 &= \phi \\
\mathcal{P}_2 &= \{=\} \\
\mathcal{P}_n(n>2) &= \phi \\
H_0 &= \{0,1\} \\
H_1 &= \{0,1,-(0),-(1)\} \\
H &= \{0,1,-(0),-(1),-(-(0)),-(-(1)),+(0,0),+(0,1),+(1,0), \\
&\quad +(1,1),+(0,-(0)),+(0,-(1)),+(-(0),0),+(-(1),1)\cdots\} \\
HB &= \{=(0,0),=(0,1),=(1,0),=(1,1),=(-(0),-(0)), \\
&\quad =(-(1),-(0)),=(+(0,1),+(-(1),-(0))),\cdots\}
\end{aligned}
$$

For extra brownie points write a Context-Free grammar that generates this set ...

(b) The language can be interpreted by taking the integers for the universe and giving 0, 1, -, + and = their usual meanings over the integers. What do those symbols denote in the corresponding Herbrand interpretation?

In the interpretation = is interpreted by the set of all ordered pairs formed from two expressions α and β such that the result of putting an equals sign between α and β is a theorem of the theory we have in mind. + similarly is the set of all ordered triples of expressions $\langle\alpha,\beta,\gamma\rangle$ such that the result of putting a '+' sign and an '=' sign between them in the obvious way gives an expression that is a theorem of, again, whatever the theory is that we have in mind. Interpretations for the others are defined similarly.

⋆ **20** ⋆ *For each of the following pairs of terms, give a most general unifier or explain why none exists.*

$f(g(x),z)$ and $f(y,h(y))$

$f(g(x),h(g(x)))$ is the most general unifier.

$j(x,y,z)$ and $j(f(y,y),f(z,z),f(a,a))$

$j(f(f(f(a,a),f(a,a)),f((a,a),f(a,a))),f(f(a,a),f(a,a)),f(a,a))$ is the most general unification.

$j(x,z,x)$ and $j(y,f(y),z)$

Any unification requires that $x=y=z$ and that $z=f(y)$ also. Therefore the terms cannot be unified without allowing $f(f(f(\cdots)))$.

$j(f(x),y,a)$ and $j(y,z,z)$

This cannot be unified because it required that $y=z=a$ and also that $y=f(x)$. This will only work if $f(x)=a$ for all x.

$j(g(x),a,y)$ and $j(z,x,f(z,z))$

$j(g(a),a,f(g(a),g(a)))$ is the most general unification.

⋆ **34** ⋆ *Convert these formulæ into clauses, showing each step: negating the formula, eliminating $\rightarrow$ and $\leftrightarrow$, moving the quantifiers, Skolemizing, dropping the universal quantifiers and converting the matrix into CNF.*

$$(\forall x)(\exists y)R(x,y)) \rightarrow ((\exists y)(\forall x)R(x,y))$$

$$
\begin{array}{rl}
\textit{negate and remove} \rightarrow & ((\forall x)(\exists y)R(x,y)) \wedge \neg((\exists y)(\forall x)R(x,y)) \\
\textit{move quantifiers} & ((\forall x)(\exists y)R(x,y)) \wedge (\forall y)(\neg(\forall x)R(x,y)) \\
 & ((\forall x)(\exists y)R(x,y)) \wedge ((\forall y)(\exists x)\neg R(x,y)) \\
\textit{skolemise and clause} & \{R(x,f(x))\},\ \{\neg R(g(x),x)\}
\end{array}
$$

$$
\begin{array}{rl}
 & ((\exists y)(\forall x)R(x,y)) \rightarrow ((\forall x)(\exists y)R(x,y)) \\
\textit{negate and remove} \rightarrow & ((\exists y)(\forall x)R(x,y)) \wedge \neg((\forall x)(\exists y)R(x,y)) \\
 & ((\exists y)(\forall x)R(x,y)) \wedge ((\exists x)(\forall y)\neg R(x,y)) \\
\textit{skolemise and clause} & \{R(x,a)\},\ \{\neg R(b,x)\}
\end{array}
$$

$$
\begin{array}{rl}
 & (\exists x)(\forall yz)((P(y) \rightarrow Q(z)) \rightarrow (P(x) \rightarrow Q(x))) \\
\textit{negate and remove} \rightarrow & \neg(\exists x)(\forall yz)((P(y) \wedge \neg Q(z)) \vee \neg P(x) \vee Q(x)) \\
 & (\forall x)(\exists yz)\neg((P(y) \wedge \neg Q(z)) \vee \neg P(x) \vee Q(x)) \\
 & (\forall x)(\exists yz)((\neg P(y) \vee Q(z)) \wedge P(x) \wedge \neg Q(x)) \\
\textit{skolemise and clause} & \{\neg P(f(x)), Q(g(x))\},\ \{P(x)\},\ \{\neg Q(x)\}
\end{array}
$$

⋆ **22** ⋆ *Find a refutation from the following set of clauses using linear resolution* $\{P(f(x,y)), \neg Q(x), \neg R(y)\}$, $\{\neg P(v)\}$, $\{\neg R(z), Q(g(z))\}$ and $\{R(a)\}$.

Unify 'x' $\mapsto$ '$g(z)$' and cut $\{P(f(x,y)), \neg Q(x), \neg R(y)\}$ against $\{\neg R(z), Q(g(z))\}$ with cut formula $Q(g(z))$ to give $\{P(f(g(z),z)), \neg R(z)\}$. Unify '$z$' $\mapsto$ 'a' and cut $\{R(a)\}$ against $\{P(f(g(z),z)), \neg R(z)\}$ to give $\{P(f(g(a),a))\}$. Unify 'v' $\mapsto$ '$f(g(a),a)$' and cut the result against $\{\neg P(v)\}$ to give a refutation ($\{\}$).

⋆ **37** ⋆ *Find a refutation from the following set of clauses using resolution with factoring.*

$\{\neg P(x,a), \neg P(x,y), \neg P(y,x)\}$, $\{P(x,f(x)), P(x,a)\}$ and $\{P(f(x),x), P(x,a)\}$.

Binding both 'y' and 'x' to 'a' in clause 1 we get (4) $\{\neg P(a,a)\}$ (with factoring). We can resolve (4) with both clauses 2 and 3 (seperately) to get (5) $\{P(a,f(a))\}$ and (6) $\{P(f(a),a)\}$. 5 resolves with 1 (bind 'y' to 'a' and 'x' to '$f(a)$') to give (7) $\{\neg P(f(a),a), \neg P(f(a),a)\}$ which reduces by factoring to $\{\neg P(f(a),a)\}$ and this resolves with (6) to the empty clause.

⋆ **24** ⋆ *Prove the following formulae by resolution, showing all steps of the conversion into clauses. Remember to negate first!*

$(\forall x)(P \vee Q(x)) \rightarrow (P \vee (\forall x)Q(x))$

$$
\begin{array}{rl}
 & (\forall x)(P \vee Q(x)) \rightarrow (P \vee (\forall x)Q(x)) \\
\textit{negate and remove} \rightarrow & \neg(\neg((\forall x)(P \vee Q(x))) \vee (P \vee (\forall x)Q(x))) \\
\textit{move quantifiers} & ((\forall x)(P \vee Q(x))) \wedge \neg((\forall x)(P \vee Q(x))) \\
 & ((\forall x)(P \vee Q(x))) \wedge ((\exists x)\neg(P \vee Q(x))) \\
 & ((\forall x)(P \vee Q(x))) \wedge ((\exists x)(\neg P \wedge \neg Q(x))) \\
\textit{skolemise and clause} & \{P, Q(x)\},\ \{\neg P\}\ \{\neg Q(a)\}
\end{array}
$$

Resolving the three clauses together gives a contradiction. Therefore the negation of the formula is inconsistent. Therefore the formula is proven.

$(\exists xy)(P(x,y) \to (\forall vw)P(v,w))$

$$\begin{array}{rl} & (\exists xy)(P(x,y) \to (\forall vw)P(v,w)) \\ \textit{negate and remove} \ \to & \neg((\exists xy)(\neg P(x,y) \vee (\forall vw)P(v,w)) \\ \textit{move quantifiers} & (\forall xy)(\exists vw)(P(x,y) \wedge \neg P(v,w)) \\ \textit{skolemise and clause} & \{P(x,y)\},\ \{\neg P(f(x,y),g(x,y))\} \end{array}$$

These two clauses resolve to the empty clause when x takes on the value of $f(x,y)$ and y the value of $g(x,y)$. Thus the formula is proven.

$\neg(\exists x)(\forall y)(R(y,x) \leftrightarrow \neg R(y,y))$

$$\begin{array}{rl} & \neg(\exists x)(\forall y)(R(y,x) \leftrightarrow \neg R(y,y)) \\ \textit{negate and remove} \ \leftrightarrow & (\exists x)(\forall y)((\neg R(y,x) \vee \neg R(y,y)) \wedge (R(y,y) \vee R(y,x))) \\ \textit{skolemise and clause} & \{\neg R(x,a), \neg R(x,x)\},\ \{R(x,x), R(x,a)\} \end{array}$$

If we resolve these two clauses together we get a contradiction. Thus formula is proven.

⋆ **25** ⋆ *Dual Skolemisation*

Let $\mathcal{L}$ be a language, and let $\Psi : \mathcal{L} \to \mathcal{L}$ be a map such that, for all formulæ ϕ, $\Psi(\phi)$ is satisfiable iff ϕ is. (Skolemisation is an example). We will now show that the map $\lambda\phi.\neg(\Psi(\neg\phi))$ preserves validity.

ϕ is valid iff

$\neg\phi$ is not satisfiable iff

$\Psi(\neg\phi)$ is not satisfiable iff

$\neg(\Psi(\neg\phi))$ is valid.

Now all we have to check is that if Ψ is skolemisation then $\lambda\phi.\neg(\Psi(\neg\phi))$ is dual skolemisation *à la Larry Paulson*. Take a formula in prenex normal form, as it might be

$$\forall x \exists y \forall z \phi$$

where ϕ is anything without quantifiers. Negate it to get

$$\exists x \forall y \exists z \neg\phi$$

and skolemize to get

$$\neg\phi[(f`y)/z, a/x]$$

and negate again to get

$$\phi[(f`y)/z, a/x]$$

... which is exactly what you would have got if you dual skolemised the formula we started with.

I suppose one might use it in the following circumstances. You want to prove ϕ from Γ. You (i) **dual**-skolemise ϕ and convert to **disjunctive** normal form; (ii) negate Γ and convert to **disjunctive** normal form. Then you write clauses which are of course conjunctions not disjunctions and resolving to the empty clause means you have established the truth of $\neg\Gamma \vee \phi$. Two things to think about: (i) is there a problem about relettering clauses? (ii) Did we really need to dual skolemise?

The way to understand what skolemisation is doing is something like this: skolemisation is supposed to preserve satisfiability not truth. Truth is a property of a formula in an environment, but satisfiability is a property of the formula. Skolemisation is something you do to a formula not to a formula-in-an-environment. (Any logical manipulation on a formula of course, like conversion to CNF is also, strictly something you do to a formula not a formula-in-an-environment, but with CNF you can carry the environment along with you). Corresponding to skolemisation is a function on environments. I suppose AC is an assertion that this function is uniform in some sense or functorial or something. For example, there is a way of turning any interpretation of '$(\exists x)(F(x))$' into an interpretation of '$F(a)$': one adds structure. To go the other way one throws structure away. The forgetful functor?

Postscript: both skolemisation and dual skolemisation are maps from a language into itself that preserve something. In one case satisfiability and in the other case validity. In this context we can anticipate a map used in the complexity theory course. Take a formula in conjunctive normal form (so it's a conjunction of disjunctions) In general the individual conjuncts may have lots of literals in them, and be something like $(p \vee q \vee \neg r \vee s)$. This conjunct has four literals in it. Now consider the result of replacing this conjunction by the (conjunction of the) two conjuncts $(p \vee q \vee t) \wedge (\neg t \vee \neg r \vee s)$, where '$t$' is a new variable not present in the original formula. The new formula is satisfiable iff the original one was. Also (although the new formula has more conjuncts) we have replaced longer conjuncts by shorter conjuncts. You will see later why this is a useful trick.

2002 p6 q11

(a) A is consistent

(b) We have derived a contradiction from something that has been negated. So the thing that had been negated is valid. That thing is the skolemised version of A. So the skolemised version of A is valid. Doesn't seem to tell us anything. Skolemisation preserves satisfiability.

(c) If $\neg A$ is refutable, then it shouldn't matter which vbl you choose for a case split: you should get the empty clause every time. OTOH if he means by the q that if you split on p say, and the clauses arising frpm p resolve to the empty clause but the clauses arising from $\neg p$ don't, then the fmla is consistent.

⋆ **34** ⋆ *Definite clauses*

If we resolve two nonempty definite clauses we get a nonempty definite clause. So, by induction, the only things we can deduce from nonempty definite clauses are other nonempty definite clauses. Since no definite clause is empty, we cannot deduce the empty disjunction, which is to say we cannot deduce the false!

One of Larry's questions

Are $\{P(x,b), P(a,y)\}$ and $\{P(a,b)\}$ equivalent?

$\{P(x,b), P(a,y)\}$ is $(\forall xy)(P(x,b) \vee P(a,y))$. Instantiating '$x$' to '$a$' and '$y$' to '$b$' we infer $P(a,b)$. The converse is obviously not going to be provable: take a universe just containing a and b and decide that $\neg P(b,b)$ and $\neg P(a,a)$. (don't worry about the truth-values of the other three atomics—they don't matter).

Are $\{P(y,y), P(y,a)\}$ and $\{P(y,a)\}$ equivalent?

This one looks uncannily like a tarted up version of Russell's paradox. Perhaps i just have a nasty suspicious mind. Let's rewrite P as $\in$ (and as infix) to get

Are $\{y \in y, y \in a\}$ and $\{y \in a\}$ equivalent?

which are $(\forall y)(y \notin y \rightarrow y \in a)$ and $(\forall y)(y \in a)$ and it's already much clearer what is going on. They are obviously not equivalent, but it might be an idea to cook up a small finite countermodel. One like this, perhaps:

$a \notin a$, $a \in b$, $c \in c$, $c \notin b$, $b \in b$.

b contains all things that are not members of themselves (namely a), but it doesn't contain everything (it's missing c).

Lots of funny symbols—a FAQ

The following symbols appear in this course at various times, and people often wonder about the differences. $\rightarrow$, $\Rightarrow$, `->`, $\models$, $\vdash$.

$\rightarrow$ and `->` are the same symbol in the sense of being different manifestations of the same symbol in two different fonts.

They have several uses. $A \rightarrow B$ is a notation denoting the set of all functions from A into B where A and B are sets (or types). Also $\rightarrow$ is a constructor of the recursive datatype of (propositional) formulæ: sticking a '$\rightarrow$' between two formulæ results in another formula. Punning between these two uses of the one symbol plays an important part in understanding λ calculus.

When '$\rightarrow$' is used as a constructor of the recursive datatype of (propositional) formulæ it is often provided with semantics to make $p \rightarrow q$ mean the same as '$\neg p \vee q$'. It was not always thus. There is also the symbol '$\supset$' which is also a constructor and is only ever used for this purpose. '$\rightarrow$' was first used in propositional logic quite explicitly to *not* mean the same thing as the truth-functional connective '$\supset$' (since there was an ideological debate around whether or not the relation encapsulated by '$\neg p \vee q$' could properly be regarded as implication!). Sadly this symbol is hardly used any longer, and '$\rightarrow$' is now used for both.

The symbol '$\vdash$' is usually placed between a term denoting a theory or a set of formulæ (to its left) and a single formula (to its right). Occasionally even the thing on the left can be a single formula. In these circumstances '$\vdash$' belongs to a metalanguage and means that there is a deduction of the formula on the right from the theory or set of formulæ on the left. Sometimes it has a subscript ('T' for example) denoting a theory or Logic in which the deduction is to be performed. The relation it captures is sometimes called **syntactic entailment** (to be contrasted with semantic entailment below).

There are people (like me!) who write sequents with '$\vdash$' in the middle instead of '$\Rightarrow$' (as Dr. Paulson does). In this usage '$\vdash$' is not a piece of metalanguage but is a constructor of a formal language. However the commonest use of the double-shafted arrow is probably its use as a piece of slang metalanguage, where it is put between two formulæ (or more typically between two lines of a proof) to mean that the second follows (in some sense to be divined from context) from the first.

Notice the difference between this use and the way we can put '$\rightarrow$' between two formulæ. Putting '$\rightarrow$' between two formulæ results in **another formula of the same language that the two given formulæ belonged to**. The slang use of '`=>`' really involves putting '$\Rightarrow$' between two *names* of formulæ so as to make an assertion about how those two formulæ are related, and this assertion is not an object of the same language that the two given formulæ belonged to, but is a piece of English (or rather: English with knobs on).

'$\mathcal{M} \models T$' and '$\mathcal{M} \models \phi$' are two correct uses of '$\models$'. The first means that the theory (set of formulæ) T is true in the structure $\mathcal{M}$, and the second means that the formula ϕ is true in the structure $\mathcal{M}$.

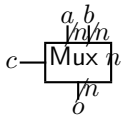
There is another use of '$\models$'. It can be used, like '$\vdash$' between a formula (or something denoting a set of formulæ) on the left, and a formula on the right. The resulting expressions mean that every interpretation making the stuff on the left true makes the stuff on the right true too, and this relation is usually called **semantic entailment**.

Chapter 7

Hardware verification exercises

1. Explain how the behaviour and structure of hardware designs can be represented in higher order logic. How is the sequential, or time-dependent, behaviour of a device modelled?

2. Consider the n-bit multiplexer Mux n shown below:

   When the control wire c is high, the n-bit value present on the input a appears on the n-bit output o. And when the control value c is low, the n-bit value present on the input b appears on the n-bit output o. Write a formal specification of the device Mux n. Design an implementation of Mux n using only combinational logic gates (AND-gates, OR-gates, etc.). Verify by formal proof that your design meets its specification.

3. Using only simple combinational logic gates, design a device to increment an n-bit binary number by one. Write a formal specification for your device, and verify the correctness of its design by formal proof.

1991:7:7

Want `pty in = out`.

To do this it is sufficient to show that for each n,

`pty in` n = `out` n

This is because we have a kind of implicit extensionality that says two functions are the same if they always give the same values to the same arguments.

Clearly the way to prove this is by induction on n. The case $n = 0$ is easy to deal with: both sides of the equation evaluate to 0.

Now we have to deal with the induction case. Assume that it is true for n, and infer that it holds for $n + 1$.

Thus we are **given**

`pty in` n = `out` n

and we **want**

`pty in` $n + 1$ = `out` $n + 1$

The LHS is `pty in` $n+1$ which (by dfn of `pty`) is

$$\texttt{in}\ n\ \texttt{=>}\ \neg(\texttt{pty in}\ n)|(\texttt{pty in}\ n)$$

which is

$$\texttt{in}\ n\ \texttt{=>}\ \neg(\texttt{out}\ n)|(\texttt{out}\ n)$$

by induction hypothesis. Now we also know

$$(\forall n)(\texttt{out}\ \ n\ =\ (n=0\ \ \texttt{=>}\ 0 \mid \texttt{1}(n-1)))$$

but we are only interested on the case $n > 0$ (it's the induction case we are examining, after all!) so wlog 'n' can be rewritten as '$n+1$' so this simplifies to

$$\texttt{out}\ \ (n+1) =\texttt{1}\ n$$

and the RHS of this equation is

$$\neg(\texttt{in}\ n = \texttt{out}\ \ n)$$

which is the same as

$$\texttt{in}\ n\ \ \texttt{=>}\ \ \neg\ (\texttt{out}\ \ n)|(\texttt{out}\ \ n)$$

as desired.

Another answer to the same question

Tom says:

What we're *really* proving is:

$$[\exists l.\texttt{XOR}(in, out, l) \wedge \texttt{REG}(in, out)]$$

So assume for some l that

1. $\forall t.l(t) = \neg(in(t) = out(t))$
2. $\forall t.out(t) = (t = 0 => 0|l(t-1))$

Prove $\forall n.out(n) = \texttt{PTY}(in(n))$ by induction on n.
Base case: $n = 0$.
by (2) $out(t) = (0 = 0 => 0|l(0-1))$ which is 0 (by definition of conditional) which is $\texttt{PTY}(in(0))$ (by definition of `PTY`)
Step case. Assume $out(n) = \texttt{PTY}(in(n))$. Show:
$out(n+1) = (n+1 = 0 => 0|l((n+1)-1))$ (specialise n to 0 in (2))
$= l(n)$
$= \neg(in(n) = out(n))$ (by (1))
$= \neg(in(n) = \texttt{PTY}(in(n)))$ by induction hypothesis.
$= (in(n) => \neg\texttt{PTY}(in(n))|\texttt{PTY}(in(n)))$ by case analysis
$= \texttt{PTY}(in(n+1))$ by dfn of `PTY`

Chapter 8

ML

1997:2:1 (a)
Q:
Give some ML text to replace <insert> in the following:

```
<insert>
fun f g g = g; f x y;
```

to make it into a valid ML program.
Answer

```
datatype w = g;
val x = g;
val y = g;
fun f g g = g; fun x y;
```

8.1 Some ML exercises

1. Working in ML, represent positive integers in binary as bool lists. Write ML code to do multiplication and addition.
2. Develop matrix manipulation routines of your choice representing matrices as list list e.g., transpose, determinants, inverses. Perhaps even declare a new datatype?
3. Like 1 but include negative integers as well.[1]
4. The knapsack problem is: given a list L of integers, and an integer n, find a subset $L' \subseteq L$ which adds up to n. Write an ML program to solve it. How long does your code take to run as a function of the length of the input list?
5. A (nonempty) Conway game is an ordered pair of sets of Conway games. Write ML code to develop an automated theory of the arithmetic of Conway games. See J.H.Conway "On numbers and Games" Academic Press.

8.2 Answers

8.2.1 Question 8.1

Pompiboon Satangput's answer

[1] Could use lazy lists of booleans for this!

```
fun Tand( one , [] ) = []
        |Tand( [] , two ) = []
        |Tand( n::ones , t::twos ) =
                (n andalso t) ::Tand(ones,twos);

fun Tor( one , [] ) = one
        |Tor( [] , two ) = two
        |Tor( n::ones , t::twos ) =
                (n orelse t) ::Tor(ones,twos);

fun NumToBi( 0 ) = []
        | NumToBi( n ) = (n mod 2 = 1) ::NumToBi(n div 2);

fun BiToNum( [] ) = 0
        | BiToNum( i::inp ) =
                if i then 2*(BiToNum(inp))+1
                else 2*(BiToNum(inp));

fun AndNumber(n,m) = BiToNum(Tand(NumToBi(n),NumToBi(m)));

fun OrNumber(n,m) = BiToNum(Tor(NumToBi(n),NumToBi(m)));
```

An answer from Andrew Rose

```
(*  ------------------------------------------------  *)
(*                 BINARY MANIPULATIONS               *)
(*                     ANDREW ROSE                    *)
(*                      12/11/97                      *)
(*  ------------------------------------------------  *)

(*  The code below defines the following functions...

        bbitAND(x,y)    - bitwise AND
        bbitOR(x,y)     - bitwise OR
        bbitNOT(x)      - bitwise NOT

        blogAND(x,y)    - logical AND
        blogOR(x,y)     - logical OR
        blogNOT(x,y)    - logical NOT

        bAdd(x,y)       - Addition
        bSub(x,y)       - Subtraction (2's Complement)
        bMult(x,y)      - Multiplication

        BinToDec(x)     - Binary to Decimal Conversion
        Bin2CToDec(x)   - Binary to Decimal Conversion
                          using 2's complement
        DecToBin(x)     - Decimal to Binary Conversion
        DecToBin2C(x)   - Decimal to Binary Conversion
                          using 2's complement

All values are represented as binary lists with MSB first.
```

```
The function bSub(x,y) subtracts y from x.              *)

(*  ------------------------------------------------  *)



(*  ------------------------------------------------  *)
(*                 BITWISE OPERATIONS                 *)
(*  ------------------------------------------------  *)


fun eqlength(x,y) =
        if length(x) = length(y) then
                (x,y)
        else
                if length(x) > length(y) then
                        eqlength(x,false::y)
                else
                        eqlength(false::x,y)
                (**)
        (**)
;

fun xbbitAND([],_) = []
  | xbbitAND(xs,[]) = []
  | xbbitAND(x::xs,y::ys) =
        if x andalso y then
                true::xbbitAND(xs,ys)
        else
                false::xbbitAND(xs,ys)
        (**)
;


fun bbitAND(x,y) = xbbitAND(eqlength(x,y));


fun xbbitOR([],_) = []
  | xbbitOR(xs,[]) = []
  | xbbitOR(x::xs,y::ys) =
        if x orelse y then
                true::xbbitOR(xs,ys)
        else
                false::xbbitOR(xs,ys)
        (**)
;


fun bbitOR(x,y) = xbbitOR(eqlength(x,y));


fun bbitNOT([]) = []
  | bbitNOT(x::xs) = (not(x))::bbitNOT(xs);

(*  ------------------------------------------------  *)
```

```
(*  ------------------------------------------------  *)
(*                LOGICAL OPERATIONS                  *)
(*  ------------------------------------------------  *)

fun IsFalse([]) = true
  | IsFalse(x::xs) =
        if (x = true) then
                false
        else
                IsFalse(xs)
        (**)
;


fun blogAND([],_) = false
  | blogAND(_,[]) = false
  | blogAND(xs,ys) =
        if IsFalse(xs) orelse IsFalse(ys) then
                false
        else
                true
        (**)
;


fun blogOR([],[]) = false
  | blogOR([],ys) =
        if IsFalse(ys) then
                false
        else
                true
        (**)
  | blogOR(xs,[]) =
        if IsFalse(xs) then
                false
        else
                true
        (**)
  | blogOR(xs,ys) =
        if IsFalse(xs) andalso IsFalse(ys) then
                false
        else
                true
        (**)
;


fun blogNOT([]) = true
  | blogNOT(xs) = not(IsFalse(xs));

(*  ------------------------------------------------  *)



(*  ------------------------------------------------  *)
(*                'MATHS' OPERATIONS                  *)
(*  ------------------------------------------------  *)
```

```
fun fst(x,y) = x;
fun snd(x,y) = y;

(*  A function xbAdd(x,y,carry) adds together two equal
    length boolean lists with MSB last.                *)

exception UnEqual;

fun xbAdd([],[],carry) =
        if carry then
                [true]
        else
                []
        (**)
  | xbAdd(xs,[],carry) = raise UnEqual
  | xbAdd([],ys,carry) = raise UnEqual
  | xbAdd(x::xs,y::ys,carry) =
        if x andalso y then
                carry::xbAdd(xs,ys,true)
        else
                if x orelse y then
                        (not carry)::xbAdd(xs,ys,carry)
                else
                        carry::xbAdd(xs,ys,false)
                (**)
        (**)
;


(*  A function bAdd(x,y) prepares two boolean lists of any
    length with MSB first by making them equal lengths
    with MSB last before passing them to xbAdd          *)

fun bAdd(x,y) = rev(xbAdd(rev(fst(eqlength(x,y))),rev(snd(eqlength(x,y))),false));

fun bComp2(x) =
        if IsFalse(x) then
                x
        else
                bAdd(bbitNOT(x),[true]);
        (**)

fun bAdd(x,y) = rev(xbAdd(rev(fst(eqlength(x,y))),rev(bComp2(snd(eqlength(x,y)))),false));

fun shifted(acc) =
        if acc=0 then
                []
        else
                false::shifted(acc-1)
        (**)
;

fun xbMult([],ys,acc) = [false]
  | xbMult(x::xs,ys,acc) =
        if x = false then
                xbMult(xs,ys,acc+1)
        else
                bAdd(ys @ shifted(acc),xbMult(xs,ys,acc+1))
```

```
        (**)
;

fun bMult(x,y) = xbMult(rev(x),y,0);

(*  ------------------------------------------------  *)




(*  ------------------------------------------------  *)
(*                      CONVERSIONS                   *)
(*  ------------------------------------------------  *)



nonfix POW;

fun POW(x,y) =
        let fun POWI(x,y,acc) =
                if y = 0 then
                        acc
                else
                        POWI(x,y-1,x*acc)
                (**)
        in
                POWI(x,y,1)
        end
;

infix POW;


fun xBinToDec([],n,Ans) = Ans
  | xBinToDec(x::xs,n,Ans) =
        if x then
                xBinToDec(xs,n+1, (2 POW n) + Ans)
        else
                xBinToDec(xs,n+1,Ans)
        (**)
;

fun BinToDec(xs) = xBinToDec(rev(xs),0,0);

fun Bin2CToDec([]) = 0
  | Bin2CToDec(x::xs) =
        if x then
                ~(BinToDec(bComp2(xs)))
        else
                BinToDec(xs)
        (**)
;


fun xDecToBin(x,bs) =
        let
                val tf = if (x mod 2) = 1 then true else false
        in
```

```
                if x=0 then
                        bs
                else
                        xDecToBin((x div 2),tf::bs)
                (**)
        end
;

exception Negative;

fun DecToBin(x) =
        if x<0 then
                raise Negative
        else
                xDecToBin(x,[])
        (**)
;


fun DecToBin2C(x) =
        if x<0 then
                bComp2(DecToBin(abs(x)))
        else
                false::DecToBin(abs(x))
        (**)
;

(*  ------------------------------------------------  *)
```

An answer from Inge Norum

```
(*                              BINARY FUNCTIONS                              *)
(*                              ================                              *)
(*                                                  by Inge Norum Oct '97 *)

(*  First  convention: Binary number represented as a list of booleans with
    the least significant bit first.                                          *)

(*  Second convention: no trailing falses (zeroes) are allowed.  The last
    element of any list must be a true or [].  ( => gain in efficiency )    *)

(*  Third  convention: [false] will be used as a fail 'signal' and will
    signify that a function failed.                                           *)

val error_b = [false];
val error_i = ~1;


(*  A recursive funtion to turn an integer into a list of booleans:  *)

fun bin (i : int) =
    if i>0 then ((i mod 2) = 1) :: bin (i div 2)
    else [];

(*  An iterative funtion to turn a bool list into an integer:  *)
```

```
fun int_b (b : bool list) =
    let
        fun intb([]   , n, i) = i : int
          | intb(b::bs, n, i) = intb(bs, n+n, if b then n+i else i)
    in
        if b=[false] then error_i else intb(b, 1, 0)
    end;

(*  A function to strip off trailing falses:  *)


fun strip (b::bs) =
    if b then b :: strip(bs)
    else let val stripped = strip(bs)
         in  if stripped = [] then []
             else b :: stripped
         end
  | strip ([]) = [];

(*  An iterative binary addition function using only logical operators:     *)

fun add_b (a, b) =
    let fun addb([],[], carry, sum) = if carry then (true :: sum) else sum
          | addb([], y, carry, sum) = addb([false], y, carry, sum)
          | addb(x ,[], carry, sum) = addb(x, [false], carry, sum)
          | addb(x::xs, y::ys, carry, sum) =
              if (x andalso y)     then addb(xs, ys, true ,     carry :: sum)
              else if (x orelse y) then addb(xs, ys, carry, not carry :: sum)
                                   else addb(xs, ys, false,     carry :: sum)
    in  rev(addb(a, b, false, []))
    end;


(*  An iterative binary subtraction function:  *)

fun sub_b (a, b) =
    let fun subb([], [], borrow, dif) =
              if borrow then error_b  (* Failed:  b>a *)
              else strip(rev(dif))    (* strip off any trailing zeroes *)
          | subb(x, [], borrow, dif) = subb(x, [false], borrow, dif)
          | subb([], y, borrow, dif) = subb([false], y, borrow, dif)
          | subb(x::xs, y::ys, borrow, dif) =
              if (y andalso borrow)     then subb(xs, ys, true ,     x :: dif)
              else if (y orelse borrow) then subb(xs, ys, not x, not x :: dif)
                                        else subb(xs, ys, false,     x :: dif)
    in  subb(a, b, false, [])
    end;


(*  Currying and Uncurrying functions!  (for functions with two argumets)  *)

fun curry(f) n m = f(n,m);

fun uncurry(f) = fn (n,m) => (f n) m;

(*  now curry add_b and sub_b:  *)

val plus_b = curry add_b;
```

```
val subt_b = curry sub_b;


(*  A binary nfold function:  *)

fun nfold_b(f,[]) x = x
  | nfold_b(f, n) x = f(nfold_b(f, sub_b(n, [true])) x);

(*  Now, a product and power funtion can be defined:  *)

fun nprod_b a b = nfold_b(plus_b a, b) [false];
fun npower_b a b = nfold_b(nprod_b a, b) [true];

(*  However, these functions are very inefficient, and hence called n... *)


(*  A right bit shifting function:  Since I have a binary number represented
    with the least significant digit first, I only have to take away
    boolean(s) from the beginning of the list in order to 'shift off' the
    least significant bit(s)                                              *)

fun shr(b::bs, n::ns) = shr(bs, sub_b(n::ns, [true]))
  | shr(b::bs, []) = b::bs
  | shr([], _) = error_b;

(*  A left bit shift function:  ( -> inserts falses onto beginning)  *)

fun shl(b, n::ns) = shl((false :: b), sub_b(n::ns, [true]))
  | shl(b, []) = b;


(*  Bit shifting can be exploited to optimize functions: a left shift
    doubles a binary number n times, and a right shift halves it n times.  *)

(*  Now a product function which explots bit shifting:  *)

fun prod_b(a, b::bs) =
    if bs = [] then if b then a else []  (* done *)
    else if b then add_b(a, prod_b(shl(a,[true]), shr(b::bs, [true])))
              else (*even*) prod_b(shl(a,[true]), shr(b::bs, [true]))
  | prod_b(_, _) = [];  (* one of the arguments must have been [] *)

(*  and using this, an efficient power function:  *)

fun power_b(a, n::ns) =
    if ns = [] then if n then a else []  (* done *)
    else if n then prod_b(power_b(prod_b(a, a), shr(n::ns, [true])), a)
              else        power_b(prod_b(a, a), shr(n::ns, [true]))
  | power_b([], _) = []
  | power_b(_, []) = [true];


(*  An iterative general binary comparison function:  *)

fun lessthan(x::xs, y::ys, ans) = if x=y then lessthan(xs, ys, ans)
                                         else lessthan(xs, ys, y)
  | lessthan([], [], ans) = ans
  | lessthan([], _ , ans) = true    (* | Works since no trailing *)
```

```
  | lessthan(_ , [], ans) = false;  (* | falses are allowed!     *)


(*  Then use this to overload the <, <=, >=, > operators for bool lists:  *)

fun op<  (a : bool list, b : bool list) = lessthan(a, b, false);
fun op<= (a : bool list, b : bool list) = lessthan(a, b, true);
fun op>= (a : bool list, b : bool list) = lessthan(b, a, true);
fun op>  (a : bool list, b : bool list) = lessthan(b, a, false);


(*  Lastly, a function to turn a curried binary function
    into a function with integer I/O ...                  *)

fun intfun_b(f_b) x y = int_b((f_b (bin x)) (bin y));

(*  Now generate some curried integer functions from the binary ones:  *)

val plus   = intfun_b plus_b;
val subt   = intfun_b subt_b;
val add    = uncurry plus;
val nprod  = intfun_b nprod_b;
val npower = intfun_b npower_b;
val prod   = intfun_b(curry prod_b);
val power  = intfun_b(curry power_b);
```

8.2.2 Question 8.2

An answer from Andrew Rose

```
(*  ------------------------------------------------  *)
(*                   MATRIX OPERATIONS                *)
(*                      ANDREW ROSE                   *)
(*                       12/11/97                     *)
(*  ------------------------------------------------  *)


(*  The code below defines the following functions...

        mCof(A)         - Cofactors of an element
        mDet(A)         - Determinent of A

        !HELP! - mCof depends on mDet and mDet depends
                 on mCof.  This means that you have to
                 define 1 of them in a dummy manner
                 before loding the code.  Then you have
                 to load the code twice!

        mTsp(A)         - Transpose of A
        mAdj(A)         - Adjoint of A
        mInv(A)         - Inverse of A

All matrices to be represented by a list of lists.
Each list is a row of the matrix.                      *)


(*  ------------------------------------------------  *)
```

```
exception mNoSuchRow;
exception mNoSuchCol;
exception mNoSuchItem;
exception mTooFewElements;
exception mNoMatrix;
exception mDivByZero;

fun mTrimRow(r,[]) = raise NoSuchRow
  | mTrimRow(r,x::xs) =
        let fun xmTrimRow(r,[],acc) = raise NoSuchRow
              | xmTrimRow(r,x::xs,acc) =
                if r = acc then
                        xs
                else
                        x::xmTrimRow(r,xs,acc+1)
                (**)
        in
                xmTrimRow(r,x::xs,1)
        end
;

fun mTrimItem r ([]) = raise NoSuchItem
  | mTrimItem r (x::xs) =
        let fun xmTrimItem(r,[],acc) = raise NoSuchItem
              | xmTrimItem(r,x::xs,acc) =
                if r = acc then
                        xs
                else
                        x::xmTrimItem(r,xs,acc+1)
                (**)
        in
                xmTrimItem(r,x::xs,1)
        end
;

fun mTrimCol(c:int,[]) = raise NoSuchCol
  | mTrimCol(c,A : int list list) = map (mTrimItem c) A
;

fun nth ([],n) = raise TooFewElements
  | nth (x::xs,n) = if n=1 then x else nth(xs,n-1);

nonfix POW;
fun POW(x,y) =
        let fun xPOW(x,y,acc) =
                if y=0 then acc else xPOW(x,y-1,x*acc)
        in
                xPOW(x,y,1)
        end
;
infix POW;

fun mCof(r,c) A = mDet(mTrimCol(c,mTrimRow(r,A))) * ((~1) POW (r+c))

fun mDet([]) = raise NoMatrix
  | mDet(A) =
        let fun xmDet(A,length,done,acc) =
                if done = length then
```

```
                        (nth(hd(A),done)*(mCof(1,done) A))+acc
                else

xmDet(A,length,done+1,(nth(hd(A),done)*(mCof(1,done) A))+acc)
                (**)
        in
                if length(hd(A))=1 then
                        hd(hd(A))
                else
                        xmDet(A,length(A),1,0)
                (**)
        end
;


fun mTsp([]) = [[]]
  | mTsp(A) =
        if hd(A)=[] then
                []
        else
                map hd(A)::mTsp(map tl(A))
        (**)
;


fun mRowAdj(r,A) =
        let fun xmRowAdj(r,A,acc,cnt) =
                if cnt=length(A)+1 then
                        acc
                else
                        xmRowAdj(r,A,acc @ [mCof(r,cnt) A],cnt+1)
                (**)
        in
                xmRowAdj(r,A,[],1)
        end
;


fun mAdj(A) =
        let fun xmAdj(A,acc,x) =
                if (x=length(A)+1) then
                        acc
                else
                        xmAdj(A,mRowAdj(x,A)::acc,x+1)
                (**)
        in
                xmAdj(A,[],1)
        end
;

fun mInv(A) =
        if mDet(A) = 0 then
                raise DivByZero
        else
                (mDet(A),mAdj(A))
        (**)
;
```

An answer from David Burleigh

```
(* Matrix Determinant Program.  By David Burleigh *)
(* All these functions treat a matrix as a list of columns *)

fun butnth([],n) = []    (* returns all the list elements except the nth *)
  | butnth(x::xs,n) =
 if n = 0 then butnth(xs,n-1)
 else x::butnth(xs,n-1);

fun nth([],_) = [] (* returns the nth element of a list *)
  | nth(x::xs,0) = x
  | nth(x::xs,n) = nth(xs,n-1);

(* getsubdet returns a the subdeterminant which is the tails of all the
columns
   except the column containing the value to which the subdeterminant refers.
*)

fun getsubdet [] _  = [[]]
  | getsubdet cols col = map tl (butnth(cols,col));

fun subdet([]) = 0 (* calculates the base case, which is a 2x2 matrix *)
  | subdet([[a,b],[c,d]]) = a*d - b*c;

fun even n = (n mod 2 = 0); (* even returns true is n is even *)

(* makedet takes a matrix and the values of its subdeterminants
   and calculates the overall determinant, using even on the column
   number n (note that the first column has n = 0) to decide
   whether to add or subtract the next value.  *)

fun makedet(mtx, subs) =
 let fun md([], [],n) = 0
   | md(c::cs, s::ss,n) =
  let val num = hd(c)
  in
   if even n
   then  num * s + md(cs,ss,n+1)
   else ~num * s + md(cs,ss,n+1)
  end
 in md(mtx,subs,0)
 end;

(* listsubdets reduces a matrix to its constituent subdeterminants,
   calling itself recusively untill it reaches a 2x2 matrix which
   it evaluates using subdet.  The subdeterminant lists are built up
   in reverse so as to avoid append, so they have to be reversed before
   being passed to makedet.  *)

fun listsubdets(mtx,n,subs) =
 if nth(mtx,n) = [] (*end of matrix*)
 then subs
 else
 let val sb = getsubdet mtx n
```

```
 in
  if length(hd(sb)) = 2   (* 2x2 matrix reached *)
  then listsubdets(mtx,n+1,subdet(sb)::subs)
  else listsubdets(mtx,n+1,
   makedet(sb,rev(listsubdets(sb,0,[])))::subs)
 end;

(* det is the actual determinant function which calls makedet on the result
   of listsubdets.  This function has O(a^n) time complexity where n is the
   dimensionality of the matrix, since T(n+1) = (n+1) * T(n).  *)

fun det(mtx) = makedet(mtx,rev(listsubdets(mtx,0,[])));

(* makematrix generates an n x n matrix whose determinant is zero, for test
purposes.
   On my P166, det does an 8x8 matrix in 5 seconds, and a 9x9 matrix in 30
seconds. *)
fun makematrix(n) =
 let val dim = n
 in let fun mm(r,mtx) =
  let fun mc(cs,ce,mtx) =
   if cs = ce
   then mtx
   else mc(cs-1,ce,cs::mtx)
  in
   if r = 0
   then mtx
   else mm(r-1, mc(r*n,(r-1)*n,[])::mtx)
  end
 in mm(n,[])
 end
 end;
```

8.2.3 Knapsack question

```
letrec knapsack nlist num = if null nlist then if num = 0 then nlist else fail
                         else let h,t = (hd nlist),(tl nlist) in
                         (if num <0 then fail else
                         else [h] @ (knapsack t (num - h) )   ?
                          knapsack t num);;
```

How long does this thing take to run as a function of the length of the input list?

```
h = 2 t = [4;8;16;32;64;5] num = 25   [2]@(knapsack [4;8;16;32;64;5] 23)

h = 4 t = [8;16;32;64;5]   num = 23   [4] @ knapsack [8;16;32;64;5] 19
```

8.3 ML ticks

8.3.1 ML tick 4

A reasoned answer from Jonathan Page, hacked about by me.

```
(*  PROBLEM 1.  A function startpoints(pairs,z) ...  *)

fun startpoints([],_)=[]
  | startpoints((start,finish)::pairs,z)=
    if (finish=z) then
      start::startpoints(pairs,z)
    else
      startpoints(pairs,z);


(*  PROBLEM 2.  A function endpoints(z,pairs) ...  *)

fun endpoints(_,[])=[]
  | endpoints(z,(start,finish)::pairs)=
    if (start=z) then
      finish::endpoints(z,pairs)
    else
      endpoints(z,pairs);


(*  PROBLEM 3.  A function allpairs(xs,ys) ...  *)

fun allpairs(xs,ys)=
  let
    fun prefix(_,[])=[]
      | prefix(x,y::ys)=(x,y)::prefix(x,ys)

    fun makeallpairs([])=[]
      | makeallpairs(x::xs)=prefix(x,ys) @ makeallpairs(xs)
  in
    makeallpairs(xs)
  end;


(*  PROBLEM 4.  A function addnew((x,y),pairs) ...  *)

fun addnew((x,y),poss)=
  let
    val us=startpoints(poss,x)
    val vs=endpoints(y,poss)
    val new=allpairs(x::us,y::vs)

    fun inject(w,zlist)=
      let
        fun doinject([])=w::zlist
          | doinject(z::zs)=
            if (z=w) then
              zlist
            else
              doinject(zs)
      in
        doinject(zlist)
      end;

    fun merge([])=poss
      | merge(w::ws)=inject(w,merge(ws))
  in
    merge(new)
```

```
  end;
```

Notice that if you are not careful merge takes time n^2 where n is the length of the longer list, whereas it shouldn't taker longer than the product of the lengths. The point is that merge is allowed to assume that the input lists are duplicate-free. If you merge two lists by testing the head of the first list for membership in the second, and adding it if the answer is 'no' you end up checking for duplicates in the first list.

```
fun merge l1 [] = l1
 |  merge (a::as) b = if mem a b then merge as b else a::(merge as b);

(*  PROBLEM 5.  A function routes(pairs) ...  *)

fun routes[]=[]
  | routes((x,y)::pairs)=addnew((x,y),routes(pairs));
```

Various things to keep in mind. Decide at the outset whether you are measuring time taken as a function of the number of *nodes* or the numbr of *edges*.

The function routes(pairs) is $O(n^5)$ in time, where n is the number of pairs ("directed roads") in the input list. This may be established by the following argument:

1. During each invocation of `addnew`$((x, y), $`pairs`$)$, at maximum one "road", and hence two new nodes may be added to the "one-way road system".

2. Thus, the maximum number of unique nodes in the road system after i invocations of `addnew`$((x, y), $`pairs`$)$ is $2i$.

3. In the case where the input to `addnew` consists of a list of edges saying: half the world (n nodes, say) can see `a`, and `b` can see the other half, together with an edge joining `a` to `b`, the output will be of length $(n + 1)^2$ when the input list was of length $2n$. This can happen for arbitrarily large n so `addnew` is of complexity at least $O(n^2)$ and clearly of complexity at most $O(n^2)$.

4. At the start of the ith invocation of `addnew`$((x, y), $`poss`$)$, `poss` will mention at most $2(i - 1) = 2i - 2$ unique nodes, due to 2). Step 3) shows that poss will never include the same pair more than once. Thus, an upper limit on the number of nodes in the list of nodes `us`, output by the function call `startpoints`$($`poss`$, x)$, is $2i - 2$, given by the case where all the nodes mentioned in the list of routes poss have a route going from them to the node x.

5. Similarly, an upper limit on the number of nodes in the list of nodes `vs`, output by the function call `endpoints`$(y, $`poss`$)$, is also $2i-2$, given by the case where all the nodes mentioned in the list of routes poss have a route going from the node y to them.

6. Now, the function `allpairs(xs,ys)` produces the list of all (x, y) for x in the list `xs` and y in the list `ys`. Thus, an upper limit on the number of pairs in the output list `new` of the function call `allpairs`$(x$ ::`us`$, y$ ::`vs`$)$ is $((2i-2)+1) \times ((2i-2)+1)$, due to 5) and 6) above, which equals $4i^2 - 4i + 1$. This is of the order i^2.

7. The function `merge` takes each pair in the list new, and via the `inject` and `doinject` functions, performs a linear search of the list `poss`, only adding the pair if it is not found in the list. The result is the concatenation of the two lists, but with duplicates removed, so that each pair appears once at most. Thus, roughly speaking, `merge()` is $O(ab)$ in time, where a is the number of pairs in `new` and b is the number of pairs in `poss`. (This corresponds to the worst case scenario, where none of the pairs in `new` are present in the list `poss`, and so the whole list must be searched each time). Actually, `merge()`'s operation is more complicated than this, as the length of the list to search increases as pairs are added to it, but this cannot affect `merge()`'s complexity if the order of the number of items added is equal to or less than the order of the number of items in the list being added to.

8. Thus, during the ith invocation of `addnew`$((x, y), \texttt{poss})$, the order of the time taken by the function call `merge(new)` is given by the worst case scenario (which could in fact be unattainable), where the following three conditions hold:

   (a) the no. of pairs in `new` is of the order of i^2 (see 7))

   (b) the no. of pairs in `poss` is of the order of i^2 (see 4))

   (c) none of the pairs in `new` are present in `poss` (see 8))

   Thus, `merge(new)` is $O((i^2)*(i^2)) = O(i^4)$ in time during the ith invocation of `addnew`$((x, y), \texttt{poss})$ (the function call `merge(new)` being made exactly once per invocation of `addnew`$((x, y), \texttt{poss})$).

9. Now, the function call `merge(new)` is made exactly n times, once for each of the n pairs in the list input to the function `routes(pairs)`. Thus, i ranges from 1 to n. So the order of the time taken by the `routes(pairs)` function is given by the order of the sum of i^4 from $i = 1$ to $i = n$ (see 9)). Thus, the function `routes(pairs)` is $O(n^5)$ in time.

10. It should be noted that each of the function calls `startpoints`$(\texttt{poss}, x)$ and `endpoints`$(y, \texttt{poss})$ is linear in time (with respect to the number of pairs in `poss`), and that the function `allpairs(xs,ys)` is $O(xy)$ in time, where x is the number of nodes in `xs`, and y is the number of nodes in `ys`. However, these times do not affect the complexity of `addnew`$((x, y), \texttt{poss})$ as they are negligible in comparison to the worst case times for the execution of `merge(new)`, and so have been ignored in evaluating the complexity of routes(pairs).

11. Although the above argument shows that `routes(pairs)` is $O(n^5)$ in time, this does not mean that `routes(pairs)` cannot also be $O(n^4)$ in time, or indeed $O(n^3)$ or $O(n^2)$ etc. However, it can fairly easily be shown (though the proof is omitted for brevity) that if `routes(pairs)` is called with pairs equal to:

    ```
    [(1,2),(2,1),(2,3),(3,2),(3,4),(4,3),... ...,(n/2,n/2+1),(n/2+1,n/2)],
    ```

    with n even, but arbitrarily large, then the execution time for `routes(pairs)` is of order n^5. This example, together with the result of 10) shows that $k = 5$ is the smallest value for which `routes(pairs)` is $O(n^k)$ in time.

Some cubic code (using mutable lists) from Andrei Legostaev

```
%(*  ML ASSESSED EXERCISES, TICK 4 AND 4* SUBMISSION FROM A. LEGOSTAEV...  *)
```

```
(*  The algorithm works by representing the given network in terms of     *)
(*  "nodes" and then finding all possible routes.                         *)
(*                                                                        *)
(*  A Node is a data structure representing all the information about a   *)
(*  particular point.  It is a four-tuple.  The elements are:             *)
(*                                                                        *)
(*  1. The point identificator (as in the input)                          *)
(*  2. Ref to a list of references to every node which can be reached     *)
(*     directly from the current node (i.e. in one step)                  *)
(*  3. A boolean flag, "ture" if the node has been visited by the         *)
(*     routine which explores possible routes.                            *)
(*  4. A boolean flag, "true" if there is a "loop" route leading back to  *)
(*     the node and this fact has been recorded.                          *)

datatype 'a Node =

     makeNode of ('a * ((('a Node) list) ref) * (bool ref) * (bool ref));




(*  function routes                                                       *)
(*                                                                        *)
(*  Argument:  a list of pairs of route points                            *)
(*  Returns:   a complete list of pairs such that there is a route from   *)
(*             the first point in the pair to the second through the      *)
(*             network given as the argument.                             *)

fun routes (pairs) = let




(************************************************************************)
(************************************************************************)

(*  !nrList is a list of Nodes.  nrList is defined as a reference in      *)
(*  order to make modification of its contents possible.                  *)

val nrList = ref [];




(*  functions n1, n2, n3, n4                                              *)
(*                                                                        *)
(*  Argument:  A node.                                                    *)
(*  Return:    n1: 1st element of the node (point identeficator)          *)
(*             n2: 2nd element (reference to a list of nodes)             *)
(*             n3: 3rd element (bool flag "has been visited")             *)
(*             n4: 4th element (bool flag "there is a loop-route back")   *)

(*                                                                        *)
(*  Worst case complexity:  constant in time and space.                   *)
```

```
fun n1 (makeNode(a, b, c, d)) = a;
fun n2 (makeNode(a, b, c, d)) = b;
fun n3 (makeNode(a, b, c, d)) = c;
fun n4 (makeNode(a, b, c, d)) = d;
```

```
(*  pseudo-function getRef                                                  *)
(*                                                                          *)
(*  Arguments:  1. node identificator z (i.e. a route point)                *)
(*              2. the current node ref list.                               *)
(*  Returns:    The reference to node z. If node z does not exist at        *)
(*              the time of call, it is created.                            *)
(*                                                                          *)
(*  Worst case complexity:  linear in time and space.                       *)

fun getRef (z, []) = let
                         val newNode =
                                 makeNode (z, ref [], ref false, ref false)
                     in
                         nrList := newNode :: !nrList;
                         newNode
                     end
  | getRef (z, y::ys) = if   z=n1(y)
                        then y
                        else getRef(z, ys);
```

```
(*  pseudo-function updateNode                                              *)
(*                                                                          *)
(*  Arguments:  A pair of route points.                                     *)
(*  Returns:    Unity. nrList is updated with the information about the     *)
(*              given pair of route points.                                 *)
(*                                                                          *)
(*  Worst case complexity:  linear in time and space.                       *)

fun updateNode (a, b) = let
                            val dest = n2(getRef(a, !nrList))
                        in
                            dest := getRef(b, !nrList) :: !dest
                        end;
```

```
(*  function buildNodes                                                     *)
(*                                                                          *)
(*  Arguments:  A complete list of pairs of route points.                   *)
(*  Returns:    Unity.  Converts the complete list of pairs into a          *)
(*              complete node list (nrList).                                *)
(*                                                                          *)
(*  Worst case complexity:  quadratic in time and linear in space.          *)

fun buildNodes [] = ()
```

```
  | buildNodes ((x,y)::rest) = ( updateNode(x,y); buildNodes(rest) );



(*  function travelNodes                                                  *)
(*                                                                        *)
(*  Arguments: 1. Start point of the search                               *)
(*             2. Current point (input should be the same as 1.)          *)
(*             3. Points to visit (list of Nodes corresponding to 2.)  *)
(*             4. [] (internal accumulator)                               *)
(*  Returns:   A complete list of pairs, where in each pair the first     *)
(*             element is the start point 1., and the second element is   *)
(*             a point that can be reached from the start point.          *)
(*                                                                        *)
(*  Using "has been visited" flags it is rather difficult to detect       *)
(*  routes which lead back to the start point while avoiding both         *)
(*  looping and repeated recording of such routes.  The clumsy structure  *)
(*  of "if... then..." constructoins in the middle of the function deals  *)
(*  with that aspect.  It can be read as:                                 *)
(*                                                                        *)
(*  If the next point has already been visited then ignore it, unless it  *)
(*  is the start point.  In this case (unless it has already been done)   *)
(*  record the route (start,start) and set a flag to show that this       *)
(*  route has been recorded.                                              *)

fun travelNodes (s, c, [],         beenThere) = beenThere
  | travelNodes (s, c, g::goThere, beenThere) =

    let
        val goThere' = !(n2 g)
    in
        (n3 c) := true;

        if   !(n3 g)
        then if   (n1 g) = s andalso not (!(n4 g))
             then ( (n4 g) := true;
                    travelNodes(s, c, goThere, [(s,s)]) )
             else travelNodes(s, c, goThere, [])
        else let
                 val new = (s,(n1 g))::travelNodes(s, g, goThere', [])
             in
                 travelNodes(s, c, goThere, new @ beenThere)
             end
    end;



(*  pseudo-function clearFlags                                            *)
(*                                                                        *)
(*  Argument: !nrList                                                     *)
(*  Returns:  Unity.  All flags in !nrList are reset to "false".          *)

    fun clearFlags [] = ()
      | clearFlags (p::ps) = ( (n3 p) := false;
                               (n4 p) := false;
                               clearFlags (ps) );
```

```
(*************************************************************************)
(*  below is the final part of the declaration of "fun routes (pairs)"    *)
(*************************************************************************)


        fun f [] = []
          | f (p::ps) = ( clearFlags( !nrList);
                          travelNodes((n1 p), p, !(n2 p), []) @ f(ps) );
    in
        buildNodes (pairs);
        f ( !nrList)
    end;




(*

    Worst case complexity of function routes(input).

    Let there be N points (in some number of pairs) in input. Also assume
that
    there are few or none of duplicate pairs in input.  (Those can be
filtered
    out in quadratic time if required).


    1)  Pseudo-function giveRef is obviously linear in space and constant in
        time complexity with respect to the size of !nrList. In the worst
case
        !nrList will be of size N.

        Complexity of giveRef: time N, space constant.


    2)  The most complex (pseudo)-function called by buildNodes is giveRef
        (called N times).

        Complexity of buildNodes: time N^2, space N.


    3)  Pseudo-function travelNodes is the heart of the algorithm.  It is
        effectively a depth-first tree search function with certain cut-off
        parameters.

        travelNodes calls itself recursively to explore a sub-tree whenever
        it encounters a node it has not yet visited.  Obviously it can do so
a
        maximum of N times.  At a breadth search stage travelNodes may look
at a
        maximum of N nodes (the number of other nodes that a given node can
be
        connected to).  The "has been visited" flag is checked for every
node,
```

```
        which only requires constant time and space.

        Therefore travelNodes could perform at most N breadth searches on N
        points each.

        A breadth search on N points takes time N and space N, since this
        operation may be performed up to N times the complexity is:

        Complexity of buildNodes: time N^2, space N^2.


    4)  Functoin "routes" builds the node list once (time N^2, space N --
see
        "buildNodes").  Then for each of the N point in the node list
(!nList)
        it calls "clearFlags" (time N, space constant) and "travelNodes"
        (time N^2, space N^2).

        Therefore the complexity of "routes" is determined by the N calls to
        "travelNodes"

        Worst case complexity of "routes": time O(N^3), space O(n^3).

*);
```

8.3.2 ML tick 5

An answer from James Bowden

```
datatype univ =
  Int of int
| Real of real
| Complex of real*real ;

fun ineg [Int m] = Int (~m);
fun rneg [Real x] = Real (~x);
fun cneg [Complex (x,y)] = Complex(~x,~y);

fun iplus[Int x, Int y] = Int(x+y);
fun rplus[Real x, Real y] = Real(x+y);
fun cplus[Complex(x,y), Complex(u,v)] = Complex(x+u,y+v);

iplus [ Int 503, Int 40 ];
cneg [ Complex(1.3, 6.23) ];

val table =
  [ (("Int","~"), ineg),
    (("Real","~"), rneg),
    (("Complex","~"), cneg),
    (("Int","+"), iplus),
    (("Real","+"), rplus),
    (("Complex","+"), cplus) ];

fun typeof(Int( u)::us) = "Int"
  | typeof(Real (u)::us) = "Real"
  | typeof(Complex (u)::us) = "Complex";

fun lookup ((x,f)::ps, y: string*string) = if x=y then f else lookup(ps,y);
```

```
fun apply(opr:string, us) = lookup(table, (typeof(us),opr)) us;


fun negsum(x,y) = apply("~",[apply("+",x@y)]);
```

8.3.3 ML tick 6

An answer from Rujith de Silva

```
datatype stream = Item of int * (unit->stream);
fun cons (x,xs) = Item (x,xs);
(* I don't think we need to rename this function to 'cons' *)
fun head (Item (i,xf)) = i;
fun tail (Item (i,xf)) = xf();
fun makeints n = cons (n,fn()=>makeints (n+1));


 (* PROBLEM 2.  The function maps f xs ...  *)

 fun maps f xs = cons (f (head xs),fn()=>maps f (tail xs));


  (* PROBLEM 3.  A function nth (s,n) ...  *)

   fun nth (s,n) =
           if n=1 then head s
           else nth (tail s,n-1);

    fun squares p = maps (fn (x:int) => x*x) (makeints p);


    (* PROBLEM 4.  A function filters f xs ...  *)

     fun filters f xs =
         if f (head xs) then cons((head xs), fn()=> filters f (tail xs))
         else filters f (tail xs);

     (*The following would have done equally well....*)

     fun filters f xs =
         if f (head xs) then xs
         else filters f (tail xs);

     fun notdiv2or3 x =
             if x mod 2 = 0 orelse x mod 3 = 0 then false
             else true;


     (* PROBLEM 5.  A function primes n ...  *)
     (*
     fun primes n =
     *)

- fun primes xs =
 let val x = head xs
 in
```

```
  cons (x,fn()=>primes ( filters (fn p => p mod x <> 0) ( tail xs ) )) end;
```

Note that the above is essentially a one-line function to generate a stream of primes.

The following code to generate Fibonacci numbers incorporates a fairly unusual use of streams, in that the embedded function has embedded in it the last *two* numbers in the series.

```
datatype intstream = Item of int * (unit->intstream);

fun fibgen (a,b) = Item ( b, fn () => fibgen (b:int,a+b) );
```

The following [rather ugly] code merges two streams using the "diagonal" method of counting - the same method used to prove that the set of rational numbers in countable. Can you think of a better way to accomplish the merging of infinite streams?

```
datatype int2str = cons of (int*int)*(unit->int2str);

fun eval ( cons ( _, b ) ) = b ();

fun head ( Item ( b, _ ) ) = b;
fun tail ( Item ( _, x ) ) = x ();

fun merge2 ( x, y, nil, prev ) =
    let
        val hx = head x;
        val tx = tail x;
        val ty = tail y;
        val hty = head ty;
        val newlist = hx :: prev
    in
        cons ( (hx, hty), fn () => merge2 ( tx, ty, prev, newlist ) )
    end
  | merge2 ( x, y, p :: prev1, prev2 ) =
        let
            val hy = head y
        in
            cons ( ( p, hy ), fn () => merge2 ( x, y, prev1, prev2 ) )
        end;

fun merge ( x, y ) =
    let
        val hx = [ head x ]
    in
        merge2 ( tail x, y, hx, hx )
    end;
```

`merge` is the actual function that is called. This sets up the parameters for `merge2`, which in turn accomplishes the actual merging. Strictly speaking, the code for `merge2` should be declared within the function `merge`, but I thought this would only make the code even uglier.

Why doesn't the code for `merge2` work if `cons` is not used, so that `merge2` returns a type `'a` such that `'a = ((int*int)*unit->'a)` ? Surely, ML should be able to handle that and deduce the type of `'a`? It manages to evaluate the equally recursive definition of `int2str`.

An answer from Charles Bayliss

```
(*      ML TICK 6 SUBMISSION FROM C G BAYLIS              *)
(* Estimated time to complete: 2hrs.   Actual time 1hr 30 *)

(* 1.   [code taken from question *)

datatype stream = Item of int * (unit->stream);
fun cons (x,xs) = Item(x,xs);
fun head (Item(i,xf)) = i;
fun tail (Item(i,xf)) = xf();
fun makeints n = cons(n,fn()=> makeints(n+1));

fun maps f xs = cons(f (head xs), fn()=> maps f (tail xs));

(* nth takes a stream and an integer and returns the nth
   value in that stream *)

fun nth (s,1) = head(s)
  | nth (s,n) = nth (tail s,n-1);

(* drop removes the first n elements from a stream.
   it is used by filters *)

fun drop xs 0 = xs
  | drop xs 1 = tail xs
  | drop xs n = drop (tail xs) (n-1);

(* inside filters:
   filt acts as filters except it uses num as a counter for the
     depth within the input stream
   g returns the next number in the stream for which the
   function f returns true. *)

fun filters (f:(int -> bool)) xs =
  let fun filt (f:(int -> bool)) xsss num =
  let fun g xss number =
    if f (head xss) then number
      else g (tail xss) (number+1)
  in
    let val count = g xsss 0
    in
      cons(count+num, fn()=>filt f (drop xsss (count+1)) (num+count+1))
    end
  end
  in
    filt f xs 1
  end;

(* is used for question 4 *)

fun divtwothree x =
(x mod 2)<>0 andalso (x mod 3)<>0
```

```
FAM fam started on 15-Jan-1998 01:20:34
   (version 4.1.00 of Nov 17 1990)
Image file <Obey$Dir>.smlcore_ex
   (written on 03-Aug-1988 17:38:32 by FAM version 4.1.00)
[Loading]

Edinburgh Standard ML (core language)                (C) Edinburgh University


- use "ml:tick6";
> () : unit
[Opening ml:tick6]
> datatype stream = Item of int * (unit -> stream)
  con Item = fn : (int * (unit -> stream)) -> stream
> val cons = fn : (int * (unit -> stream)) -> stream
> val head = fn : stream -> int
> val tail = fn : stream -> stream
> val makeints = fn : int -> stream
> val maps = fn : (int -> int) -> (stream -> stream)
> val nth = fn : (stream * int) -> int
> val drop = fn : stream -> (int -> stream)
> val filters = fn : (int -> bool) -> (stream -> stream)
[Closing ml:tick6]

> val divtwothree = fn : int -> bool
-
- val squares=maps (fn x:int=>x*x) (makeints 1);
> val squares = Item (1,fn) : stream
- tail squares;
> Item (4,fn) : stream
- tail it;
> Item (9,fn) : stream
- tail it;
> Item (16,fn) : stream
-
- nth (squares,49);
> 2401 : int
-
- 49*49;
> 2401 : int
```

8.3.4 Tick 6*

An answer from Charles Bayliss

```
(*     ML TICK 6* SUBMISSION FROM C G BAYLIS              *)
(* Estimated time to complete: 2hrs.   Actual time 1hr 30 *)

(* Uses functions from Tick6 *)

fun map2 f xs ys = cons(f (head xs) (head ys), fn()=> map2 f (tail xs) (tail ys));

fun plus m n = m+n:int;
```

```
fun fibs () =
    cons (1, fn()=>
          cons(1, fn()=> map2 plus (fibs()) (tail(fibs())) ));

fun merge (xs,ys) =
  if (head xs)=(head ys) then
    cons(head ys,fn()=> merge (tail xs,tail ys))
   else
    if (head xs)>(head ys) then
      cons(head ys,fn()=> merge (xs,tail ys))
     else
      cons(head xs,fn()=> merge (tail xs,ys));

fun double x = 2*x:int;

fun triple x = 3*x:int;

(* ij answers question 4 *)

fun ij () =
    cons (1, fn()=> merge (maps double (ij()),maps triple (ij())));

fun xfive x = 5*x:int;

fun merge3 (xs,ys,zs) = merge(xs,merge (ys,zs));

(* ijk answers question 5 *)

fun ijk() =
    cons (1, fn()=> merge3 (maps double (ijk()),maps triple (ijk()),maps xfive (ijk()) ));

FAM fam started on 15-Jan-1998 01:27:01
   (version 4.1.00 of Nov 17 1990)
Image file <Obey$Dir>.smlcore_ex
   (written on 03-Aug-1988 17:38:32 by FAM version 4.1.00)
[Loading]

Edinburgh Standard ML (core language)                 (C) Edinburgh University


- use "ml:tick6";
> () : unit
[Opening ml:tick6]
> datatype stream = Item of int * (unit -> stream)
  con Item = fn : (int * (unit -> stream)) -> stream
> val cons = fn : (int * (unit -> stream)) -> stream
> val head = fn : stream -> int
> val tail = fn : stream -> stream
> val makeints = fn : int -> stream
> val maps = fn : (int -> int) -> (stream -> stream)
> val nth = fn : (stream * int) -> int
> val drop = fn : stream -> (int -> stream)
> val filters = fn : (int -> bool) -> (stream -> stream)
```

```
[Closing ml:tick6]
> val divtwothree = fn : int -> bool
-
- use "ml:tick6star";
> () : unit
[Opening ml:tick6star]
> val map2 = fn : (int -> (int -> int)) -> (stream -> (stream -> stream))
> val plus = fn : int -> (int -> int)
> val fibs = fn : unit -> stream
> val merge = fn : (stream * stream) -> stream
> val double = fn : int -> int
> val triple = fn : int -> int
> val ij = fn : unit -> stream
> val xfive = fn : int -> int
> val merge3 = fn : (stream * stream * stream) -> stream
> val ijk = fn : unit -> stream
[Closing ml:tick6star]
-
- nth (fibs (),15);
> 610 : int
-
- val squares = maps (fn x:int=>x*x) (makeints 1);
> val squares = Item (1,fn) : stream
-
- val threes = maps (fn x=>3*x) (makeints 1);
> val threes = Item (3,fn) : stream
-
- (* a stream of multiples of three and square numbers *);
-
- merge (threes,squares);
> Item (1,fn) : stream
- tail it;
> Item (3,fn) : stream
- tail it;
> Item (4,fn) : stream
- tail it;
> Item (6,fn) : stream
- tail it;
> Item (9,fn) : stream
- tail it;
> Item (12,fn) : stream
- tail it;
> Item (15,fn) : stream
- tail it;
> Item (16,fn) : stream
-
- ij();
> Item (1,fn) : stream
- tail it;
> Item (2,fn) : stream
- tail it;
> Item (3,fn) : stream
- tail it;
> Item (4,fn) : stream
```

```
- tail it;
> Item (6,fn) : stream
- tail it;
> Item (8,fn) : stream
- tail it;
> Item (9,fn) : stream
- tail it;
> Item (12,fn) : stream
- tail it;
> Item (16,fn) : stream
- tail it;
> Item (18,fn) : stream
- tail it;
> Item (24,fn) : stream
-
- ijk();
> Item (1,fn) : stream
- tail it;
> Item (2,fn) : stream
- tail it;
> Item (3,fn) : stream
- tail it;
> Item (4,fn) : stream
- tail it;
> Item (5,fn) : stream
- tail it;
> Item (6,fn) : stream
- tail it;
> Item (8,fn) : stream
- tail it;
> Item (9,fn) : stream
- tail it;
> Item (10,fn) : stream
- tail it;
> Item (12,fn) : stream
- tail it;
> Item (15,fn) : stream
- tail it;
> Item (16,fn) : stream
- tail it;
> Item (18,fn) : stream
- tail it;
> Item (20,fn) : stream
```

8.3.5 Larry's arithmetic

(* Two's complement binary arithmetic – could form the basis for numerals and arithmetic in theorem provers, by rewriting

The sign Plus stands for an infinite string of leading 0's; the sign Minus stands for an infinite string of leading 1's.

A number can have multiple representations, namely leading 0's with sign Plus and leading 1's with sign Minus. See `int_of_binary` for the numerical interpretation.

The representation expects that (m *mod* 2) is 0 or 1, even if m is negative; For instance, -5 *div* $2 = -3$ and -5 *mod* $2 = 1$; thus $-5 = (-3) * 2 + 1$.

Still needs an implementation of division!

```
System.Control.Print.printDepth := 350;
*)

(*Infix constructor and arithmetic operators*)
infix 5 %
infix 6 |+| |-|
infix 7 |*|;

(*Recursive datatype of binary integers*)
datatype bin = Plus | Minus | op % of bin * int;

(** Conversions between bin and int **)

fun int_of_binary Plus = 0
  | int_of_binary Minus = ~1
  | int_of_binary (w%b) = 2 * int_of_binary w + b;

fun binary_of_int 0 = Plus
  | binary_of_int ~1 = Minus
  | binary_of_int n = binary_of_int (n div 2) % (n mod 2);

(*** Addition ***)

(*Adding a carry to a number*)
fun carry_plus (0, w) = w
  | carry_plus (1, Plus) = Plus%1
  | carry_plus (1, Minus) = Plus
  | carry_plus (1, w%a) = carry_plus(a, w) % (1-a);

(*Adding a carry and ~1 to a number*)
fun carry_minus (1, w) = w
  | carry_minus (0, Plus) = Minus
  | carry_minus (0, Minus) = Minus%0
  | carry_minus (0, w%a) = carry_minus(a, w) % (1-a);

(*sum of two bins with carry*)
fun binsum (c, Plus, w) = carry_plus (c,w)
  | binsum (c, w, Plus) = carry_plus (c,w)
  | binsum (c, Minus, w) = carry_minus (c,w)
  | binsum (c, w, Minus) = carry_minus (c,w)
  | binsum (c, w%a, v%b) = binsum((a+b+c) div 2, w, v) % ((a+b+c) mod 2);

fun v1 |+| v2 = binsum(0, v1, v2);

(*** Subtraction ***)

(*Unary minus*)
fun neg Plus = Plus
  | neg Minus = Plus%1
  | neg (w%0) = neg(w) % 0
  | neg (w%1) = carry_minus (0, neg(w)%0);

fun v1 |-| v2 = v1 |+| (neg v2);
```

```
(*** Multiplication ***)

(*product of two bins*)
fun Plus  |*| _ = Plus
  | Minus |*| v = neg v
  | (w%0) |*| v = (w |*| v) % 0
  | (w%1) |*| v = ((w |*| v) % 0) |+| v;

(*** Testing ***)

(*tests addition*)
fun checksum m n =
    let val wm = binary_of_int m
        and wn = binary_of_int n
        val wsum = wm |+| wn
    in  if m+n = int_of_binary wsum then (wm, wn, wsum, m+n)
        else raise Match
    end;

fun bfact n = if n=0 then  Plus%1  else  (binary_of_int n) |*|  bfact(n-1);

bfact 5;
int_of_binary it;
bfact 69;
int_of_binary it;
```

8.4 Tripos Questions

8.4.1 1996:1:6

```
datatype LR = C1 of int * (unit->LR);

datatype LF = C2 of (int->int) * (unit->LF);

datatype LRs = C3 of LR * (unit->LRs);
```

Most of the functions are easy. The last one is something like:

```
fun newnumb fs = newnumb1 fs 1

and newnumb1 (C2(f,next)) i =
    C1( (f i + 1) mod 10), fn()=>newnumb1 (next()) (i+1);
```

1996:1:B5

```
fun rotations [] = []
  | rotations [x] = [[x]]
  | rotations (x::xs) =
       let fun swap ([],qs) = []
             | swap (p::ps,qs) = ((p::ps)@qs)::swap(ps,qs@[p])
       in swap (x::xs,[])
     end;
```

8.5 Miscellaneous titbits

Tal Kubo and Ravi Vakil have illuminated the behavior of the somewhat infamous recurrence relation $a(n) = a(a(n-1))+a(n-a(n-1))$ (with initial conditions $a(1) = a(2) = 1$) through the following combinatorial model: Enumerate the finite subsets of the positive integers in the order $\{\}$, $\{1\}$, $\{1,2\}$, $\{2\}$, $\{1,2,3\}$, $\{1,3\}$, $\{2,3\}$, $\{3\}$, $\{1,2,3,4\}$, $\{1,2,4\}$, $\{1,3,4\}$, $\{2,3,4\}$, $\{1,4\}$, $\{2,4\}$, $\{3,4\}$, $\{4\}$, $\{1,2,3,4,5\}$, ... so that one finite set of positive integers $A = \{a_1, a_2, \ldots, a_r\}$ is "prior to" another finite set $B = \{b_1, b_2, ..., b_s\}$ if $a_r < b_s$, or if $a_r = b_s$ and $r > s$, or if $a_r = b_s$ and $r = s$ but $a_{r-1} < b_{r-1}$, or if $a_r = b_s$ and $r = s$ and $a_{r-1} = b_{r-1}$ but $a_{r-2} < b_{r-2}$, etc. Associate the positive integer n with the nth set on this list. Then the operation $n \mapsto a(n)$, carried over to the domain of finite sets of positive integers, corresponds to the simple operation of deleting the smallest element of a set and reducing every remaining element by 1.

E.g.: $\{1,3,4\} \mapsto \{2,3\}$, and $a(11) = 7$.

(The sequence $a(n)$ is associated with the names of Hofstadter, Conway, and Mallows; the "infamy" comes from the fact that Conway accidentally offered a larger cash prize than he'd meant to. The work of Kubo and Vakil appeared in Discrete Mathematics, v.152 (1996) no.1-3,225-252.)

Question: In terms of this model, is there a direct combinatorial explanation for the recurrence relation? (The proof in the paper is not too unnatural, but it doesn't really explain what's going on.)

Jim Propp Department of Mathematics M.I.T.

Here's the ML version :

```
(*-------------------------------------------------------------------------*
 * Defines an interesting sequence, allegedly due to J. H. Conway.         *
 *-------------------------------------------------------------------------*)

fun Conway 1 = 1
  | Conway 2 = 1
  | Conway n = Conway(Conway(n-1)) + Conway(n - Conway(n-1));


(*-------------------------------------------------------------------------*
 * That's too slow, because it's always recomputing everything. We can use *
 * a cache of previously computed results to speed things up.              *
 *-------------------------------------------------------------------------*)
local fun nth 0 _ = raise Fail "Implementation Error"
        | nth 1 (x::_) = x
        | nth n (_::t) = nth (n-1) t
        | nth n []     = raise Fail "Cache miss"
      val cache = ref [1,1]
in
fun fastConway n =
  let val _ = cache := [1,1]
      fun Conway 1 = 1
        | Conway 2 = 1
        | Conway n =
             nth n (!cache)
             handle Fail "Cache miss"
                    => let val a = Conway(Conway(n-1))
                                 + Conway(n - Conway(n-1))
                           val _ = cache := !cache @ [a]
                       in a end
                  | e => raise e
  in Conway n
end end;

(*-------------------------------------------------------------------------*
 * Some test harness bits.                                                 *
 *-------------------------------------------------------------------------*)\

fun iter bot top =
   if bot > top then []
   else bot :: iter (bot+1) top;

val nums = iter 1 30;
val nums =
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
     22, 23, 24, 25, 26, 27, 28, 29, 30]

map Conway nums;  (* takes quite a long time *)
val it = [1, 1, 2, 2, 3, 4, 4, 4, 5, 6, 7, 7, 8, 8, 8, 8, 9, 10, 11, 12, 12,
          13, 14, 14, 15, 15, 15, 16, 16, 16]


(* Larger example. Note interesting behaviour around powers of 2.  *)
```

```
map fastConway (iter 1 128);

> val it =
    [1, 1, 2, 2, 3, 4, 4, 4, 5, 6, 7, 7, 8, 8, 8, 8, 9, 10, 11, 12, 12, 13, 14,
     14, 15, 15, 15, 16, 16, 16, 16, 16, 17, 18, 19, 20, 21, 21, 22, 23, 24,
     24, 25, 26, 26, 27, 27, 27, 28, 29, 29, 30, 30, 30, 31, 31, 31, 31, 32,
     32, 32, 32, 32, 32, 33, 34, 35, 36, 37, 38, 38, 39, 40, 41, 42, 42, 43,
     44, 45, 45, 46, 47, 47, 48, 48, 48, 49, 50, 51, 51, 52, 53, 53, 54, 54,
     54, 55, 56, 56, 57, 57, 57, 58, 58, 58, 58, 59, 60, 60, 61, 61, 61, 62,
     62, 62, 62, 63, 63, 63, 63, 63, 64, 64, 64, 64, 64, 64, 64]
    : int list
```

```
fun prefix [ ] [ ] = [ ]
 |  prefix (x::xs) (y::ys) = (x::y)::prefix xs ys;

fun sep [ ] = [[ ], [ ]]
 |  sep [x] = [[x], [ ]]
 |  sep (x::y::rest) = prefix [x,y] (sep rest);


fun merge [[ ], y] = y : int list
 |  merge [x,[ ]]  = x
 |  merge [x::xs, y::ys] =
      if x < y then x:: merge[xs,y::ys]
               else y:: merge[x::xs, ys];

fun s [ ] = [ ]
 |  s [x] = [x]
 |  s  x  =  merge (map s (sep x));
```

Something pretty from David Burleigh

```
(*  The following functions combine manage the dictionary.
  This includes getting user input into the right form (listwords),
  and udating the dictionary according to the links between the words. *)

(* Mutable list datatype *)

datatype 'a mlist = Nil
                  | Cons of 'a *'a mlist ref;

(* mlistof turns a list to a mutable list *)

fun mlistof [] = Nil
  | mlistof (x::xs) = Cons(x, ref (mlistof xs));

(* dic is the reference that stores the dictionary *)
```

```
val dic = ref (mlistof ([ref ("$",[])]));

(* mutapp puts its argument on the end of dic *)

fun mutapp x =
        let val r = !dic
        in dic:=Cons(x, ref r)
        end;

(* fst and snd functions that work on references *)

fun fstr r = let val (x,y) = !r in x end;
fun sndr r = let val (x,y) = !r in y end;

(* dlookup looks for a word in the dictionary
   and returns the list of words it links t0. *)

val blankd = ref ("", [ref ("",0)]);
fun dlookup dic str =
        case !dic of
            Nil => blankd
          | Cons(d,ds) => if fstr d = str then d
                else dlookup ds str;

(* wlookup looks for a word in a word list, returning it if found
  or a blank if not. *)

val blankw = ref ("",0);
fun wlookup dic str =
        case !dic of
            Nil => blankw
          | Cons(w,ws) => if fstr w = str then w
                else wlookup ws str;

(* sort uses a simple insertion sort to arrange the items in the
  word list by their associated number. *)

fun sort ([],lst) = lst
  | sort ((x::[]),lst) = x::lst
  | sort ((x::xs) :((string * int) ref) list ,lst) =
        if (sndr (hd xs)) > (sndr x) then sort (xs,x::lst)
        else sort ((tl xs),x::(hd xs)::lst);

(* wupdate takes two consecutive words, looks up the first in the
dictionary then
  looks up the second in the list that is returned.  If the second word
exists
  already then its number is incremented, otherwise it is added to the
list. *)

fun wupdate strs stra =
        let val dr = dlookup dic strs
            val wr = wlookup (ref (mlistof (sndr dr))) stra
```

```
        in
                if sndr wr <> 0
                then (wr:=(fstr wr, (sndr wr)+1);
                      dr:=(fstr dr, (sort (rev(sndr dr),[]))))
                else dr:=(fstr dr, ((sndr dr) @ [ref(stra,1)]))
        end;

(* dlookup looks up the first word in the dictionary, and creates a new
record if
  it doesn't exist, or calls wupdate if it does. It does not matter if the
second
  word is not yet known, because it will be added next time round. *)

fun dupdate strs stra =
        let val dr = dlookup dic strs
        in
                if fstr dr = ""
                then mutapp (ref(strs,[ref(stra,1)]))
                else wupdate strs stra
        end;

(* lupdate moves through a list of words calling dupdate. *)

fun lupdate [] = ()
    | lupdate (w::[]) = dupdate w " "
    | lupdate (w::ws) = (dupdate w (hd ws); lupdate ws);

(* listwords function turns a string into a list of words,
   removing all punctuation *)

fun listwords str =
        let val charlist = explode(str)
        in let fun findwords ([],word,words) = (implode(rev(word)))::words
                 | findwords (x::xs,word,words) =
                        if ord(x) < 48
                        orelse (ord(x) > 57 andalso ord(x) < 65)
                        orelse (ord(x) > 90 andalso ord(x) < 97) then
                                if word <> [] then

findwords(xs,[],(implode(rev(word)))::words)
                                else findwords(xs,word,words)
                        else findwords(xs,x::word,words)
                in "$"::rev(findwords (charlist,[],[]))
                end
        end;

(* ------------------------------------------------------------- *)
(* The following functions combine to produce a random response.
  the choice is weighted towards words that commonly link.  *)

(* random number generator *)

local val a = 16807.0 and m = 2147483647.0
in fun nextrandom seed =
```

```
        let val t = a * seed
        in t - m * real(floor(t/m)) end
   and truncto k r = 1 + floor((r/m) * (real k))
end;

(* rnd function returns a random integer between 1 and the argument.
   references used to track the seed value.                          *)

val rndseed = ref (nextrandom 1.0);
fun rnd (max :int) =
        (let val seed = !rndseed
        in rndseed := nextrandom seed
        end;
        let val newseed = !rndseed
        in truncto max newseed
        end);

(* this nth function looks at the number associated with each word,
 reducing its conter by that value, and returning the word whose number
 makes the counter less than 0.  When ued with random number and a list
 sorted according to number, this produces the neccessary weighted choice.
*)

exception error;
fun nth [] _ = raise error
    | nth (x::xs) n =
        if (n-(sndr x)) <= 0 then x
        else nth xs (n-(sndr x));

(* wlen sums the numbers associated with the words in the list *)

fun wlen lst =
let fun wl ([],n) = n
      | wl ((x::xs) :((string * int) ref) list,n) = wl (xs,n+(sndr x));
in wl (lst,1)
end;

(* randlist picks a random element from a weighted list. *)

fun randlist lst = nth lst ((rnd (wlen lst))-1);

(* makesent picks a word that it's argument links to and adds it to the
  sentence accumulator, which it retuns on reaching the end of a sentence.
*)

fun makesent (str,sent) =
        if str = " " then sent
        else let val wrd = fstr (randlist (sndr (dlookup dic str)))
             in makesent (wrd, sent ^ wrd ^ " ")
             end;

(*  These are the functions used to interface with the program.
  tt allows you to type a sentence which the program processes, but
     does not reply to.
```

```
  tb calls for a response without any input.
  tar allows input, and gives a reply. *)

fun tt str = (lupdate (listwords str));
fun tb () = makesent ("$","");
fun tar str = (tt str; tb());
```

Chapter 9

Miscellaneous

9.1 Combinatorics

9.2 Dynamic binding

Under the current dispensation if i say

```
letref fred = 3;;
let sam = fred * 10;;
let fred = 8
```

it then knows that `sam = 80`. I think what my student had in mind is where the concept of fancy-variable is not one such that :

> anything that calls a fancy variable gets updated whenever the fancy vbl does

but

> any fancy variable gets updated whenever anything it calls gets updated

so that a sample session would be

```
let fred = 3;;
letfancy sam = fred * 7;;
let fred = 5;;
```

and then it thinks that sam is 35

> Mike replies

That is called "dynamic binding" and is found in old versions of Lisp (e.g. Franz Lisp).

There is quite a large literature on the pros and cons of such binding. The main snag is that late definitions can mess up earlier ones.

Mike

P.S. . . . assignable variables in the old ML sense don't exist in Standard ML. As far I know all the ML courses taught in the Lab are the same dialect: Standard ML. I don't think Cambridge ML has ever been taught. In my view Standard ML's assignment contructs (references, weak type variables etc) are not as nice as the old constructs (though they are much more powerful) – but they seem to be what we are stuck with.

Re exams: there are important type-checking differences between letref-variables and Standard ML's references (e.g. the need to explicitly dereference):

```
  Old ML: x := x + 1

  New ML: x := !x + 1
```

For example:

```
- val x = ref 1;
val x = ref 1 : int ref
- x := x + 1;

std_in:3.1-3.10 Error: operator and operand don't agree (tycon mismatch)
  operator domain: int ref * int ref
  operand:         int ref * int
  in expression:
    + : overloaded (x,1)
std_in:3.8 Error: overloaded variable "+" not defined at type:
   int ref
-
```

If you want to follow this up you could look at Section 9.7 (entitled "Dynamic Binding") on Page 162 of "Programming Language Theory and its Implementation" by M.J.C. Gordon (i.e. me). Prentice-Hall.

Mike

Chapter 10

Discrete Maths

10.1 Notes for a course

This is the sequel to a course on baby number theory and digelec. It assumes ML as well

set and itself. Game-theoretic account of meaning. Also used in exegesis of hebrew texts (Henle)

Something about plants that flower every p years where p is a moderately large prime.

Trig examples of functions that one wants to store as embedded code not as a look-up table: quite easy to compute them quickly by using Chebyshev polys if you know in advance the precision you need.

Bit of relational algebra. Sets of ordered pairs with boolean operations plus composition and inversion. Composition of relations is matrix multiplication. Do not confuse complement, converse and mult inverse (ain't none) Exercise on enemy and friends, that sort of thing.

Set closed under an operation. $\mathbb{N}$ closed under + but not -. Then chat about closure operations. Convex sets.

excercise on idempotence (closures)

additive grp of reals iso to mult grp of +ve reals

10.1.1 Notation

$[1, n]$ is the set of (natural) numbers from 1 up to n. S is the successor relation on $\mathbb{N}$. ("`succ`" in 1a ML!)

10.2 Exercises

Starred exercises have model answers. The following relevant tripos questions also have model answers in this file.

Maths tripos questions
1988:6:9E, 1988:6:10E, 1995:5:4X,
Comp Sci tripos questions
1990:1:9, 1990:1:11, 1993:11:11, 1994:10:11, 1996:1:8

10.2.1 Elementary

"Elementary" doesn't mean 'easy'. It means that on the whole for the questions in this section no very sophisticated ideas are needed.

Exercises on (binary) relations

1. (Do not do more than a sample of the bits of this question: if you are making any mistakes they will always be the same mistakes, and there is no point in making the discovery more than once!)

   Given the operations of composition and union, express the following relations in terms of brother-of, sister-of, father-of, mother-of, son-of, daughter-of. (You may use your answers to earlier questions in answering later questions.)

   (a) i. parent-of
   ii. uncle-of
   iii. aunt-of
   iv. nephew-of
   v. niece-of
   vi. grandmother-of
   vii. grandfather-of
   viii. first-cousin-of

   You can also express some of the relations in the original list in terms of others by means of composition and union. Do so.

   (b) Do the same to include all the in-law and step relations, by adding spouse-of to the original list. This time you may use intersection and complement as well.

   (c) If the formalisation of "x is a parent of y" is "(father-of$(x, y) \vee$mother-of(x, y))" (i.e., use logical connectives not $\cup$ and $\cap$. You will also need to use quantifiers) what is the formalisation of the other relations in the preceding list? And for a bonus point, formalise "x is the double cousin of y".[1] Hint: might need new variables!

   (d) Using the above gadgetry, plus set inclusion ("$\subseteq$") formalise
   i. Every mother is a parent.
   ii. The enemy of [my] enemy is [my] friend
   iii. The enemy of my friend is my enemy.
   iv. The friend of my enemy is my enemy.
   v. no friend is an enemy

2. What is a graph? How many graphs are there on n vertices?

[1] Fred and Bert are double cousins if they are first-cousins in two different ways.

3. When doing this question remember that relations are relations-in-extension. You will find it helpful to think of a binary relation on n things as an $n \times n$ matrix whose entries are **true** or **false**.

   (a) How many binary relations are there on a set of size n?

   (b) How many of them are reflexive?

   (c) How many are fuzzies? (A *fuzzy* is a binary relation that is symmetric and reflexive)

   (d) How many of them are symmetrical?

   (e) How many of them are antisymmetrical?

   (f) How many are total ordersg?

   (g) How many are trichotomous? (A relation R on X is *trichotomous* iff $(\forall x, y \in X)(\langle x, y\rangle \in R \vee \langle y, x\rangle \in R \vee (x = y))$.

   (h) How many are antisymmetrical and trichotomous?

   (i) There are the same number of antisymmetrical relations as trichotomous. Prove this to be true without working out the precise number.

   (j) (for the thoughtful student) If you have done parts 3h and 3d correctly the answers will be the same. Is there a reason why they should be the same? (Revisit this later in connection with *natural bijections*.)

   (k) What is a partial order? *Do not answer the rest of this question.* How many partial orders are there on a set of size n?

   (l) *Do not answer this question.* A *strict* partial order is a transitive relation R satisfying
   $$(\forall x \forall y)(\neg R(x, y) \vee \neg R(y, x))$$
   How many strict partial orders are there on a set of size n?

   (m) Should the answers to the two previous questions be the same or different? Give reasons. (Compare this with your answer to question 3j above.)

   (n) An *extensional* relation on a set X is a binary relation R satisfying
   $$(\forall x, y)(x = y \longleftrightarrow (\forall z)(zRx \longleftrightarrow zRy))$$

      i. If R is extensional is R^{-1} also extensional?

      ii. How many extensional relations are there on a set of size n?

      iii. Show that the proportion of relations on a set with n members that are extensional tends to 1 as $n \to \infty$.

   (o) The five properties *symmetrical, transitive, reflexive, trichotomous, antisymmetrical* give rise to 2^5 possible combinations of properties. In each case find relations exhibiting the appropriate combination of properties or explain why there cannot be one. On second tho'rts do this only for a random sample of such combinations, or you will exhaust your supervisor's patience!

4. Can a relation be both symmetrical and antisymmetrical?

5. * Write out a formal proof that the intersection of two transitive relations is transitive.

6. * Let R be a relation on A. ('r', 's' and 't' denote the reflexive, symmetric and transitive closure operations respectively.)

(a) Prove that $rs(R) = sr(R)$.

(b) Does R transitive imply $s(R)$ transitive?

(c) Prove that $rt(R) = tr(R)$ and $st(R) \subseteq ts(R)$.

(d) If R is symmetrical must the transitive closure of R be symmetrical? Prove or give a counterexample.

7. Think of a binary relation R, and of its graph, which will be a directed graph $\langle V, E\rangle$. On any directed graph we can define a relation "I can get from vertex x to vertex y by following directed edges" which is certainly transitive, and we can pretend it is reflexive because after all we can get from a vertex to itself by just doing nothing at all. Do this to our graph $\langle V, E\rangle$, and call the resulting relation S. How do we describe S in terms of R?

8. * Show that—at least if $(\forall x)(\exists y)(\langle x, y\rangle \in R)$—$R \circ R^{-1}$ is a fuzzy. What about $R \cap R^{-1}$? What about $R \cup R^{-1}$?

9. * Given any relation R there is a least $T \supseteq R$ such that T is transitive, and a least $S \supseteq R$ such that S is symmetrical, namely the transitive and symmetric closures of R. Must there also be a unique maximal (aka **maximum**) $S \subseteq R$ such that S is transitive? And must there be a unique maximal (maximum) $S \subseteq R$ such that S is symmetrical? The answer to one of these last two questions is 'yes': find a cute formulation.

10. What are the transitive closures of the following relations on $\mathbb{N}$?

(a) $\{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \ldots\}$: i.e., $\{\langle n, n+1\rangle : n \in \mathbb{N}\}$,

(b) $\{\langle n, 2n\rangle : n \in \mathbb{N}\}$.

11. What is an antichain? Let D_n be the poset whose elements are the divisors of n, with $x \leq y$ if $x|y$. Find a maximum antichain in D_{216}.

12. * Define xRy on natural numbers by

xRy iff $x \leq y + 1$

What are the following relations?[2]

(a) $R \cap R^{-1}$

(b) $R \setminus R^{-1}$

(c) The transitive closure of the relation in (a)

(d) The transitive closure of the relation in (b)

13. * Are the two following conditions on partial orders equivalent?

(a) $(\forall xyz)(z < x \not\leq y \not\leq x \rightarrow z < y)$

(b) $(\forall xyz)(z > x \not\leq y \not\leq x \rightarrow z > y)$.

14. * Show that $R \subseteq S$ implies $R^{-1} \subseteq S^{-1}$

15. * Show that composition of relations is associative: *i.e.* if R, S and T are relations, show $(R \circ S) \circ T = R \circ (S \circ T)$.

[2]The structure $\langle \mathbb{N}, R\rangle$ is known to students of modal logic as the *Recession Frame*.

16. (a) Consider the following non-deterministic algorithm. A bag contains b black balls and w white balls. Two balls are removed. If they are both white, a white ball is replaced. If they are both black, an arbitrary and unspecified quantity of white balls from an inexhaustible supply is put in the bag. If one is black and one is white, the black ball is replaced. This process is repeated till the bag has only one ball. Show that the colour of the ball is determined by b and w alone and hence the algorithm determines a function of b and w.

    (b) * How can you be sure that the algorithm always terminates whatever you pluck out of the bag at each stage? *hint: think about the lexicographic order of* $\mathbb{N}^2$.

    (c) The purpose of this question was to make a point about lexicographic orders: in this case, about the order on $\mathbb{N} \times \mathbb{N}$. Check that you have really understood what is going on by rewriting the question for the scenario in which the balls come in three colours $\ldots k$ colours.

    (d) (abstruse: not for a first pass) Extend the product order of $\mathbb{N} \times \mathbb{N}$ by stipulating that $\langle x, y\rangle < \langle y, S(x)\rangle$ and taking the reflexive transitive closure. Write the result $\leq_{\mathcal{B}}$. Is $\leq_{\mathcal{B}}$ a total order? Define $\leq$ between finite subsets of $\mathbb{N} \times \mathbb{N}$ by $X \leq Y$ iff $(\forall x \in X)(\exists y \in Y)(x \leq_{\mathcal{B}} y)$. Is $\leq$ wellfounded?

17. Functions are just special kinds of relations, okay? What can you say about a function that is also a symmetrical relation? What about a function that is also a transitive relation? (That is, $f(x) = y \wedge f(y) = z \rightarrow f(x) = z$) Embarrass your supervisor by demanding explanations of the words *involution* and *idempotent*.

18. Let $K = \lambda x.(\lambda y.x)$. Evaluate $K8$, $K(K8)$ and $(KK)8$.

19. This question concerns *(binary) games of length* n. For each set $X \subseteq \{0, 1, 2, 3, 4 \ldots 2^n - 1\}$ we have a game G_x of length n; there are two players `I` and `II`; they play by writing down either 0 or 1, *ad libitum*, alternating (with `I` starting), and carry on until n 0s and 1s have been written down (that's why the game is of length n); the result is a a string of n 0s and 1s, which is to say, a binary number $k < 2^n$. The rule is that `I` wins iff $k \in X$. Show that for every $n \in \mathbb{N}$ and for every game of length n, one of the two players must have a winning strategy. How many (binary) games of length n are there? (Easy) Let $\texttt{II}_n$ be the proportion of these games for which player `II` has a winning strategy: what is the limit of $\texttt{II}_n$ as n gets large? (Easy).

20. What is a wellordering? What is an initial segment of an ordering? (If you don't know what a **chain** in a poset is you probably won't know what an initial segment in a total ordering is either.) If $\langle X, \leq\rangle$ is a total order, then a *suborder* of it is a subset $X' \subseteq X$ ordered by the obvious restriction of $\leq$. Prove that $\langle X, \leq\rangle$ is a wellordering if every suborder of it is isomorphic to an initial segment of it. (The converse is also true but involves more work.)

21. Consider the argument: "If Anna can cancan or Kant can't cant, then Greville will cavil vilely. If Greville will cavil vilely, Will won't want. But Will will want. Therefore, Kant can cant." By rewriting the statement in terms of four Boolean variables, show it is tautologous and hence a valid argument. (*There are loads of similar exercises in any number of introductory logic books. Try, for example, Lewis Carroll*, Symbolic Logic.)

22. Bracket '$[(a \rightarrow b) \vee (a \rightarrow d)] \rightarrow (b \vee d) \longleftrightarrow a \vee b \vee d$' and test all versions for validity.

23. * "Everybody loves my baby, but my baby loves nobody but me". Demonstrate that my baby = me.

24. "Brothers and sisters have I none, but this man's father is my father's son" *What?!*

10.2.2 Slightly less elementary

1. * Show that $\bigcup_{n\in\mathbb{N}} R^n$ is the smallest transitive relation extending R.

2. $t(R)$ is the transitive closure of R.[3]

   (a) * Give an example of a relation R on a set of size n for which $t(R) \neq R^1 \cup R^2 \cup \ldots \cup R^{n-1}$.

   (b) Give an example of a set and a relation on that set for which $t(R) \neq R^1 \cup R^2 \cup \ldots \cup R^n$ for any finite n.

   (c) If R is reflexive then $t(R)$ is clearly the reflexive transitive closure of R (often called just the transitive closure): if you are not happy about this, attempt to write out a proof.

   (d) Find an example of an *irreflexive*[4] relation R on a set such that $t(R)$ is indeed the reflexive transitive closure of R.

3. Think about $\mathbb{N}$ and S. What is the transitive closure of S? For integers n and m when do we have $(S^n)^* \subseteq (S^m)^*$? When do we have $(S^n \cup (S^n)^{-1} \cup S^m \cup (S^m)^{-1})^* = (S \cup S^{-1})^*$?

4. * Show that the smallest equivalence relation containing the two equivalence relations R and S is $t(R \cup S)$.

5. If $R \subseteq X \times X$ is a fuzzy on X, is there a largest equivalence relation on X that $\subseteq R$? Is there a smallest equivalence relation on X that $\supseteq R$?

6. (a) Suppose that for each $n \in \mathbb{N}$, R_n is a transitive relation on a (presumably infinite) set X. Suppose further that for all n, $R_n \subseteq R_{n+1}$. Let R_∞ be $\bigcup_{n\in\mathbb{N}} R_n$, the union of all the R_n. Prove that R_∞ is also transitive.

   (b) Give an example to show that the union of two transitive relations is not always transitive.

7. For all the following choices of allegations, prove the strongest of the correct options; explain why the other correct options are not best possible and find counterexample to the incorrect ones. If you find you are doing them with consummate ease, break off and do something else instead.

   (a) An intersection of a fuzzy and an equivalence relation is (i) an equivalence relation (ii) a fuzzy (ii) neither

   (b) A union of a fuzzy and an equivalence relation is (i) an equivalence relation (ii) a fuzzy (iii) neither

   (c) An intersection of two fuzzies is (i) an equivalence relation (ii) a fuzzy (iii) neither

   (d) An intersection of the complement of a fuzzy and an equivalence relation is (i) an equivalence relation (ii) a fuzzy (iii) neither

[3] Misleadingly people often use the expression "transitive closure of R" to mean the transitive reflexive closure of R.

[4] You don't know what 'irreflexive' means? There are only two things it can possibly be, so what are they? Answer this question for *both* versions! That'll teach you ask silly questions!

(e) An intersection of a fuzzy and the complement of an equivalence relation is (i) an equivalence relation (ii) a fuzzy (iii) neither

(f) A union of a fuzzy and the complement of an equivalence relation is (i) an equivalence relation (ii) a fuzzy (iii) neither

(g) An intersection of a fuzzy and the complement of a fuzzy is (i) an equivalence relation (ii) a fuzzy (iii) neither

(h) An intersection of the complement of a fuzzy and the complement of an equivalence relation is (i) an equivalence relation (ii) a fuzzy (iii) neither

(i) A union of two fuzzies is (i) an equivalence relation (ii) a fuzzy (iii) neither.

8. A PER ('*P*artial *E*quivalence *R*elation') is a binary relation that is symmetrical and transitive. Is the complement of a PER a fuzzy? Is the complement of a fuzzy a PER? In each case, if it is false, find sensible conditions to put on the antecedents that would make it true.

9. Let $<$ be a transitive relation on a set X. Consider the two relations (i) $\{\langle x, y\rangle : (x \in X) \wedge (y \in X) \wedge (x < y) \wedge (y < x)\}$ and (ii) $\{\langle x, y\rangle : (x \in X) \wedge (y \in X) \wedge (x \not< y) \wedge (y \not< x)\}$.

   (a) Are either of these fuzzies, or equivalence relations?

   (b) If one of these isn't a fuzzy, but "ought to be", what was the correct definition?

   (c) If the relation in (i) was an equivalence relation, what sort of relation does $<$ induce on the equivalence classes? Why is the result a mess? What extra condition or conditions should i have put on $<$ to start with to prevent this mess occurring?

   (d) If (the correct definition of) relation (ii) is an equivalence relation, what can we say about the quotient?

10. Explain how to find the two greatest numbers from a set of n numbers by making at most $n + \lfloor \log_2 n \rfloor - 2$ comparisons. Can it be done with fewer? How about the 3 biggest numbers? The k biggest numbers, for other values of k? What happens to your answer as k gets bigger and bigger and approaches n?

11. * Show that the largest and smallest elements of a totally ordered set with n elements can be found with $\lceil 3n/2 \rceil - 1$ comparisons if n is odd, and $3n/2 - 2$ comparisons if n is even.

12. Construct *natural* bijections between the following pairs of sets. (For the purposes of this exercise a natural map is (expressed by) a closed λ-term; a natural bijection is (expressed by) a closed λ term (L, say) with an inverse L'. That is to say, both `compose`(L, L') and `compose`(L', L) simplify to $\lambda x.x$. Alternatively, a natural function is one you can write an `ML` program for. If you want to think more about what a natural bijection is, look at your earlier answers to the questions: If A is a set with n members, how many symmetrical relations are there on A, and how many antisymmetrical trichotomous relations are there on A? The answers to these two questions are the same, but there doesn't seem to be any 'obvious' or 'natural' bijection between the set of symmetrical relations on A and the set of antisymmetrical trichotomous relations on A.) You will need to assume the existence of primitive pairing and unpairing functions which you might want to write as '`fst`', '`snd`' and $\langle x, y\rangle$

$A \to (B \to C)$ and $B \to (A \to C)$;
$A \times B$ and $B \times A$;
$A \to (B \times C)$ and $(A \to B) \times (A \to C)$;
$(A \times B) \to C$ and $A \to (B \to C)$;

You may wish to try the following pairs too, but only once you have done the `ML` machinery for disjoint unions of types:

$(A \to C) \times (B \to C)$ and $(A + B) \to C$;
$A + (B + C)$ and $(A + B) + C$;
$A \times (B + C)$ and $(A \times B) + (A \times C)$.

Let Z be a set with only one element. Find a natural bijection between $(Y + Z)^X$ and the set of partial functions from X to Y.

Find natural functions[5]

(i) from A into $B \to A$;
(ii) from A into $(A \to B) \to B$;
(iii) from $A \to (B \to C)$ into $(A \to B) \to (A \to C)$;
(iv) from $(((A \to B) \to B) \to B)$ into $A \to B$. (This one is hard: you will need your answer to (ii))
(v) from $(A \to B) \to A$ into $(A \to B) \to B$.

(it might help to think of these as invitations to write `ML` code of types `'a -> 'b -> 'a`, `'a -> ('a -> 'b) -> 'a` etc.)

13. What is a fixed point? What is a fixpoint combinator? Let T be your answer to the last bit of the preceding question. (So T is a natural function from $(A \to B) \to A$ into $(A \to B) \to B$.) Show that something is a fixpoint combinator iff it is a fixed point for T.

14. Let $P = \lambda G.(\lambda g.G(gg))(\lambda g.G(gg))$. Show that P is a fixpoint combinator. Why is it not typed? After all, T was typed!

15. Give `ML` code for a higher-order function `metafact` such that any fixed point for `metafact` will turn out to be good old `fact`. Do the same for something tedious like `fibo`. Delight your supervisor by finding, for other recursively defined functions, higher-order functions for which they are fixed points.

16. Think of '$\to$' as implication: is $(((p \to q) \to p) \to p)$ a truth-table tautology? Now think of $\to$ as "set of all functions from ...". Is there a natural map from $((p \to q) \to p)$ to p? (Very Hard!)[6]

17. * Solve

$$x^{x^{x^{x^{x^{\cdots}}}}} = 2$$

and comment on the notation. Then think about

$$x^{x^{x^{x^{x^{\cdots}}}}} = 4.$$

[5]These do not have to be either injective or surjective. They only have to be functions.

[6]Hints: Suppose p has five members and q is a subset of p with two members. Use the pigeonhole principle to find a map $((p \to q) \to p)$. Reflect on how **natural** maps must interact with permutations. See Dana Scott's article "Semantic archæology" in Harman and Davidson (eds.) *Semantics of Natural language* Reidel 197?.

18. Prove that $2^n - 1$ moves are sufficient to solve the Towers of Hanoi problem.

19. The fellows of Porterhouse ring each other up every sunday to catch up on the last week's gossip. Each fellow passes on (in all subsequent calls that morning) all the gossip (s)he has picked up, so there is no need for each fellow to ring every other fellow directly. How many calls are needed for every fellow to have aquired every other fellow's gossip?

20. A *triomino* is an L-shaped pattern made from three square tiles. A $2^k \times 2^k$ chessboard, whose squares are the same size as the tiles, has one of its squares painted puce. Show that the chessboard can be covered with triominoes so that only the puce square is exposed.

21. Is it possible to tile a standard (8×8) chessboard with thirty-one 2×1 rectangles (dominoes) to leave two diagonally opposite corner squares uncovered?

22. * Let $k \in \mathbb{N}$ and let $\mathcal{F}$ be a family of finite sets closed under symmetric difference, such that each set in $\mathcal{F}$ has at most k elements. How big is $\bigcup \mathcal{F}$? How big is $\mathcal{F}$?

23. Fix a set X. If π_1 and π_2 are partitions of it, we say π_1 *refines* π_2 if every piece of π_1 is a subset of a piece of π_2. What properties from the usual catalogue (transitivity, symmetry, etc.) does this relation between partitions of X have?

24. Let X be a set, and R the refinement relation on partitions of X. Let $\Pi(X)$ be the set of partitions. Why is it obvious that in general the structure $\langle \Pi(X), R \rangle$ is not a boolean algebra?

Boolean Algebra

1. Write down the truth tables for the 16 functions $\{T, \bot\}^2 \to \{T, \bot\}$, and give them sensible names (such as $\wedge, \vee, \to$, `NOR`, `NAND`). Which of these functions `splat` that you have identified have the feature that if p `splat` q and p both hold, then so does q? Why are we interested in only one of them?

2. (a) Show that `NAND` and `NOR` cannot be constructed by using $\wedge$ and $\vee$ and $\to$ alone

   (b) Show that none of `NAND`, `NOR`, $\to$, $\wedge$, $\vee$ can be constructed by using `XOR` alone. (hard)

   (c) Show that `XOR` and $\longleftrightarrow$ and $\neg$ cannot be defined from $\vee$ and $\wedge$ alone.

   (d) *(for enthusiasts only)* Can $\wedge$ and $\vee$ be defined in terms of $\longleftrightarrow$ and $\to$?

   (e) *(for enthusiasts only)* Show that all connectives can be defined in terms of `XOR` and $\to$.

   (f) A *monotone* propositional function is one that will output 1 if all its inputs are 1. Show that no nonmonotone function can be defined in terms of any number of monotone functions. (easy)

3. What is a boolean algebra? Find a natural partial order on the set of functions from question 1 that makes them into a boolean algebra.

4. How many truth-functions of three propositional letters are there? Of four? Of n?

5. Prove that $\mathcal{P}([0, 2]$ and $\{T, \bot\}^3$ are isomorphic posets.

Generating functions etc.

1. Let u_n be the number of strings in $\{0,1,2\}^n$ with no two consecutive 1's. Show $u_n = 2u_{n-1} + 2u_{n-2}$, and deduce $u_n = \frac{1}{4\sqrt{3}}[(1+\sqrt{3})^{n+2} - (1-\sqrt{3})^{n+2}]$.

2. Let m_n be the number of ways to obtain the product of n numbers by bracketing. (For example, $((ab)c)d$, $(ab)(cd)$, $(a(bc))d$, $a((bc)d)$ and $a(b(cd))$ show $m_4 = 5$.) Prove $m_n = \frac{1}{n}\binom{2n-2}{n-1}$.

3. Prove that $\mathbb{N} \times \mathbb{N}$, with the lexicographical order, is well-ordered, and that $\mathbb{N} \times \mathbb{N}$ with the product order has no infinite antichain.

4. Say $n \in m$ (where $n, m \in \mathbb{N}$) if the nth bit of m is 1. $n \subseteq m$ is defined in terms of this in the obvious way. Prove that $n \subseteq m$ iff $\binom{m}{n}$ is odd. (Hint: use the fact that $\binom{m+1}{n+1} = \binom{m}{n} + \binom{m}{n+1}$.)

5. Let p_n be the number of ways to add $n-3$ non-crossing diagonals to a polygon with n sides, thus splitting it into $n-2$ triangles. So $p_3 = 1$, $p_4 = 2$, $p_5 = 5$, and we define $p_2 = 1$. Show that

$$p_n = p_2 p_{n-1} + p_3 p_{n-2} + \ldots + p_{n-1} p_2 \quad \text{for} \quad n \geq 3\,,$$

6. and hence evaluate p_n.

7. A question on generating functions which will keep you out of mischief for an entire afternoon![7] Let A_n be the number of ways of ordering the numbers 1 to n such that each number is either bigger than (or smaller than) *both* its neighbours. ("zigzag permutations"). Find a recurrence relation for $(A_n/2)$. (*Hint* Think about how many zigzag permutations of $[1,n]$ there are where n appears in the rth place.) Further hints: you will have to divide the nth term by $n!$ and solve a (fairly simple) differential equation.

8. What can you say about

$$q_0 =: 1; q_{n+1} =: 1 - e^{-q_n}?$$

Truth-definitions

An `ML` question which will prepare you for the 1b courses entitled "Logic and Proof" and "Semantics". You should make a serious attempt at—at the very least—the first part of this question. The fourth part is the hardest part and provides a serious work-out to prepare you for the semantics course. Parts 2 and 3 are less central, but are educational. *If you are a 1b student treating this as revision you should be able to do all these questions.*

Propositional Logic	Predicate (first-order) Logic
A recursive datatype of formulæ	A recursive datatype of formulæ
	An interpretation $\mathcal{I}$ is a domain $\mathcal{D}$ with: for each n-place predicate letter F a subset $\mathcal{I}`F$ of $\mathcal{D}^n$; for each $n$-ary function letter $f$ a function $\mathcal{I}`f$ from $\mathcal{D}^n \to \mathcal{D}$. (Also constants).
`states`: `literals` → `bool`. A (recursively defined) satisfaction relation `SAT`:`states`×`fmla`→`bool`	(Fix $\mathcal{I}$ then) `states`: $vbls \to \mathcal{D}$; a recursively defined satisfaction function: $sat_{\mathcal{I}}$: formulæ × `states` → `bool`
A formula ϕ is **valid** iff for all `states` v, `SAT`(v, ϕ) =`true`.	ϕ is **true** in an interpretation $\mathcal{I}$ iff for all states v, $sat_{\mathcal{I}}(\phi, v)$ =`true`. ϕ is **valid** iff it is true in all interpretations.

[7] This comes from a book called "100 great puzzles in maths" or some such title: the author's name is Dörrie, it is published by Dover, and there is a copy in the DPMMS library. This is problem 16 on p 64.

1. Write `ML` code to implement the left-hand column. If you are completely happy with your answer to this you should skip the next two questions of this section.

2. (*For enthusiasts*). Expand the propositional language by adding a new unary connective, written '$\Box$'. The recursive definition of `SAT` for the language with this extra constructor has the following additional clause:

   > if s is a formula of the extended language and v is a state then $\texttt{SAT}(v, \Box s) = 1$ iff for all states v' we have $\texttt{SAT}(v', s) = 1$

   Then redo the first question with this added complication.

3. (*For enthusiasts*). Complicate further the construction of the preceding question by altering the recursive step for $\Box$ as follows. Accept as a new input a (binary) relation R between states (presumably presented as a list of pairs, tho' there may be prettier ways of doing it). The new clause is then:

   > if t is a formula of the form $\Box s$ and v is a state then $\texttt{SAT}(v, t) = 1$ iff for all states v' such that $v'\ R\ v$ we have $\texttt{SAT}(v', s) = 1$

4. Declare a recursive datatype which is the language of partial order. That is to say you have a set of variables, quantifiers, connectives etc., and two predicate letters '$\leq$ and '$=$'. Fix an interpretation of it, possibly the `ML` type `int`. Implement as much as you can of the apparatus of states, truth etc.

5. Declare a recursive datatype which is the language of fields. That is to say you have a set of variables, quantifiers, connectives etc.; two constants '0' and '1'; a binary predicate letter '=' and two function symbols, '+' and '$\times$'. Fix an interpretation of it, for example the natural numbers below 17. Implement as much as you can of the apparatus of states, truth etc. You should be able to write code that will accept as input a formula in the language of fields and evaluate to `true` or `false` depending on what happens in the naturals mod 17.[8]

In the last two questions you could make life easier for yourself (but less natural) by assuming that the language has only finitely many individual variables. This would enable you, for example (by somehow generating all the possible states, since there are now only finitely many of them) to verify that the naturals as an ordered set are a model for the theory of total order, and that the naturals mod 17 are a model for the theory of fields.

When you have done this ask the system minders or any member of the hvg group about how to run *HOL* on the machines available to you. In *HOL* is a dialect of *ML* in which all the needed datatypes are predefined.

Other logic: for 1b revision, mainly

1. π and e are transcendental. By considering the equation

   $$x^2 - (\pi + e)x + \pi e = 0$$

   prove a trivial but amusing fact. (If you cannot see what do do, read the footnote for a HINT). [9] What have you proved? Is your proof constructive? If not, does this give rise to a constructive proof of something else?

[8] It won't run very fast!

[9] At least one of $\pi + e$ and πe must be transcendental.

2. The uniqueness quantifier $\exists!x$' is read as "There is precisely one x such that ...". Show how to express the uniqueness quantifier in terms of the old quantifiers $\exists$ and $\forall$ (and $=$).

   (a) Find an example to show that $(\exists!x\exists!y)\phi(x,y)$ is not always the same as $(\exists!y\exists!x)\phi(x,y)$

   (b) Is the conjunction of $\exists!x\phi(x)$ and $\exists!y\psi(y)$ equivalent to something of the form $\exists!x\exists!y\ldots$?

3. Explain what a *model* of a sentence is. If Φ is a sentence the **spectrum** of Φ is the set of $n \in \mathbb{N}$ such that Φ has a model of size n. Is every spectrum recursive? Use a diagonal argument to find a recursive set that is not a spectrum.

4. You are the computer officer in charge of a system that is not secure, in the sense that it is possible to write viruses for it *ad lib*. (A virus is something that corrupts the operating system). Use a diagonal argument to show that any program running under this system that accepts a body of code as input and outputs 0 if the input is a virus and 1 otherwise must itself be a virus.

Propositions as types

Check that you know what is meant by "natural deduction". In this section, Greek letters will range over expressions built up from 'A', 'B', etc. by putting '$\rightarrow$' between such expressions. Thus they can be read indifferently as propositional formulæ or as types. Call these chaps *formulæ*. Attempt these in connection with question 12 from section 10.2.1.

1. Prove that if α is a formula such that there is a closed lambda term of type α then there is a natural deduction proof of α. And conversely!

2. If $\mathcal{D}$ is some natural deduction with conclusion α and premisses $\beta_1 \ldots \beta_n$ show that any valuation defined on the propositional letters in $\beta_1 \ldots \beta_n$ and α that makes all the $\beta_1 \ldots \beta_n$ true must also make α true too.

3. The relation $\neg\neg(x = y)$ is (intuitionistically) distinct from the relation $x = y$. Prove that it is a fuzzy. Is it an equivalence relation? Prove it or explain why you think it isn't

4. Find a natural deduction proof of
   $$(X \rightarrow (Y \rightarrow (Z \rightarrow W))) \rightarrow ((X \rightarrow (Y \rightarrow Z)) \rightarrow ((X \rightarrow Y) \rightarrow (X \rightarrow W)))$$
   and a λ-term to go with it.

Horn clauses

1. What is a horn clause? What is an intersection-closed property of relations?[10] Let $\phi(\vec{x})$ be a horn clause (in which 'R' appears and the $\vec{x}$ range over the domain of R). Show that the property $\forall\vec{x}(\phi(\vec{x}))$ is intersection closed. (The converse is also true but do not attempt to prove it!)

2. Let I be an index set, and for each $i \in I$, P_i is a person, with an associated set of beliefs, B_i. We assume (unrealistically) that each B_i is deductively closed and consistent. Show that $\bigcap_{i\in I} B_i$ is deductively closed and consistent. What about the set of all propositions p such that p is believed by a majority of

[10] A horn clause is a formula of the kind $\bigwedge_{i\in I} \psi_i \rightarrow \phi$ where ϕ and all the ψ_i are atomic. $F()$ is an intersection-closed property of relations if an intersection of any number of relations that have property F also has property F.

people? (You may assume I is finite in this case, otherwise it doesn't make sense). What about the set of things believed by all but finitely many of the P_i? (You may assume I is *in*finite in this case, otherwise it doesn't make sense).[11]

3. We are given a set $\mathcal{L}$ of literals. We are also given a subset $K_0 \subseteq \mathcal{L}$. ('$K$' for '$K$nown'.) Also a set C_0 ('C' for 'Conditionals) of formulaæ of the kind

$$(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \rightarrow q$$

If we are given two such sets, of literals and of conditionals, we can get a new set of Known literals by adding to K_0 any q that is the consequent of a conditional all of whose antecedents are in K_0. Of course we can then throw away that conditional.

(a) Turn this into a precise algorithm that will tell us, given K_0, C_0 and a candidate literal q, whether or not q can be deduced from K_0 and C_0. By coding this algorithm in `ML`, or by otherwise concentrating the mind, determine how efficient it is.

(b) What difference does it make to the implementation of your algorithm if the conditionals are of the form

$$p_1 \rightarrow (p_2 \rightarrow (p_3 \rightarrow \ldots q) \ldots)?$$

(c) What happens to your algorithm if Conditionals are allowed to be of the (more complicated) form:

$$(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \rightarrow (q_1 \vee q_2)?$$

Can anything be saved?

(d) Define a quasi-order (remember what a quasi-order is?[12]) on $\mathcal{L}$ by setting $p \; R \; q$ if there is a conditional in C_0 which has q as its consequent and p as one of its antecedents, and letting $<$ be the transitive closure of R. Is $<$ reflexive? Irreflexive? Antisymmetrical? What happens if $p < p$? What happens if $(p < q) \wedge (q < p)$?

[11] What about the set of propositions believed by an even number of people?

[12] And don't lose sleep over the reflexivity condition: we can add lots of silly clauses like $p \rightarrow p$ at no cost!

10.3 Answers

1988:6:9E (maths tripos)

The Master asked $2n+1$ people and got $2n+1$ different answers. Since the largest possible answer is $2n$ and the smallest is 0, there are in fact precisely $2n+1$ possible answers and that means he has got every possible answer from 0 up to $2n$ inclusive.

Think about the person who shook $2n$ hands. This person shook hands with everyone that they possibly could shake hands with : that is to say everyone except their spouse. So everybody except their spouse shook at least one hand. So their spouse shook no hands at all. Thus the person who shook $2n$ hands and the person who shook 0 hands are married. Henceforth disregard these two people and their handshakes and run the same argument to show that the person who shook $2n-1$ hands and the person who shook 1 hands are married. And so on.

Where does this get us? It tells us, after n iterations, that the person who shook $n+1$ hands and the person who shook $n-1$ hands are married. So what about the person who shook n hands, the odd man out? Well, it must be the odd *woman* out, because the only person of whom the Master asks this question who isn't married to another person of whom the Master asks this question is his wife.

Let's name people (other than the Master) with the number of hands they shook. (This is ok since they all shook different numbers of hands.) $2n$ didn't shake hands with its spouse, or itself, and there are only $2n$ people left, so it must have shaken hands with the Master. Correspondingly 0 didn't shake hands with anyone at all, so it certainly didn't shake hands with the Master. We continue reasoning in this way, about $2n-1$ and 1. $2n-1$ didn't shake hands with itself or its spouse or with 0, and that leaves only $2n-1$ people for it to shake hands with and since it shook $2n-1$ hands it must have shaken all of them, so in particular it must have shaken hands with the Master. Did 1 shake hands with the Master? No, because 1 shook only one hand, and that must have been $2n-1$'s. And so on. The people who shook the Master's hand were $2n$, $2n-1$, $2n-2 \ldots n+1$ and the people who didn't were 1, 2, 3, $\ldots n-1$. And of course, the Master's wife. So he shook n hands.

1988:6:10E (maths tripos)

> Let R be a relation on a set X. Define the reflexive, symmetric and transitive closures $r(R)$, $s(R)$ and $t(R)$ of R. Let Δ be the relation $\{\langle x,x\rangle : x \in X\}$. Prove that
>
> 1. $R \circ \Delta = R$
> 2. $(R \cup \Delta)^n = \Delta \cup (\bigcup_{i \le n} R^i)$ for $n \ge 1$
> 3. $tr(R) = rt(R)$.
>
> Show also that $st(R) \subseteq ts(R)$. If $X = \mathbb{N}$ and $R = \Delta \cup \{\langle x,y\rangle : y = px$ for some prime $p\}$ describe $st(R)$ and $ts(R)$.

The reflexive (symmetric, transitive) closure of R is the intersection of all reflexive (symmetric, transitive) relations of which R is a subset.

1. $R\Delta$ is R composed with the identity relation. x is related to y by R-composed-with-S if there is z such that x is related to z by R, and z is related to y by S. Thus $R\Delta = R$. (I would normally prefer to write '$R \circ \Delta$' here, using a standard notation for composition of relations: '$\circ$')

2. It is probably easiest to do this by induction on n. Clearly this is true for $n = 1$, since the two sides are identical in that case. Suppose it is true for

$n = k$.

$$(R \cup \Delta)^k = \Delta \cup (\bigcup_{1 \leq i \leq k} R^i)$$

$(R \cup \Delta)^{k+1} = (R \cup \Delta)^k \circ (R \cup \Delta)$. By induction hypothesis this is

$$(\Delta \cup (\bigcup_{1 \leq i \leq k} R^i)) \circ (R \cup \Delta)$$

Now $(A \cup B) \circ (C \cup D)$ is clearly $(A \circ C) \cup (A \circ D) \cup (B \circ C) \cup (B \circ D)$ and applying this here we get

$$(\Delta \circ R) \cup (\Delta \circ \Delta) \cup ((\bigcup_{1 \leq i \leq k} R^i) \circ R) \cup ((\bigcup_{1 \leq i \leq k} R^i) \circ \Delta)$$

Now $\Delta \circ R$ is R; $\Delta \circ \Delta$ is Δ; $(\bigcup_{1 \leq i \leq k} R^i) \circ \Delta$ is $\bigcup_{1 \leq i \leq k} R^i$ and $(\bigcup_{1 \leq i \leq k} R^i) \circ R$ is $(\bigcup_{1 < i \leq k+1} R^i)$ so we get

$$R \cup \Delta \cup (\bigcup_{1 < i \leq k+1} R^i) \cup (\bigcup_{i \leq k} R^i)$$

which is

$$\Delta \cup \bigcup_{1 \leq i \leq k+1} R^i$$

3. The transitive closure of the reflexive closure of R is the transitive closure of $R \cup \Delta$ which is $\bigcup_{n \in \mathbb{N}} (R \cup \Delta)^n$ which (as we have—more-or-less—just proved) is $\Delta \cup (\bigcup_{i \in \mathbb{N}} R^i)$ which is the reflexive closure of the transitive closure of R.

s is *increasing* so $R \subseteq s(R)$. t is *monotone*, so $t(R) \subseteq t(s(R))$. But the transitive closure of a symmetrical relation is symmetrical so $t(R) \subseteq t(s(R))$ implies $s(t(R)) \subseteq t(s(R))$ as desired.

Finally if $X = \mathbb{N}$ and $R = \Delta \cup \{\langle x, y \rangle : y = px \text{ for some prime } p\}$ then $st(R)$ is the relation that holds between two numbers when they are identical or one is a multiple of the other, and $ts(R)$ is the universal relation $\mathbb{N} \times \mathbb{N}$.

1990:1:9

Peter Dickman's model answer

We are asked to use generating functions to prove that:

$$c_n = \frac{1}{n} \left(\begin{array}{c} 2n-2 \\ n-1 \end{array} \right)$$

where c_n is the number of binary trees with n leaves (NB not n vertices) where no vertex has precisely one descendent. Now the formula given is remarkably similar to the one for Catalan numbers – which were introduced in the section of the course concerned with generating functions. So these may well be useful in answering this question.

Recall that for Catalan numbers:

$$C_n = \frac{1}{n+1} \left(\begin{array}{c} 2n \\ n \end{array} \right)$$

so it is (hopefully) clear that $C_n = c_{n+1}$.

To use generating functions it is necessary to find a recurrence relation...

Consider trees of the form described in the question. Clearly, any such tree which has more than one leaf can be viewed as being composed of two trees joined together by a single (new) root vertex, whose descendents are the two roots of the component smaller trees. Now the sum total of leaves in these sub-trees will be the same as the number of leaves in the composite; and each tree will have at least one leaf. So, the number of trees with some given number of leaves can be determined by considering all of the ways such a tree can be split into left & right subtrees, and the parts combined together.

It follows that:

$$\forall n \geq 1 : c_{n+1} = c_1 c_n + c_2 c_{n-1} + c_3 c_{n-2} + \ldots + c_n c_1$$

Note that we have $n \geq 1$ in the above, because the equation is giving us an expression for $n+1$. The recurrence only holds for the trees with two or more leaves (as we assumed that the root had two descendents).

Also we know that $c_1 = 1$ by inspection.

Note that I've written this out for the case $n+1$ not, as I would normally do, the case n because it makes everything neater later. The result can be achieved from the n case but is a bit messier. The only hint I can give as to how to tell that this is helpful **in advance** is that we already knew that there was an "off by one" effect present in this question.

Now, let us consider $d_n = c_{n+1}, \forall n \geq 0$. Then we have that:

$$\forall n \geq 1 : d_n = d_0 d_{n-1} + d_1 d_{n-2} + \ldots + d_{n-1} d_0$$

Now, if we define $d_k = 0, \forall k < 0$ then we have that

$$\forall n \geq 1 : d_n = \sum_{i=0}^{n-1} d_i d_{n-1-i} = \sum_{i=0}^{\infty} d_i d_{n-1-i}$$

Now, the generating function for the d_n, called $D(z)$ say, has the property that the coefficient of z^n in $D(z)$ is d_n. So we have that:

$$[z^n]D(z) = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{i=0}^{\infty} d_i d_{n-1-i} & \text{otherwise} \end{cases}$$

Whence we derive:

$$\begin{aligned}
D(z) &= \sum_{n=0}^{\infty} d_n z^n \\
&= 1 + \sum_{n=1}^{\infty}\sum_{i=0}^{\infty} d_i d_{n-1-i} z^n \\
&= 1 + \sum_{n=1}^{\infty}\sum_{i=0}^{\infty} d_i d_{n-1-i} z^i z^{n-1-i} z \\
&= 1 + z\sum_{n=1}^{\infty}\sum_{i=0}^{\infty} d_i d_{n-1-i} z^i z^{n-1-i} \\
&= 1 + z\sum_{n=1}^{\infty}\sum_{i=0}^{\infty} d_i z^i \cdot d_{n-1-i} z^{n-1-i} \\
&= 1 + z(D(z))^2
\end{aligned}$$

since the penultimate line is a convolution.

This is a formula we recognise from the Catalan numbers, so we proceed by following the same argument as in the lecture notes...

Reorganising this gives us:

$$z(D(z))^2 - D(z) + 1 = 0$$

Solving this we find that:

$$D(z) = \frac{1 \pm \sqrt{1-4z}}{2z}$$

Since d_n is non-negative $\forall n$ and since $\sqrt{1-4z}$ has only negative signs after the first term we can eliminate the form with an addition in and find:

$$D(z) = \frac{1 - \sqrt{1-4z}}{2z}$$

Which, from a standard binomial identity leads us to:

$$D(z) = \frac{1 - \sqrt{1-4z}}{2z} = \sum_{k\geq 0} \frac{1}{k+1} \left(\begin{array}{c} 2k \\ k \end{array} \right) z^k$$

So we find that:

$$d_n = [z^n]D(z) = \frac{1}{n+1} \left(\begin{array}{c} 2n \\ n \end{array} \right)$$

However, $d_n = c_{n+1}, \forall n \geq 0$ therefore we have that:

$$\forall n \geq 1 : c_n = \frac{1}{n} \left(\begin{array}{c} 2n-2 \\ n-1 \end{array} \right)$$

as required.

Note that the formula is obviously useless for $n = 0$ as it would give $c_0 = \infty$ so we clearly aren't being expected to worry about that case. However it might be worth pointing this out

The second part of the question asks how many trees of the form considered, with n leaves, have depth $n-1$. Again let's look for a recurrence relation. I'll skip

through this fairly quickly... I suggest that you draw some pictures as you read through this. Be aware of the assumption that $n > 2$ in the following.

Consider the trees of this form, that have n leaves and are of depth $n-1$, for arbitrary $n > 2$. Given such a tree, let the number of leaves at depth $n-1$ (ie the maximal depth) be k.

From such a tree we can construct k distinct trees of depth n which have $n+1$ leaves by taking one of the leaves at the $n-1$ level and replacing it with a vertex with two descendents, which are themselves leaves.

Now consider a tree of depth n with $n+1$ leaves, satisfying the condition on numbers of descendents. Selecting any leaf at the maximal depth, its parent is at depth $n-1$ and, by the condition on numbers of descendents, this has another child at depth n. Replacing these two leaves and their parent vertex with a single leaf at depth $n-1$ we either construct a tree with n leaves of depth $n-1$ (if we have removed the only pair of leaves at the maximal depth) or we have a tree of n leaves of depth n.

However the depth of one of our trees must be strictly less than the number of leaves. Assume otherwise, ie that for some such tree, the number of leaves is less than or equal to the depth. Since each 'plucking' operation of the form described above reduces the number of leaves by one and the number of levels by at most one, we would be able to construct a tree with 2 leaves and depth of at least 2 – which is clearly impossible.

So, we have shown that each such tree has precisely 2 leaves at its terminal level, and that the only possible constructions are the k variants of each of the trees of one smaller size. But k is the number of leaves at the terminal level *i.e.* 2, so we have a doubling of the number of possible trees at each level. Given that there are $1 = 2^0$ trees with 2 leaves of depth 1, $2 = 2^1$ trees with 3 leaves of depth 2 and so forth we have that,

$$\forall n \geq 2, \exists\, 2^{n-2} \text{ trivalent trees with } n \text{ leaves and depth } n-1$$

1990:1:11

Equivalence relations correspond to partitions. A PER $\langle X, R\rangle$ that fails to be an equivalence relation features elements $x \in X$ such that $\langle x, x\rangle \notin R$. Such elements are not related to anything at all, since if x is related to y, then by symmetry y is related to x and by transitivity x is related to x.

So we put on one side all the $x \in X$ such that $\langle x, x\rangle \notin R$, leaving behind a subset $X' \subseteq X$ consisting of all those elements related to themselves by R. What is the restriction of R to this set? Clearly it is reflexive. Actually it is transitive and symmetrical as well, because $X' \times X'$ is transitive and symmetrical, and transitivity and symmetry are intersection-closed properties so $R \cap (X' \times X')$ will be transitive and symmetrical. So $R \cap (X' \times X')$ is an equivalence relation on X'.

How many PER's on a set with 4 elements?

There is one way of throwing away no elements, leaving 4. These can be either

all in one piece	1
one singleton, one triple	4
two pairs	3
two singletons	6
all singletons	1

There are 4 ways of throwing away one element, leaving 3. These can be either

all in one piece	$1 \times 4 = 4$
one singleton, one pair	$3 \times 4 = 12$
three singletons,	$1 \times 4 = 4$

There are 6 ways of throwing away two elements, leaving 2. These can be either

all in one piece	$1 \times 6 = 6$
two singletons	$1 \times 6 = 6$

There are 4 ways of throwing away 3 elements leaving 1.

This can be partitioned in only one way	$1 \times 4 = 4$
	$\overline{52}$

Let us prove that T is a PER. We first show that it is transitive. Suppose
(i) $\langle f, g\rangle \in T$ and
(ii) $\langle g, h\rangle \in T$. We want $\langle f, h\rangle \in T$.
By definition of T we infer
(iii) $(\forall x_1, x_2 \in X)(\langle x_1, x_2\rangle \in R \to \langle f`x_1, g`x_2\rangle \in S)$ and
(iv) $(\forall x_2, x_3 \in X)(\langle x_2, x_3\rangle \in R \to \langle g`x_2, h`x_3\rangle \in S)$. (We have relettered variable to make life easier)

Now let x_1 and x_3 be two elements of X such that $\langle x_1, x_3\rangle \in R$. We want to infer $\langle f(x_1), g(x_3)\rangle \in S$. R is symmetrical so $\langle x_3, x_1\rangle \in R$ too. So by transitivity we have $\langle x_1, x_1\rangle \in R$. By (iii) we can infer $\langle f(x_1), g(x_1)\rangle \in S$. We now use (iv) on our assumption that x_1 and x_3 are two elements of X such that $\langle x_1, x_3\rangle \in R$ to infer that $\langle g(x_1), h(x_3)\rangle \in S$. Finally, by transitivity of S we infer that $\langle f(x_1), h(x_3)\rangle \in S$ as desired.

It is much easier to show that T is symmetric. Suppose $\langle f, g\rangle \in T$ and let x_1 and x_2 be two elements of X such that $\langle x_1, x_2\rangle \in R$. We want to infer $\langle f(x_1), g(x_2)\rangle \in S$. R is symmetric, so we infer $\langle x_2, x_1\rangle \in R$, whence $\langle f(x_1), g(x_2)\rangle \in S$ as desired.

To show that T is not in general reflexive, even if R and S both are, take R to be the universal relation on X and S to be the identity relation on Y, where both X and Y have at least two members.

1993:11:11

$\langle A, \leq \rangle$ is a partially ordered set if

1. $(\forall x, y, z \in A)(x \leq y \rightarrow (y \leq z \rightarrow x \leq z)$ ($\leq$ is transitive)
2. $(\forall x, y \in A)(x \leq y \rightarrow (y \leq x \rightarrow x = y))$ ($\leq$ is antisymmetrical)
3. $(\forall x \in A)(x \leq x)$ ($\leq$ is reflexive)

In what follows we write '$x < y$' for '$x \leq y \wedge y \neq x$'

> (a) If $\langle A, \leq \rangle$ is to form a totally ordered set then in addition $\leq$ must satisfy *connexity*.
>
> $$(\forall x, y \in A)(x \leq y \vee y \leq x)$$
>
> or equivalently $<$ must satisfy *trichotomy*
>
> $$(\forall x, y \in A)(x < y \vee x = y \vee y < x)$$
>
> (b) If $\langle A, \leq \rangle$ is to be wellfounded then in addition $<$ (which is the strict version of $\leq$, namely $\{\langle x, y \rangle : x \leq y \wedge x \neq y\}$) must satisfy *wellfoundedness*:
>
> $$(\forall A' \subseteq A)(\exists x \in A')(\forall y \in A')(y \not< x)$$

(This *détour* via strict partial orders is necessary beco's no wellfounded relation can be reflexive.)

> (c) If $\langle A, \leq \rangle$ is to be a complete partially ordered set then one of the following conditions on $\leq$ must be satisfied, depending on what your definition of complete poset is:
>
> One definition is that every *subset* of A must have a least upper bound in the sense of $\leq$. This is
>
> $$(\forall A' \subseteq A)(\exists x \in A)[(\forall y \in A')(y \leq x) \wedge (\forall z \in A)((\forall y \in A')(y \leq z) \rightarrow x \leq z)]$$
>
> ... or that every directed subset of A has a least upper bound. A' is a directed subset of A if $(\forall x, y \in A')(\exists z \in A')(x \leq z \wedge y \leq z)$. (They probably don't mean that tho'.)

To show that the restriction of a partial order of A to some subset B of A is a partial order of B we have to check that $R \cap (B \times B)$ is reflexive transitive and antisymmetrical. Now $B \times B$ is reflexive and transitive, as is R; reflexivity and transitivity are intersection-closed properties, so $R \cap (B \times B)$ is reflexive and transitive. To verify antisymmetry we have to check that if $\langle x, y \rangle$ and $\langle y, x \rangle$ are both in $R \cap (B \times B)$ then $x = y$. But if $\langle x, y \rangle$ and $\langle y, x \rangle$ are both in $R \cap (B \times B)$ then they are both in R, and we know R is antisymmetrical, whence $x = y$ as desired.

(A deeper proof can be obtained by noting only that all the clauses in the definition of partial order are universal. Any universal sentence true in A is true in any subset of A. After all, a universal sentence is true as long as there is no counterexample to it. If A contains no counterexamples, neither can any subset of A. This shows that a substructure of a total order is a total order which is useful later on in the question ...)

Z (i) $\leq$ is a partial order of $\mathbf{Z}$. Indeed it is a total order. (ii) It isn't wellfounded (e.g.: no bottom element) nor (iii) is it a complete poset (e.g.: no top element).

Divisibility (i) is not a partial order because for any integer n, n and $-n$ divide each other but are distinct, so it isn't antisymmetrical. (ii) The relation "n divides m but not *vice versa*" is wellfounded on $\mathbf{Z}$ however. If $X \subseteq \mathbf{Z}$, then its minimal elements under "n divides m but not *vice versa*" are precisely the minimal elements of $\{|n| : n \in X\}$ under "n divides m but not *vice versa*", and this relation, being a subset of a wellfounded relation (and $\leq$ *is* wellfounded on $\mathbb{N}$) is itself wellfounded. (iii) $\mathbf{Z}$ is not a complete poset under divisibility for the same reason as before.

$\mathbb{N}$ $\leq$ is a partial order of $\mathbb{N}$. Indeed it is a total order. It is also wellfounded but it is not a complete poset (as before)

Divisibility is a partial order on $\mathbb{N}$ but not a total order, it is wellfounded. This time we do get a complete poset, because everything divides 0.

$\mathbb{N}^+$ As for $\mathbb{N}$ except that it is not a complete poset (e.g.: no top element)

1994:10:11

The way to do part 2 is to stop trying to be clever and do it the easy way. Let A_n, B_n, C_n be the number of valid strings in $\{A, B, C\}^n$ ending in A, B and C respectively. Clearly

$$C_{n+1} = A_n + B_n + C_n$$

and

$$A_{n+1} = B_{n+1} = B_n + C_n$$

This is because if the last character of a legal string is an A or a B then the penultimate character cannot be an A. We are not going to try to do anything clever like *derive* the equality we have been given, but we can at least confirm it! So let's try to simplify

$$2(A_{n+1} + B_{n+1} + C_{n+1}) + A_n + B_n + C_n$$

and hope that it simplifies to $A_{n+2} + B_{n+2} + C_{n+2}$.

Take out $B_{n+1} + C_{n+1}$ twice to give $A_{n+2} + B_{n+2}$, leaving $2A_{n+1} + A_n + B_n + C_n$. The last three terms add up to C_{n+1}, and $2A_{n+1} = A_{n+1} + B_{n+1}$ so this is $A_{n+1} + B_{n+1} + C_{n+1}$ which is C_{n+2}. Together with the $A_{n+2} + B_{n+2}$ this adds up to $v(n + 2)$ as desired.

Part 3 is 'A'-level maths that you remember from your crèche.

1995:5:4X (maths 1a)

Well, adapted from it!

```
fun f n = if n = 0 then 0 else g(f(n-1) + 1, 1) -1
and g(n,m) = f(f(n-1)) + m + 1;
```

What are the ML types of these two functions?

What are the running times of f and g?

By inspection we notice that $(\forall n \in \mathbb{N})(f(n) = n)$, but we had better prove it! It's true for $n = 0$. For the induction step the recursive declaration tells us that

$$f(n+1) = g(f(n)+1, 1) - 1 \text{ (by substituting } n+1 \text{ for } n)$$

But $f(n) = n$ by induction hypothesis so this becomes

$$f(n+1) = g(n+1, 1) - 1$$

Now, substituting $(n+1)$ for n and 1 for m in the declaration for `g` we get

$$g(n+1, 1) = (n+1-1) + 1 + 1$$

which is $n+2$ giving $f(n+1) = n+1$ as desired.

(d)

The mutual recursion gives us a pair of mutual recurrence relations:

A: $F(n) = G(f(n-1)+1, 1) + F(n-1)$
B: $G(n, m) = F(n-1) + F(f(n-1)) + k$

where F is the cost function for f and G is the cost function for g.

Using $f(n) = n$ we can simplify our recurrence relations as follows.

A′: $F(n) = G(n, 1) + F(n-1)$
B′: $G(n, m) = F(n-1) + F(n-1) + k$ whence
B″: $G(n, m) = 2 \cdot F(n-1) + k$

This gives

$$F(n) = F(n-1) + F(n-1) + F(n-1) + k$$

so $F(n)$ grows like 3^n.

G is exponential too. We have assumed that the cost of adding the second agument ('m') is constant, but altho' this simplification will cause no problems it is a simplification nevertheless. Adding two arguments takes time proportional to the logarithm of the larger of the two. Fortunately the cost functions of these algorithms are so huge that an extra log or two will make no difference to the order.

1996:1:7

A partial ordering is a relation that is reflexive, antisymmetrical and transitive.

'Topological sort' is Compsci jargon for refining a partial ordering, which just means adding ordered pairs to a partial ordering to get a total ordering. The two partial orders of $\mathbb{N} \times \mathbb{N}$ that you have seen are the **pointwise product** ($\langle x, y\rangle \leq_p \langle x', y'\rangle$ iff $x \leq x' \wedge y \leq y'$) and the **lexicographic product** ($\langle x, y\rangle \leq_{lex} \langle x', y'\rangle$ iff $x < x' \vee (x = x' \wedge y \leq y')$). The second is clearly a refinement of the first. It is also clear that the lexicographic product $\mathbb{N} \times \mathbb{N}$ is not isomorphic to $\mathbb{N}$ in the usual ordering, since it consists of ω copies of $\mathbb{N}$. (ω is the length of $\mathbb{N}$ in its usual ordering: the length of $\mathbb{N} \times \mathbb{N}$ in the product ordering is therefore said to be ω^2).

To get a refinement of the product ordering of $\mathbb{N} \times \mathbb{N}$ that is isomorphic to the usual ordering on $\mathbb{N}$ we notice that for a wellordering to be isomorphic to the usual ordering on $\mathbb{N}$ it is sufficient for each point to have only finitely many things below it (given that is also a wellordering, that is). Try $\langle x, y\rangle \leq \langle x', y'\rangle$ iff $(x + y) < (x' + y') \vee (x + y = x' + y' \wedge x \leq x')$. It's a total order, each element has only finitely many things below it (so it's isomorphic to the usual order on $\mathbb{N}$) and it refines the pointwise product ordering.

1996:1:8

The recurrence

$$R: \ w(n, k) = w(n - 2^k, k) + w(n, k - 1)$$

can be justified as follows. Every representation of n pfatz as a pile of coins of size no more than 2^k pfatz either contains a 2^k pfatz piece or it doesn't. Clearly there are $w(n, k - 1)$ representations of n pfatz as a pile of coins of size no more than 2^{k-1} pfatz so that's where the $w(n, k - 1)$ comes from. The other figure arises from the fact that a representation of n pfatz as a pile of coins of size no more than 2^k pfatz and containing a 2^k pfatz piece arises from a representation of $n - 2^k$ pfatz as a pile of coins of size no more than 2^k.

Base case. $w(n, 0) = 1$. That should be enough.

To derive $w(4n, 2) = (n + 1)^2$, substitute $4n$ for n, and 2 for k in R, getting

$$w(4n, 2) = w(4n - 2^2, 2) + w(4n, 1)$$

But this rearranges to

$$w(4n, 2) = w(4(n - 1), 2) + w(4n, 1)$$

$w(4n, 1)$ is $2n + 1$, since we can have between 0 and $2n$ 2-pfatz pieces in a representation of $4n$. This gives

$$w(4n, 2) = w(4(n - 1), 2) + 2n + 1$$

This is a bit clearer if we write this as $f(n) = f(n - 1) + 2n + 1$. This recurrence relation obviously gives $f(n) = (n + 1)^2$ as desired.

We can always get an estimate of $w(n, k)$ by applying equation R recursing on n, and this works out quite nicely if n is a multiple of 2^k because then we hit 0 exactly, after $n/(2^k)$ steps. Each time we call the recursion we add $w(n, k - 1)$ (or rather $w(n - y, k - 1)$ for various y) and clearly $w(n, k - 1)$ is the biggest of them. So $w(n, k)$ is no more than $n/(2^k) \cdot w(n, k - 1)$.

Finally, using R with 2^{k+1} for n again we get $w(2^{k+1}, k) = w(2^k, k) + w(2^{k+1}, k - 1)$. The hint reminds us that every representation of 2^k pfatz using the first k coins gives rise to a representation of 2^{k+1} pfatz using the first $k + 1$ coins. Simply double the size of every coin. It's also true that every representation of 2^k pfatz using the

first k coins gives rise to a representation of 2^{k+1} pfatz using the first $k+1$ coins by just adding a 2^k pfatz piece. The moral is: $w(2^{k+1}, k+1) = 2 \cdot w(2^k, k)$. This enables us to prove the left-hand inequality by induction on k.

To prove the right-hand inequality we note that any manifestation of 2^k pfatz using smaller coins can be tho'rt of as a list of length k where the ith member of the list tells us how many 2^i pfatz coins we are using. How many lists of length k each of whose entries are at most 2^k are there? Answer $(2^k)^k$, which is 2^{k^2}.

1997:1:2

1997:1:7

(a) Yes: equality is a partial order, and it is tree-like beco's the set of strict predecessors is always empty.

(b) Yes. The usual order is a partial (indeed *total*) order and every total order is tree-like.

(c) No. This is a partial order but is not tree-like beco's (for example) 6 has two immediate strict predessors.

(d) This is reflexive and antisymmetrical (if xRy and yRx so that x and y are either equal or each is the greatest prime factor of the other then they are equal). The hard part is to show that it is transitive. Suppose xRy and yRz. If $x = y$ or $z = z$ we deduce xRz at once so consider the case where xRy and yRz hold *not* in virtue of $x = y$ or $y = z$. But this case cannot arise, because if yRz and $y \neq z$, then y is a prime, and the only x such that xRy is y itself. Finally, it's easy to show this relation is tree-like, because no number can have more than one greatest prime factor.

It seem to me that the number of treelike partial orderings of n elements is precisely $n!$. Each treelike partial ordering of n chaps gives rise to n new partial orderings beco's the extra chap can be stuck on top of any of the n things already there. No new partial ordering gets counted twice.

2002:1:8

The last part seems to have caused problems for some. Let's have a look.

We are contemplating relations that hold between elements of Ω and subsets of Ω. An example of the sort of thing the examiner has in mind is the relation that a point y in the plane bears to a (typically non-convex) region X when y is in the convex hull of X. (A picture would be nice at this point!) The idea is that y one of the points you have to "add" to obtain something convex. (Check that you know what a convex set is, as i'm going to procede on the assumption that you do, and use it as a—one hopes!—illuminating illustration)

What is $\mathcal{R}$? $\mathcal{A}$ is an intersection-closed family of subsets of Ω. (As it might be, the collection of convex subsets of the plane). We are told that it is the relation that relates y to X whenever anything in A that extends X also contains y. In our illustration—where $\mathcal{A}$ is the collection of convex subsets of the plane— $\mathcal{R}$ is the relation that hold between X and y whenever y is in the convex hull of X. Certainly in this case any set that is $\mathcal{R}$-closed is convex.

Assume C is $\mathcal{R}$-closed. That is to say

$$\forall(X, y) \in \mathcal{R}.X \subseteq C \rightarrow y \in C \tag{10.1}$$

But $\mathcal{R} = \{(X, y) \in \mathcal{P}(\Omega) \times \Omega | \forall A \in \mathcal{A} X \subseteq A \rightarrow y \in A\}$. Substituting this for '$\mathcal{R}$' in 10.1 we obtain

$$\forall(X, y) \in \{(X, y) \in \mathcal{P}(\Omega) \times \Omega | \forall A \in \mathcal{A} X \subseteq A \rightarrow y \in A\}.X \subseteq C \rightarrow y \in C \tag{10.2}$$

which reduces to

$$\forall(X, y)[(\forall A \in \mathcal{A})(X \subseteq A \rightarrow y \in A) \wedge X \subseteq C. \rightarrow y \in C] \tag{10.3}$$

The examiners suggest you should consider the set $\{A \in \mathcal{A} : C \subseteq A\}$. I think they want you to look at $\bigcap\{A \in \mathcal{A} : C \subseteq A\}$.

If you've followed the action this far you would probably think of this anyway, since this is a set that you know must be in $\mathcal{A}$ and it seems to stand an outside chance of being equal to C. So let's look again at 10.3 to see if it does, in fact, tell us that $\bigcap\{A \in \mathcal{A} : C \subseteq A\}$ is C.

And—of course—it does. First we instantiate 'X' to 'C' in 10.3 to obtain:

$$\forall y[(\forall A \in \mathcal{A})(C \subseteq A \rightarrow y \in A) \rightarrow y \in C] \tag{10.4}$$

Now let y be an arbitrary member of $\bigcap\{A \in \mathcal{A} : C \subseteq A\}$. That means that y satisfies the antecedent of 10.4. So it satisifes the consequent of 10.4 as well. So we have proved that $\bigcap\{A \in \mathcal{A} : C \subseteq A\}$ is a subset of C. It was always a superset of C, so it is equal to C. So $C \in \mathcal{A}$ as desired.

Question 10.2.1.1

Show that $\bigcup_{n\in\mathbb{N}} R^n$ is the smallest transitive relation extending R.

To do this it will be sufficient to show

1. $\bigcup_{n\in\mathbb{N}} R^n$ is transitive
2. If S is a transitive relation $\supset R$ then $\bigcup_{n\in\mathbb{N}} R^n \subseteq S$

For (1) We need to show that if $\langle x, y\rangle$ and $\langle y, z\rangle$ are both in $\bigcup_{n\in\mathbb{N}} R^n$ then $\langle x, z\rangle \in \bigcup_{n\in\mathbb{N}} R^n$. If $\langle x, y\rangle \in \bigcup_{n\in\mathbb{N}} R^n$ then $\langle x, y\rangle \in R^k$ for some k and if $\langle y, z\rangle \in \bigcup_{n\in\mathbb{N}} R^n$ then $\langle y, z\rangle \in R^j$ for some j. Then $\langle x, z\rangle \in R^{j+k} \subseteq \bigcup_{n\in\mathbb{N}} R^n$.

For (2) Let $S \supset R$ be a transitive relation. So $R \subseteq S$. We prove by induction on $\mathbb{N}$ that for all $n \in \mathbb{N}$, $R^n \subseteq S$. Suppose $R^n \subseteq S$. Then

$$R^{n+1} = R^n \circ R \ \subseteq^{(a)} \ S \circ R \ \subseteq^{(b)} \ S \circ S \ \subseteq^{(c)} \ S.$$

(a) and (b) hold because $\circ$ is *monotone*: if $X \subseteq Y$ then $X \circ Z \subseteq Y \circ Z$.
(c) holds because S is transitive.

Question 10.2.1.4

Let R and S be equivalence relations. We seek the smallest equivalence relation that is a superset of $R \cup S$. We'd better note first that this really is well defined, and it is, because being-an-equivalence-relation is the conjunction of three properties all of them intersection closed, so it is itself intersection-closed.

This least equivalence relation extending $R \cup S$ is at least transitive, so it must be a superset of $t(R \cup S)$, the transitive closure of $R \cup S$. Wouldn't it be nice if it actually *were* $t(R \cup S)$? In fact it is, and to show this it will be sufficient to show that $t(R \cup S)$ is an equivalence relation. Must check: transitivity, reflexivity and symmetry. Naturally $t(R \cup S)$ is transitive by construction. R and S are reflexive so $R \cup S$ is reflexive. In constructing the transitive closure we add new ordered pairs but we never add ordered pairs with components we haven't seen before. This means that we never have to add any ordered pairs $\langle x, x \rangle$ beco's they're all already there. Therefore $t(R \cup S)$ is reflexive as long as R and S are. Finally we need to check that $t(R \cup S)$ is symmetrical. The transitive closure of a symmetrical relation is also symmetrical. First we show by induction on n that R^n is symmetrical as long as R is. Easy when $n = 1$. Suppose R^n is symmetrical: i.e., $R^n = (R^n)^{-1}$. $R^{n+1} = R \circ R^n$ anyway. The inverse of this is $(R^{-1})^n \circ R^{-1}$. $(R^{-1})^n$ is of course R^{-n}, so $(R^{-1})^n \circ R^{-1}$ is $(R^{-n}) \circ R^{-1}$. $R^{-n} = R^n$ by induction hypothesis so $(R^{-1})^n \circ R^{-1}$ is $R^n \circ R$ which is of course R^{n+1}. Then the union of a lot of symmetrical relations is symmetrical, so the transitive closure (which is the union of all the (symmetrical) iterates of R) is likewise symmetrical.

Actually we can give another—perhaps simpler—proof of this. $t(R) = \bigcap\{S : R \subseteq S \wedge S^2 \subseteq S\}$, or $\bigcap X$ for short. Notice that R is symmetrical, then X is closed under taking inverses (the inverse of anything in X is also in X). And clearly the intersection of a class closed under taking inverses is symmetrical.

Question 10.2.1.5

Show that if R and S are transitive relations, so is $R \cap S$.

$(R \cap S) \circ (R \cap S) \subseteq R \circ R \subseteq R$

$(R \cap S) \circ (R \cap S) \subseteq S \circ S \subseteq S$

so

$(R \cap S) \circ (R \cap S) \subseteq R \cap S$

Notice that the same argument shows that the intersection of any number of transitive relations is a transitive relation: i.e., transitivity is an intersection closed property of relations.

Question 10.2.1.6

Let R be a relation on A. ('r', 's' and 't' denote the reflexive, symmetric and transitive closure operations respectively.)

1. Prove that $rs(R) = sr(R)$.
2. Does R transitive imply $s(R)$ transitive?
3. Prove that $rt(R) = tr(R)$ and $st(R) \subseteq ts(R)$.
4. If R is symmetrical must the transitive closure of R be symmetrical? Prove or give a counterexample.
5. Think of a binary relation R, and of its graph, which will be a directed graph $\langle V, E \rangle$. On any directed graph we can define a relation "I can get from vertex x to vertex y by following directed edges" which is certainly transitive, and we can pretend it is reflexive because after all we can get from a vertex to itself by just doing nothing at all. Do this to our graph $\langle V, E \rangle$, and call the resulting relation S. How do we describe S in terms of R?

(a) Prove that $rs(R) = sr(R)$:

$$\begin{aligned} r(s(R)) &= s(R) \cup I \\ &= (R \cup R^{-1}) \cup I \\ &= (R \cup I) \cup (R^{-1} \cup I) \\ &= (R \cup I) \cup (R^{-1} \cup I^{-1}) \\ &= (R \cup I) \cup (R \cup I)^{-1} \\ &= s(r(R)) \end{aligned}$$

(b) The symmetric closure of a transitive relation is not automatically transitive: take R to be set inclusion on a power set.

(c) Prove that $rt(R) = tr(R)$:

$$\begin{aligned} r(t(R)) = t(R) \cup I = R \cup R^2 \cup \ldots R^n \ldots \cup I \\ = (R \cup I) \cup (R^2 \cup I) \cup (R^n \cup I) \ldots \end{aligned}$$

At this point it would be nice to be able to say $(R^n \cup I) = (R \cup I)^n$ but this isn't true. $(R \cup I)^n$ is actually $R \cup R^2 \ldots R^n \cup I$. But this is enuff to rewrite the last line as

$$(R \cup I) \cup (R \cup I)^2 \cup (R \cup I)^3 \ldots$$

which is of course $t(r(R))$ as desired.

The transitive closure of a symmetrical relation is also symmetrical. First we show by induction on n that R^n is symmetrical as long as R is. Easy when $n = 1$. Suppose R^n is symmetrical. $R^{n+1} = R \circ R^n$. The inverse of this is $(R^{-1})^n \circ R^{-1}$ which by induction hypothesis is $R^n \circ R$ which is of course R^{n+1}. Then the union of a lot of symmetrical relations is symmetrical, so the transitive closure (which is the union of all the (symmetrical) iterates of R is likewise symetrical.

Finally S is the reflexive transitive closure of R.

Question 10.2.1.8

> Show that—at least if $(\forall x)(\exists y)(\langle x, y\rangle \in R)$—$R \circ R^{-1}$ is a fuzzy. What about $R \cap R^{-1}$? What about $R \cup R^{-1}$?

If $\langle x, y\rangle \in R$ then $\langle y, x\rangle \in R^{-1}$ so $\langle x, x\rangle \in R \circ R^{-1}$. That takes care of reflexivity. Suppose $\langle x, z\rangle \in R \circ R^{-1}$. Then there is a y such that $\langle x, y\rangle \in R$ and $\langle y, z\rangle \in R^{-1}$. But then $\langle z, y\rangle \in R$. So $\langle x, z\rangle \in R \circ R^{-1}$ is the same as $(\exists y)((\langle x, y\rangle \in R) \wedge (\langle z, y\rangle \in R))$. But this is clearly symmetric in x and z, so we can rearrange it to get $(\exists y)((\langle y, x\rangle \in R^{-1}) \wedge (\langle z, y\rangle \in R))$ which is $\langle z, x\rangle \in R \circ R^{-1}$ as desired.

$R \cup R^{-1}$ is the symmetric closure of R and is of course symmetric, but there is no reason to expect it to be reflexive: it'll be reflexive iff R is reflexive.

Question 10.2.1.9

> Given any relation R there is a least $T \supseteq R$ such that T is transitive, and a least $S \supseteq R$ such that S is symmetrical, namely the transitive and symmetric closures of R. Must there also be a maximal $S \subseteq R$ such that S is transitive? And must there be a maximal $S \subseteq R$ such that S is symmetrical? The answer to one of these last two questions is 'yes': find a cute formulation.

$R \cap R^{-1}$ is the largest symmetrical relation included in R. The unwary sometimes think *this* is the symmetric closure of R. The point is that altho' being-the-complement-of-a-transitive-relation is not an intersection-closed property, nevertheless being-the-complement-of-a-symmetric-relation **is** intersection-closed, since it is the same as being symmetrical. $R \cap R^{-1}$ is the complement of the symmetric closure of the complement of R. Do not confuse complements with converses!!

Question 10.2.1.12

If $x \leq S(y)$ and $y \leq S(x)$ then x and y are neighbouring naturals. This is $R \cap R^{-1}$. x and y are related by the transitive closure of this relation iff there is a finite sequence $x_0, x_1, x_2 \ldots x_n = y$ such that each x_i is adjacent to x_{i+1}. But clearly any two naturals are connected by such a chain, so the transitive closure is the universal relation. For part 2, remember that x is related to y by $R \setminus R^{-1}$ if it is related to y by R but not by R^{-1}. In this case that means $x \leq S(y) \wedge y \not\leq S(x)$. This is $x \leq S(y) \wedge S(x) < y$. The second conjunct implies the first so we can drop the first, getting $S(x) < y$. Getting the transitive closure of this is easy, 'cos it's transitive already!

Question 10.2.1.13

Are the two following conditions on partial orders equivalent?

1. $(\forall xyz)(z > x \not\leq y \not\leq x \rightarrow z < y)$
2. $(\forall xyz)(z > x \not\leq y \not\leq x \rightarrow z > y)$.

Assume (i) $(\forall xyz)(z \leq x \not\leq y \not\leq x \rightarrow z \leq y)$ and aim to deduce (ii) $(\forall xyz)(z \geq x \not\leq y \not\leq x \rightarrow z \geq y)$. To this end assume $z \geq x$, $x \not\leq y$ and $y \not\leq x$ and hope to deduce $z \geq y$.

$x \leq z$ tells us that $z \not\leq y$ for otherwise $x \leq y$ by transitivity, contradicting hypothesis. Next, assume the negation of what we are trying to prove. This gives us $y \not\leq z$. But then we have $y \not\leq z \not\leq y$ and $x \leq z$ so by (i) we can infer $x \leq y$, contradicting assumption.

I think the proof in the other direction is similar but i haven't checked it.

For the record: to any partial order there corresponds in a obvious way a strict partial order. (like $\leq$ and $<$ on $\mathbb{N}$, for example.) Consider the strict partial order corresponding to a partial order satisfying this condition we have just been discussing. If it is wellfounded it is said to be a **prewellordering**. This is because we can think of it as a total ordering of the equivalence classes (under the relation $x \simeq y$ iff $x = y \vee x \not\leq y \not\leq x$), and if $<$ is wellfounded this in in fact a wellordering of the equivalence classes.

Question 10.2.1.14

Show that $R \subseteq S$ implies $R^{-1} \subseteq S^{-1}$

The way to do this is to assume that $R \subseteq S$ and let $\langle x, y \rangle$ be an arbitrary ordered pair in R^{-1}. We then want to infer that $\langle x, y \rangle$ is in S^{-1}.

If $\langle x, y \rangle$ is in R^{-1} then $\langle y, x \rangle$ is in R, because R^{-1} is precisely the set of ordered pairs $\langle x, y \rangle$ such that $\langle y, x \rangle$ is in R. (We would write this formally as: $R^{-1} = \{\langle x, y \rangle : \langle y, x \rangle \in R\}$.) But $R \subseteq S$, so $\langle y, x \rangle$ is in S, and so (flip things round again) $\langle x, y \rangle$ is in S^{-1}.

Notice that to tell this story successfully we have to come out of the closet and think of R and S as sets of ordered pairs, that is, as relations-in-extension.

Question 10.2.1.15

Show that $R \circ (S \circ T) = (R \circ S) \circ T$

That is to say $xR \circ (S \circ T)y$ iff $x(R \circ S) \circ Ty$

Now, by definition of relational composition,

$$xR \circ (S \circ T)y$$

is

$$(\exists z)(xRz \wedge z(S \circ T)y)$$

and expand the second '$\circ$' to get

$$(\exists z)(xRz \wedge (\exists w)(zSw \wedge wTy))$$

We can pull the quantifiers to the front because[13] '$(\exists u)(A \wedge \phi(u))$' is the same as '$A \wedge (\exists u)\phi(u)$' getting

$$(\exists z)((\exists w)(xRz \wedge (zSw \wedge wTy)))$$

and

$$(\exists z)(\exists w)(xRz \wedge (zSw \wedge wTy))$$

and we can certainly permute the quantifiers getting

$$(\exists w)(\exists z)(xRz \wedge (zSw \wedge wTy))$$

we can permute the brackets in the matrix of the formula because '$\wedge$' is associative getting

$$(\exists w)(\exists z)((xRz \wedge zSw) \wedge wTy)$$

import the existential quantifier again getting

$$(\exists w)((\exists z)(xRz \wedge zSw) \wedge wTy)$$

and reverse the first few steps by using the definition of $\circ$ to get

$$(\exists w)(x(R \circ S)w \wedge wTy)$$

and

$$x(R \circ S) \circ Ty$$

as desired.

Question 10.2.1.16b

The lexicographic order on $\mathbb{N}^2$ is wellfounded, so we can do wellfounded induction on it. This means that if we can prove that, if every ordered pair below p has some property ϕ then the pair p has property ϕ as well, then every ordered pair in $\mathbb{N}^2$ has that property.

Now let $\phi(\langle x, y\rangle)$ say that if the bag is started with x black balls and y white balls in it the process will eventually halt with only one ball in the bag. Suppose $\phi(\langle x', y'\rangle)$ holds for every $\langle x', y'\rangle$ below $\langle x, y\rangle$ in the lexicographic product $\mathbb{N}^2$. We want to be sure that if the bag is started with x black balls and y white balls in it the process will eventually halt with only one ball in the bag. The first thing that happens is that we pick two balls out of the bag and the result is that at the next stage we have either $x - 2$ black balls and an unknown number of white balls, or we have x black balls and $y - 1$ white balls. But both these situations are described by ordered pairs below $\langle x, y\rangle$ in the lexicographic product $\mathbb{N}^2$, so by induction hypothesis we infer that if the bag is started with x black balls and y white balls in it the process will eventually halt with only one ball in the bag, as desired.

[13] At least as long as 'u' is not free in A.

Question 10.2.1.23

Everybody loves my baby, so in particular my baby loves my baby. **My baby loves nobody but me.** That is to say, if x is loved by my baby, then $x =$ me. So my baby = me.

Question 10.2.2.2a

The answer is the relation that holds between k and $k+1$ for $0 \leq k < n$ and between n and 0.

Question 10.2.2.12

Paula Buttery's answer to one of the fiddly ones.

```
- fun f g b a = g a b;
val f = fn : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
- fun ff g = let fun fa a = let val (b,c) = g a in b end;
= fun fe a = let val (b,c) = g a in c end;
= in (fa, fe) end;
val ff = fn : ('a -> 'b * 'c) -> ('a -> 'b) * ('a -> 'c)
-
```

Question 10.2.2.11

Show that the largest and smallest elements of a totally ordered set with n elements can be found with $\lceil 3n/2 \rceil - 1$ comparisons if n is odd, and $3n/2 - 2$ comparisons if n is even.

First check this for a few small values. If $n = 2$ we need 1, for $n = 3$ we need 3, for $n = 4$ we need 4.

The induction step requires us to show that adding two more elements to a set requires us to perform no more than three extra comparisons.

So suppose we have a set X with n members, and we have found the top and bottom elements in $3n/2 - 1$ comparisons. Call them t and b. Let the two new elements be x and y. With one comparison we can find out which is bigger. Without loss of generality suppose it is x. Compare x with t to find the biggest element of $X \cup \{x, y\}$, and compare y with b to find the smallest. This has used three extra comparisons.

Question 10.2.2.17

The exponent on the LHS is $x^{x^{x^{x^{\cdots}}}}$ which is 2, so $x^2 = 2$ and $x = \sqrt{2}$. That was easy. The problem with this is that the second equation gives $x^4 = 4$ and thence $x = \sqrt{2}$ again. They can't both be right!

Of course the answer is that the reasoning that led us to conclude that $x = \sqrt{2}$ in the first place doesn't prove that that is the answer. All we have done is show that **if** there is a solution it must be $\sqrt{2}$. We haven't shown that there **is** a solution. In fact it is a simple matter to show by induction that the approximants to the LHS, which we generate as follows

$$a_0 =: \sqrt{2}; \quad a_{n+1} =: \sqrt{2}^{a_n}$$

... are all less that 2. So the sequence has a limit which is ≤ 2.

Let's see what we can do that is more general.

Let $F(x) =_{df} x^{x^{x^{x^{\cdots}}}}$

We have $x^{F(x)} = F(x)$. The inverse to this function is the function $\lambda x.x^{1/x}$. This is much easier to understand. For example we can differentiate it. It is the same as $e^{(log\ x)/x}$ whose differential is of course $e^{(log\ x)/x} \cdot (1/x^2 - (log\ x)/x^2)$. This is zero when $x = e$, and this is clearly a maximum. The fact that the differential is zero there of course means that F reaches a maximum at $e^{1/e}$ and that $F'(e^{1/e})$ is infinite. This gives us the amusing but (as far as i know) useless fact that

$$(e^{1/e})^{(e^{1/e})^{(e^{1/e})^{(e^{1/e})^{(e^{1/e})^{\cdots}}}}} = e$$

(Check this: if the LHS is to evaluate to x we must have $(e^{(1/e)})^x = x$ and e is certainly a solution to this equation.)

We can get a power series expansion of F for values of x not much bigger than 1. Let Σ be the power series for $F(1 + x)$. Then we have

$$(1 + x)^{\Sigma} = \Sigma$$

and we can use the binomial theorem to expand the left hand side. This gives us a sequence of equations expressing later coefficients of Σ in terms of earlier coefficients in a wellfounded way. I haven't worked out the general formula for a_n the coefficient of x^n in $F(1 + x)$ tho' in principle it could be done. ($a_0 = 1$ for a start!)

Question 10.2.2.22

Let $m = |F|$ and $p = |\bigcup F|$. Let $C = \{\langle x, A\rangle : x \in A \in F\}$.

Given $x \in \bigcup F$, pick $B \in F$ with $x \in B$. Let $Y_x = \{A \in F : x \in A\}$ and $N_x = \{A \in F : x \notin A\}$. The map $\lambda A.(A \Delta B)$ permutes F and swaps Y_x and N_x. Hence $|Y_x| = |N_x| = m/2$.

So $|C| = (1/2)mp$, as each x is in exactly $m/2$ A's. But each A contains $\leq k$ things, and one A contains none at all, so $|C| \leq (m-1)k$ whence $p \leq \frac{m-1}{m} \cdot 2k < k$.

Question 10.2.2

The problem is to find a cute way of getting lots of variables, literals, call them what you will. One solution is to cast `int`s as literals. That way one can represent a state as a list of `bool`s, and one has a nice recursion over `int`s for the base (literal) case of the declaration of the `eval` function.

Various strategies are adopted in the following code submitted by my supervisees over several years. I don't know who did the second answer.

```
(*  This code was cooperatively produced by Mike Bond and Joseph Lord.  It
produces truth results for all possible inputs into a boolean expression.
To run use the function tttable (x) where x is a boolean expression using
the characters a-z excluding o as the variables and symbols ! (NOT), &
(AND) and + (OR).  Precedence is set in that order.  & and + are infixes
and ! precedes the complemented variable.  *)

infix 7 !;
infix 6 &;
infix 5 +;

nonfix !;


datatype BOOL  = VAR of string
       | NOT of BOOL
       | AND of BOOL*BOOL
       | OR of BOOL*BOOL;

fun (a & b) = ( AND( a , b ) );
fun (a + b) = ( OR( a , b ) );
fun ! x = NOT ( x );

fun lvars( VAR(x) ) = [x] |
    lvars( AND(x,y) ) = lvars(x) @ lvars(y) |
    lvars( OR(x,y) ) = lvars(x) @ lvars(y) |
    lvars( NOT(x) ) = lvars(x);

fun ispresent(x,[]) = false |
    ispresent(x,t::ts) = if x=t then true
     else ispresent(x,ts);

fun member(bh,[]) = false |
    member(bh,mainh::maint) = if bh=mainh then true
      else member(bh,maint);

fun fr(main,[]) = main |
    fr(main,bh::bt) = if member(bh,main) then fr(main,bt)
else fr(bh::main,bt);

fun evl(expression,alist) =
    let fun evaluate( AND(x,y) ) = evaluate(x) andalso evaluate(y) |
    evaluate( OR(x,y) ) = evaluate(x) orelse evaluate(y) |
    evaluate( NOT(x) ) = not (evaluate(x)) |
    evaluate( VAR(x) ) = ispresent(x,alist)
     in
```

```
evaluate(expression)
     end;

fun tline(expression,alist) =
     if evl(expression,alist) then (alist,"TRUE")
      else (alist,"FALSE");


fun powset ([],base) = [base]
  | powset (x::xs,base) = powset (xs,base) @ powset(xs,x::base);


fun ttable(expression,[]) = []
  | ttable(expression,p::ps)=[tline(expression,p)]::ttable(expression,ps);

fun tttable(expression) =
ttable(expression,powset( fr([],lvars(expression)) , [] ));


val a=VAR("a");
val b=VAR("b");
val c=VAR("c");
val d=VAR("d");
val e=VAR("e");
val f=VAR("f");
val g=VAR("g");
val h=VAR("h");
val i=VAR("i");
val j=VAR("j");
val k=VAR("k");
val l=VAR("l");
val m=VAR("m");
val n=VAR("n");
val p=VAR("p");
val q=VAR("q");
val r=VAR("r");
val s=VAR("s");
val t=VAR("t");
val u=VAR("u");
val v=VAR("v");
val w=VAR("w");
val x=VAR("x");
val y=VAR("y");
val z=VAR("z");



(* We'll need things for dealing with lazy-lists so here's some.        *)
datatype lazylist = Tip
                  | Cell of (int->bool) * (unit->lazylist);

fun head (Cell (x,y)) = x;
fun tail (Cell (x,y)) = y ();
```

```
(* Basic list functions.                                                  *)
(*                                                                        *)
(* Membership test                                                        *)
fun mem [] x = false
  | mem (y::ys) x = if x=y then true else mem ys x;


(* Merge two lists set-union style.  Only x add to ys if it isn't       *)
(* already in ys.                                                         *)
fun merge ([], ys) = ys
  | merge (x::xs, ys) = if mem ys x then merge (xs, ys)
                                    else x::merge (xs, ys);


(* Simply prefixes x to each list y in the list of lists ys.            *)
fun prefix (x, []) = []
  | prefix (x, y::ys) = (x::y) :: prefix (x, ys);


(* Return a list of all the possible subsets of the list given as an    *)
(* argument.                                                              *)
fun subsets [] = [[]]
  | subsets (x::xs) = prefix (x, subsets xs) @ subsets xs;


(* Here's the data-type we're going to use for our formulae.            *)
datatype fmla   = Lit of int
                | Not of fmla
                | And of fmla * fmla
                | Or of fmla * fmla
                | Implies of fmla * fmla
                | Iff of fmla * fmla;


(* The SAT function.                                                      *)
(*                                                                        *)
(* Evaluates the function given that the literals take the states       *)
(* returned by the function given as the second parameter.  States      *)
(* should be a function with type int->bool.                              *)
fun SAT (Lit (l), states) = states l
  | SAT (Not (f), states) = not (SAT (f, states))
  | SAT (And (f, g), states) = SAT (f, states) andalso SAT (g, states)
  | SAT (Or (f, g), states) = SAT (f, states) orelse SAT (g, states)
  | SAT (Implies (f, g), states) = (not (SAT (f, states))) orelse
                                       SAT (g, states)
  | SAT (Iff (f, g), states) = SAT (Implies (f, g), states) andalso
                               SAT (Implies (g, f), states);
  (*SAT (Iff (f, g), states) = (SAT (f,states) = SAT(g,states))   *)

(* All the below is for the valid function.                               *)

(* First we want a list of the literals in the function.  We will need  *)
(* this in order to work out all the possible true/false combinations.  *)
fun getlits (Lit l) = [l]
  | getlits (Not l) = getlits l
  | getlits (And (l, m)) = merge (getlits l, getlits m)
  | getlits (Or (l, m)) = merge (getlits l, getlits m)
```

```
  | getlits (Implies (l, m)) = merge (getlits l, getlits m)
  | getlits (Iff (l, m)) = merge (getlits l, getlits m);


(* Return a lazy list the head of which is a function which returns     *)
(* whether its argument is a member of that list (in the lazy list of   *)
(* lists which it is.                                                   *)
fun allstates [] = Tip
  | allstates (x::xs) = Cell (mem x, fn () => allstates xs);


(* Now the crunchy bit.  Valid takes a single argument - the function.  *)
(* It extracts all the literals from it and then finds all the subsets  *)
(* of the list.  We will use this to find all the true/false            *)
(* combinations: Each subset will count as a separate case.  If the     *)
(* literal in question is a member of the subset then it is true        *)
(* in this case.  If it is not a member then it is false.   We test     *)
(* phi with each case returning false as soon as we get a false, but    *)
(* only returning true when we've tested all the cases and all were     *)
(* true.                                                                *)
fun valid phi =
    let fun validbit (Tip) = true
          | validbit (Cell (f, g)) =
            if SAT (phi, f) then validbit (g ())
                            else false
    in  validbit (allstates (subsets (getlits phi)))
    end;


(* Test formulae.                                                       *)
(*      P /\ Q -> Q /\ P                                                *)
val ok = Implies (And (Lit 1, Lit 2), And (Lit 2, Lit 1));

(*      P \/ Q -> P /\ Q                                                *)
val bad = Implies (Or (Lit 1, Lit 2), And (Lit 1, Lit 2));

(*      ((P -> Q) -> P) -> P                                            *)
val peirceslaw = Implies (Implies (Implies (Lit 1, Lit 2), Lit 1), Lit 1);

----------------------------------->8---------------------------------------
```

Question 10.2.2 part b

```
Below is a version of SAT which has been extended to cope with the
unary box operator, and a revised data-type which includes the
operator.

----------------------------------->8---------------------------------------
(* Here's the data-type we're going to use for our formulae.           *)
datatype fmla   = Lit of int
                | Not of fmla
                | Box of fmla
                | And of fmla * fmla
                | Or of fmla * fmla
                | Implies of fmla * fmla
```

```
               | Iff of fmla * fmla;
(* The SAT function.                                                    *)
(*                                                                      *)
(* Evaluates the function given that the literals take the states       *)
(* returned by the function given as the second parameter.  States      *)
(* should be a function with type int->bool.                            *)
(* In this version the unary box operator has been implemented.         *)
fun SAT (Lit (l), states) = states l
  | SAT (Not (f), states) = not (SAT (f, states))
  | SAT (And (f, g), states) = SAT (f, states) andalso SAT (g, states)
  | SAT (Or (f, g), states) = SAT (f, states) orelse SAT (g, states)
  | SAT (Implies (f, g), states) = (not (SAT (f, states))) orelse
                                        SAT (g, states)
  | SAT (Iff (f, g), states) = SAT (Implies (f, g), states) andalso
                               SAT (Implies (g, f), states)
  | SAT (Box (f), states) =
    let fun validbit (Tip) = true
          | validbit (Cell (g, h)) =
            if SAT (f, g) then validbit (h ())
                          else false
    in  validbit (allstates (subsets (getlits f)))
    end;
----------------------------------->8---------------------------------------
```

Question 10.2.2 part c

```
Below is a version of SAT which has been extended to cope with the R
relation on the set of valid states.

----------------------------------->8---------------------------------------
(* The SAT function. (part c)                                           *)
(*                                                                      *)
(* Evaluates the function given that the literals take the states       *)
(* returned by the function given as the second parameter.  States      *)
(* should be a function with type int->bool.                            *)
(* In this version the unary box operator has been implemented in the   *)
(* style of part c.  The idea is that the third parameter R is the      *)
(* relation between states.  Given a state R it will return a lazy list *)
(* of states (in the manner of allstates).                              *)
fun SAT (Lit (l), states, R) = states l
  | SAT (Not (f), states, R) = not (SAT (f, states, R))
  | SAT (And (f, g), states, R) = SAT (f, states, R) andalso SAT (g, states, R)
  | SAT (Or (f, g), states, R) = SAT (f, states, R) orelse SAT (g, states, R)
  | SAT (Implies (f, g), states, R) = (not (SAT (f, states, R))) orelse
                                        SAT (g, states, R)
  | SAT (Iff (f, g), states, R) = SAT (Implies (f, g), states, R) andalso
                               SAT (Implies (g, f), states, R)
  | SAT (Box (f), states, R) =
    let fun validbit (Tip) = true
          | validbit (Cell (g, h)) =
            if SAT (f, g, R) then validbit (h ())
                          else false
    in  validbit (R states)
    end;
```

And an answer from David Burleigh

```
(* A formula datatype and its implementation.  By David Burleigh. *)
datatype formula = lit of string
   | fand of formula * formula
   | for of formula * formula
   | fimp of formula * formula
   | fnot of formula
   | feq of  formula * formula;

fun eval (fand(x,y) :formula) statefun =
 statefun x andalso statefun y
  | eval (for(x,y) :formula) statefun =
 statefun x orelse statefun y
  | eval (fimp(x,y) :formula) statefun =
 not(statefun x) orelse statefun y
  | eval (feq(x,y) :formula) statefun =
 (statefun x) = (statefun y)
  | eval (fnot(x) :formula) statefun =
 not(statefun x)
  | eval (a :formula) statefun = statefun a;

fun state (s::ss :(string * bool) list) x =
 let val states = s::ss
 in
  let fun staten [] x = eval x (staten states)
          | staten ((a,b)::ss) x =
    if x = lit a then b
    else staten (ss) x
  in
   staten states x
  end
 end;

val statelist = [[("p",false),("q",false)],[("p",true),("q",false)],
   [("p",false),("q",true)],[("p",true),("q",true)]];

val statefuns = map state statelist;

fun truthtable fmula = map (eval fmula) statefuns;

val nd = fand(lit "p",lit "q");
val or = for(lit "p", lit "q");=09

(* A formula datatype and its implementation.  By
David Burleigh. *)
datatype formula = lit of = string | fand of formula * formula | for of formula * formula| fimp of form
formula | feq of formula * formula
fun eval (fand(x,y) :formula) statefun = statefun
x andalso statefun y | eval (for(x,y) :formula) statefun = statefun x orelse statefun y | eval (fimp(x,
not(statefun x) orelse statefun y | eval (feq(x,y) = :formula)
statefun = (statefun x) = (statefun y) | eval =
(fnot(x) :formula)
statefun = not(statefun x) | eval (a :formula) statefun =
= statefun a

fun state (s::ss :(string * bool) list) x =
   let val states = s::ss;
```

```
let fun staten [] x = eval x = (staten states) | =
staten((a,b)::ss) x if x = lit a then = else staten (ss) x; staten states =
x;

val statelist = [[(&quot;p&quot;,false),(&quot;q&quot;,false)],[(&quot;p&quot;,true),(&qu=
ot;q&quot;,false)],<BR>  
[(&quot;p&quot;,false),(&quot;q&quot;,true)],[(&quot;p&quot;,true),(&quot=
;q&quot;,true)]];</FONT> </DIV>
<DIV><FONT color=#000000></FONT> </DIV>
<DIV><FONT color=#000000>val statefuns = map state =
statelist;</FONT></DIV>
<DIV><FONT color=#000000></FONT> </DIV>
<DIV><FONT color=#000000>fun truthtable fmula = map (eval fmula)
statefuns;</FONT></DIV>
<DIV><FONT color=#000000></FONT> </DIV>
<DIV><FONT color=#000000>val nd = fand(lit &quot;p&quot;,lit
&quot;q&quot;);<BR>val or = for(lit &quot;p&quot; lit
```

Some funny stuff here ...

10.2.24

$$\dfrac{\dfrac{\dfrac{X \quad X \to Y}{Y} \quad \dfrac{X \quad X \to (Y \to Z)}{Y \to Z}}{Z} \quad \dfrac{\dfrac{X \quad X \to Y}{Y} \quad \dfrac{X \quad X \to (Y \to (Z \to W))}{Y \to Z \to W}}{Z \to W}}{\dfrac{W}{\dfrac{X \to W}{\dfrac{(X \to Y) \to (X \to W)}{\dfrac{(X \to (Y \to Z)) \to ((X \to Y) \to (X \to W))}{X \to (Y \to (Z \to W)) \to (X \to (Y \to Z)) \to ((X \to Y) \to (X \to W))}}}}}$$

The weighing problem

Weigh four against four. If they don't balance, weigh two (potentially) heavy and two (potentially) light against two normals and a heavy and a light.

1993:10:6

$f(n,r) = f(n-1,r-1) + rf(n-1,r)$

Equivalence relations give rise to partitions

The way to explain how equivalence relations give rise to partitions is to suppose that you start with a set X with an equivalence relation on it. You pick an element of X and stick it in a bin. Thereafter you pick up an object and stick it in the bin (if any) containing things to which it is related. If there aren't any start a new bin. Compscis understand that sort of thing.

****

First check that you know what is meant by the word 'partition'. This matters, since people use it in different ways: number theorists talk of partitions of members of $\mathbb{N}$ (and that stuff is a lot of fun too). However here we are concerned with partitions of *sets*. A partition of a set X is a collection of subsets of X which are disjoint (make sure you know what that means) and between them contain every member of X.

Suppose R is an equivalence relation on a set X. We must find a partition of X that corresponds to R in a natural way. The partition we want is the partition of X into R-equivalence classes.

For $x \in X$, the R-equivalence class of x, (written "$[x]_R$") is $\{y \in X : yRx\}$.

We want to know that the collection of all equivalence classes is a partition of X, that is to say, every $x \in X$ belongs to an equivalence class, and any two equivalence classes are either identical or disjoint. The first is easy: every x belongs to $[x]_R$ (even if it is its sole member). The second needs a bit of work.

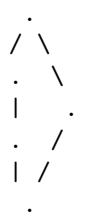
Let us suppose there are two equivalence classes which overlap but are distinct. Then there are x and y (which are not R-equivalent) and there is a z such that $z \in [x]_R \cap [y]_R$. But then xRz and yRz and so on, so by transitivity and symmetry xRy, so x and y belong to the same equivalence class, so $[x]_R = [y]_R$ as desired.

The difficulty with this problem is to work out exactly what it is you have to prove, since once you understand it it is all so obvious. One thing worth making a big fuss about is that the notation '$[x]_R$' for equivalence classes can be a wee bit misleading, because *any* member of an equivalence class can be used to grab it in this fashion: the various x's in it are all, well, equivalent!

1992:11:8

First of all, the word 'cover' in here is a red herring, and we can just as well think of **partitions** of P into antichains. Why? Well, if we have a cover of P by antichains, we can wellorder the cover, and delete from any later member of the cover any element of P that has already appeared in an earlier member of the cover. This may even remove elements of the cover altogether. At all events, we can be sure that there will be at least one cover of minimal size that is a partition, so wlog we can assume that we are dealing with a partition.

"Length of a longest chain in P" is a bit naughty, because it could mean either "maximal chain" or "chain of maximal length". In the partial order whose Hasse diagram is



there are two maximal chains, one of length three (has three elements) and the other of length four(has four elements). However whoever it was that dreamed up this question probably meant "*chain of maximal length*" rather than "*maximal chain*". Let us procede on that assumption.

So what we are trying to prove it that for any (finite!) partial ordering $\langle P, \leq_P \rangle$, the least $n \in \mathbb{N}$ such that there is a covering of P by antichains (call it n_0) is also the length of a chain of maximal length in P (call that n_1).

If Π is a partition of P into antichains, and C is a chain in P, clearly every element of Π must meet C, and no two elements of Π can contain the same element of C, so $n_0 \leq n_1$.

There is a **canonical partition** of P into antichains, as follows. The *rank* $\rho(x)$ of an element x of P is defined by recursion on $\leq_P$ by $\rho(x) = \sup$ of $\{(\rho(y)) + 1 : y <_P x\}$. We then partition P according to the ranks of its elements. Think of an element of P of maximal rank, n, say. It must have something below it of rank $n-1$, and so on down. That way we show there is a chain in P of length n. Clearly no chain in P can be longer, so this is of maximal length, namely n_1. Fix one such chain and call it C.

Now let Π be an arbitrary partition of P into antichains. Each antichain must contain at most one element of C: if it contained more it wouldn't be an antichain! So there must be at least as many antichains in Π as there are elements of C, namely n_1. So $n_1 \leq n_0$.

1992:10:8

Part the first: a message from AGT

There are $\binom{r+k-1}{k-1}$ ways to select r fruits from a greengrocer who sells k kinds, so thats it. For $k_i > 0$, put $l_i = k_{i-1}$ and note the $l - i$ sum to $r - k$.

For lower bounds on codes, just pick greedily. Each codeword picked rules out $b =$ size of $B(x, 2e+1)$ where e is number of errors you hope to correct. So you can get a gode of size N/b, where N is total number of words.

Second part:

$n(r, A)$ is one less than the smallest number we cannot express as a sum of r things in whose denominations belong to A.

The tricky part is to work out precisely *how* to prove this triviality. The first thing that leaps to mind is that we should attempt to prove it by induction. The trouble with proving things by induction is that we have to do induction on some integer vbls and (perhaps) UG on others. Therefore, **the first thing to do is to put back all the implicit quantifiers that have been left out**. This gives us

$$(\forall k \in \mathbb{N})(\forall r)(\forall A)(|A| = k \to n(r, A) \leq \binom{k+r}{k}$$

All the initial quantifiers are universal so it doesn't matter which order they are written in. So what do we do? We can do UG on r by fixing r and then proving the theorem by induction on k, or we can prove by induction on k that for all r ... or we can prove it by induction on r.

Commentary on Peter Robinson's proof of Inclusion-Exclusion

This proof is elliptical but perfectly correct. It's elliptical beco's he has a lot of material to get thru' and not much time, and alse beco's he wants you to put in some work. Like me, he is a director of studies and probably believes like me that students should be introduced—very early on—to the idea that Life is Hard. I think its present form is the result of many years of people honing the proof to get it over and done with as quickly as possible: surgeons in the days before anæsthetics had the same problem. I append some comments my supervisees have found helpful.

I'm assuming you understand the result and have some idea of why it should be true. Let's read it. (Look for "The principle states that ..."). Vertical bars either side of a notation for a set form a notation for the number of elements of that set. This is standard. The thing on the left hand side of the equation is the number of things that belong to the union of the A_i. The family of As is **indexed**: each A has a pointer pointing at it from an **index set**—which in this case is called 'S'.

So in English the equation reads something like

> "The number of things in the union of the As is minus the sum—over nonempty subsets T of S—of minus-one-to-the-power-of-the-number-of-things-in-T times the number of things in the intersection of all those As whose subscripts are in T."

Or—plainer still—for each nonempty $T \subseteq S$, take the intersection of all the As whose subscripts are in T (those As pointed to by elements of T) take its cardinality and take it negative or positive depending on whether T has an odd or even number of elements. Add them all together, for all such T, and make the answer positive.

The first thing to take note of is a bit of **overloading**. Primarily we write 'A_s' to denote one of the As, and the subscript is a member of the index set. However we are now going to write 'A_T', where T is a *subset* of S not a member, and this expression will denote the intersection of all the As whose subscripts are in T. It's easy to detect which of these two usages are in play at any one time, beco's the indices themselves are lower-case Roman letters and the *sets* of indices are upper-case Roman letters. This is a common use of the difference between upper and lower case Roman letters. Notice that $A_\emptyset$ is the whole universe of discourse—Ω. This probably bears thinking about, so we pause for a brief digression on the empty disjunction and the empty conjunction. Any disjunction D can always be tho'rt of as $D \vee$ `false`. so the empty disjunction is just `false`. Analogously the empty conjunction is true: any conjunction C can be tho'rt of as $C \wedge$ `true`.

Now **indicator functions**. You will need to know about these for a variety of reasons. For example they crop up in 1b computation theory where they are called **characteristic** functions. Each subset B of Ω has its own indicator function, written 'I_B'. This is the function which (on being given $x \in \Omega$) returns **true** or **false** depending on whether or not $x \in B$. Except that it returns `int`s instead of `bool`s. This piece of **casting** is so that we can use *arithmetic* operations on the truth-values. It's universal practice in machine code Hacky but clever. This ensures that

- $I_{B\cap C}(x) = I_A(x) \cdot I_B(x)$ (PR alludes to this on the line beginning "indicator functions", tho' he leaves out the 'x') and
- $I_{\overline{B}}(x) = 1 - I_B(x)$.

Notice that PR uses a convention (standard, and not explained here) that $\overline{A_s}$ is the complement of A_s. ("Overlines mean complementation").

For the third displayed line (the one beginning with the big Σ) consider what happens when you muliply out things like $(1-a)(1-b)(1-c)(1-d)$: you get $1-$ lots of things like $-abc$ and $+bd$ which are positive or negative depending on the number of factors. "But shouldn't it start with a '$1-$' before the big Σ?" i hear you cry. It should indeed, but that $1-$ is in fact included beco's one of the Ts you sum over is the empty set! Very cunning.

The next nonindented line begins "Now sum over ...". What's going on here? $I_{\overline{A_1}\cap\overline{A_2}\cap\ldots\overline{A_n}}(x)$ looks nasty so just ignore the subscript for the moment. The sum of $I_B(x)$ over all $x \in \Omega$ is simply the number of things in B, or—using the vertical bar notation—$|B|$. This gives the left hand side of the equation on that line.

10.4 Answers to some of Peter Robinson's exercises

Here is why Euclid's algorithm works and what it means. Anything that divides x and y divides $x - y$, so if i want to find HCF(x, y) i should keep repeating the step of replacing the larger of x and y by $|(x - y)|$. The HCF of the pair of numbers in hand is a loop invariant, and when the process stops with both elements the same we have found the HCF.

If the bigger number is *much* bigger than the smaller one then we could end up subtracting the smaller one many times, and we can save ourselves time by conflating lots of these subtractions together by dividing the bigger number by the smaller and keeping only the remainder.

Remember there is no sensible notion of **betweenness** for integers mod n.

10.4.1 Exercises pp. 5-6

4 is the only one i hadn't seen before.

RTP: $5|2^{3n+1} + 3^{n+1}$

Check that it is true for $n = 0$. Suppose true for n, and aspire to deduce it for $n + 1$. For this it will be sufficient to show that the difference $2^{3n+4} + 3^{n+2} - (2^{3n+1} + 3^{n+1})$

is divisible by 5. Collect like terms to get

$2^{3n+4} - 2^{3n+1} + 3^{n+2} - 3^{n+1})$

$2^{3n+4} - 2^{3n+1} + 3^{n+2} - 3^{n+1})$

$7 \cdot 2^{3n+1} + 2 \cdot 3^{n+1}$

$5 \cdot 2^{3n+1} + 2 \cdot (2^{3n+1} + 3^{n+1})$

Both things being added up are multiples of 5.

question 8: see 10.3

10.4.2 Exercises pp. 12-13

1.
   - No: try $(3, 6)(2, 5)$
   - No: try $(2, 4)(2, 5)$
   - Yes. Think: disjoint multisets!

2. Euclid's algorithm gives us

   $57 = 44 + 13$

   $44 = 3.13 + 5$

   $13 = 2.5 + 3$

   $5 = 3 + 2$

   $3 = 2 + 1$

   $2 = 1 + 1$

   We now work downwards to get, one after the other, the numbers on the LHS of these equations expressed as differences of multiples of 44 and 57.

   $13 = 57 - 44$

   $5 = 44 - 3 \cdot 13 = 44 - 3(57 - 44) \qquad = 4 \cdot 44 - 3 \cdot 57$

   $3 = 13 - 2 \cdot 5 = (57 - 44) - 2 \cdot (4.44 - 3.57) \qquad = 7 \cdot 44 - 10 \cdot 57$

   $2 = 5 - 3 = 4 \cdot 44 - 3 \cdot 57 - 7 \cdot 57 + 9 \cdot 44 \qquad = 13 \cdot 44 - 10 \cdot 57$

   $1 = 3 - 2 = 7 \cdot 57 - 9 \cdot 44 - 13 \cdot 44 + 10 \cdot 57 \qquad = 17 \cdot 57 - 22 \cdot 44$

This gives the solution $x = 17; y = -22$. Then

$57 \cdot 17 + 44 \cdot (-22) = 1$

$57 \cdot x \quad + 44 \cdot y \quad = 1$ ... and subtract to get

$57 \cdot (17 - x) = 44 \cdot (22 + y)$. This gives

$(22 + y)/57 = (17 - x)/44$. Call this quantity k. Then

$y = 57k - 22$ and $x = 17 - 44k$ as desired.

Why is k an integer? Beco's 44 and 57 are coprime. 44 divides the LHS and it doesn't divide 57 so it must divide $22 + y$.

3. It has no solution in integers beco's $(1992, 1752) = 24$. The rest of the question is a rerun of the last one.

4. Ternary Euclid. Look at the first two variables—in this case $56x + 63y$. Do a Euclid on them to discover that the HCF is 7. That means that all you are ever going to get out of $56x + 63y$ is a multiple of 7, so replace $56x + 63y$ by $7a$ and solve $7a + 72z = 1$.

5.

6. Use the hint. Any prime factor of N must a a prime bigger than p_n. Any prime must be congruent to 1 or -1 mod 4. N itself is congruent to -1 mod 4 so at least one of its factors is a prime congruent to -1 mod 4, which is to say a prime of the form $4k + 3$, and it is bigger than p_n, n was arbitrary, so there are arbitrarily large primes of this form.

7.

```
fun factor(n,2) = if n<3   then [n]
                  else if n mod 2 = 0 then 2::factor(n div 2,2)
                  else factor(n,3)
   |factor(n,a) = if n<a*a then [n]
                  else if n mod a = 0 then a::factor(n div a,a)
                  else factor(n,a+2);

fun out(0) = ""
   |out(x) = out(x div 10)^chr(48+(x mod 10));

fun foutput(x::[]) = out(x)
   |foutput(x::xs) = out(x) ^ "*" ^ foutput(xs);

fun findfactor(n)  = foutput(factor(abs(n),2));

(* test data: fermat prime 2^(2^5)+1 *)
findfactor(floor(exp(exp(5.0*ln(2.0))*ln(2.0)))+1);
```

8. P. Satangput's `ML` function that implements Euclid

```
fun dogcd(m,0,a,b,c,d) = [a,b,m]
  | dogcd(m,n,a,b,c,d) = dogcd(n ,m mod n,c,d,a-c*(m div n), b-d*(m div n));

fun gcd(m,n) = dogcd(m,n,1,0,0,1);
```

9. (a) Show that $f_{n+k} = f_k \cdot f_{n+1} + f_{k-1} \cdot f_n$.
   Let's take the hint. (Bear in mind that in any problem with lots of integer variables there may be only one variable you can do the induction on!)

We shall prove by induction on k that for all n, $f_{n+k} = f_k \cdot f_{n-1} + f_{k-1} \cdot f_n$. Best to check first that this is true for $k = 1$. If we take $f_0 = f_1 = 1$ this becomes $f_{n+1} = f_{n-1} + f_n$ which is of course the recursive declaration of the Fibonacci numbers.

Suppose true for k. Then $f_{n+k+1} = f_{n+k} + f_{n+k-1}$. Rearrange this last term on the RHS to $f_{(n-1)+k}$ (the induction hypothesis is that the identity holds for *all* n!) to get

$$\begin{aligned} f_{n+k+1} &= f_{n+k} + f_{(n-1)+k} \\ &= f_k \cdot f_n + f_{k-1} \cdot f_n + f_k \cdot f_n + f_{k-1} \cdot f_{n-1} \\ &= f_k \cdot (f_{n-1} + f_n) + f_{k-1} \cdot (f_n + f_{n-1}) \\ &= f_k \cdot f_{n+2} + f_{k-1} \cdot f_{n+1} \end{aligned}$$

as desired.

(b) First we prove by induction on n that $(\forall l)(f_n | f_{n \cdot l})$. It's true for $l = 0$. We want $f_n | (f_{n \cdot l + l})$. Using part one to expand the RHS we reduce the problem to showing $f_n | (f_n \cdot f_{n \cdot l + 1} + f_{n-1} \cdot f_{n.l})$. Obviously $f_n | f_n \cdot f_{n \cdot l + 1}$ and $f_n | f_{n-1} \cdot f_{n.l}$ by induction hypothesis.

(c) ("Deduce also that $(f_m, f_n) = (f_{(m-n)}, f_n)$"), substitute $m - n$ for n and n for k in part one to get

$$A : f_m = f_n \cdot f_{(m-n)+1} + f_{(n-1)} \cdot f_{(m-n)}.$$

Then for any x, if $x | f_n$ and $x | f_{(m-n)}$ then x divides $f_n \cdot f_{(m-n)+1} + f_{(n-1)} \cdot f_{(m-n)}$ and therefore $x | f_m$. One such x is $(f_{(m-n)}, f_n)$ whence $(f_{(m-n)}, f_n) | f_m$. $(f_{(m-n)}, f_n) | f_n$ holds anyway giving $(f_{(m-n)}, f_n) | (f_m, f_n)$. For the other direction suppose $x | f_m$ and $x | f_n$. We can rearrange the equation A to get

$$f_m - f_n \cdot f_{(m-n)+1} = f_{(n-1)} \cdot f_{(m-n)}$$

Then x divides the LHS and so $x | f_{(n-1)} \cdot f_{(m-n)}$. This isn't quite what we wanted: we needed $x | f_{(m-n)}$. But we deduce this once we know that x does not divide $f_{(n-1)}$. That will follow once we can show that f_n and $f_{(n+1)}$ are always coprime. That is proved by an induction so easy i sha'n't write it out.

So we have proved $x | f_m$ and $x | f_n$ implies that $x | f_{(m-n)}$. In particular $(f_m, f_n) | f_{(m-n)}$. In any case we have $(f_m, f_n) | f_n$ giving $(f_m, f_n) | (f_{(m-n)}, f_n)$. We have already proved $(f_{(m-n)}, f_n) | (f_m, f_n)$ so we infer $(f_m, f_n) = (f_{(m-n)}, f_n)$.

(d) Show that $f_m \cdot f_n | f_{mn}$ iff $(m, n) = 1$.

10.4.3 Exercises pp. 19-21

1. Every power of 10 is congruent to 1 mod 9. So if we express a number as a sum of multiples of powers of 10 its residue mod 9 is just the sum of the coefficients of the powers of 10 that we have used.

2. Odd powers of 10 are congruent to -1 mod 11, and even powers are congruent to 1 mod 11.

3.

4. The probability is 1. A number is divisible by 99 iff it is divisible by 9 and by 11. Rearranging the digits makes no difference to divisibility by 9, and reversing the digits makes no difference to divisibility by 11.

5. ISBN numbers

6. - Since (11,40) = 1 we can divide thru' by 11 getting $7x \equiv 1 \pmod{40}$. We then use Euclid, but we can do it by inspection. $6.7 = 42 \equiv 2$, so $12.7 = 84 \ (\equiv 4)$ so $14.7 \equiv 8 \bmod 40$ and $13.7 \equiv 1 \bmod 40$. So x is 13.
   - By inspection 54 + 30 = 84 which by a happy accident is 12.7 so $y = 7$. It would be a bit of a bugger otherwise since 12 and 54 aren't coprime.
   - Chinese remainder theorem here ...

7. $20! \cdot 21^{20}$ is $21! \cdot 21^{19}$. We want to use Wilson's theorem and we are doing this mod 23 so we will turn $21! \cdot 21^{19} \pmod{23}$ into

   $\frac{22! \cdot 21^{19}}{22} \pmod{23}$

   and since division is OK as long as the base of the modulus is prime. 22 = -1 (mod 23) and 1/(-1) = -1 so this becomes

   $22! \cdot 21^{19} \cdot (-1) \pmod{23}$

   Now we can use Wilson's theorem to turn 22! into -1. The two minus signs cancel and we are left with 21^{19}. This is $(-2)^{19}$ which in turn is -2^{19} which is $-16 \cdot 16 \cdot 16 \cdot 16 \cdot 8$. 16 = -7 so this is $-7 \cdot 7 \cdot 7 \cdot 7 \cdot 8$. 7^2 is 49 which is 3; 9.8 = 72 which is 3. So i make it -3.

8.

9. 2^8 is congruent to -1 mod 257. The monstrous number is an even power of 2^8 and so is congruent to 1 mod 257.

10. Let's try to factorise $n^7 - n$ and see what happens. It becomes

$$n(n-1)(n+1)(n^2+n+1)(n^2-n+1)$$

    The first three factors guarantee that the product is a multiple of 6. and as long as n is congruent to 0, 1 or 6 mod 7 they ensure it will be a multiple of 7 as well. That leaves only the cases where n is congruent to 2, 3, 4 or 5 mod 7 and we can check by hand that in the even cases $n^2 + n + 1$ is a multiple of 7 and in the odd cases $n^2 - n + 1$ is.

    There is a cuter proof (Thank you, Nicola Whiteoak!). Think about $(n^7 - n) \pmod{7}$. Check that for $n = 1, 2, 3, 4, 5, 6$, $n^7 \equiv n \bmod 7$. Or we can use Fermat's little theorem.

11. $3901 = 83.46$. $\phi(m) = 82.46$. Seek multiplicative inverse of 1997 mod 3772. It is .....

12.

13. a^{k+1} is alleged to be the square root of a mod p as long as $p = 4k + 3$ and is a prime. If this is so then $a^{2\cdot(k+1)}$ ($= a^{2k+2}$—let's abbreviate it to 'b') should be congruent to a mod p. We know at least—from Fermat's little theorem—that $a^{4k+3} \equiv a \pmod{p}$. This gives $a^{4k+4} \equiv a^2 \pmod{p}$ which gives $b^2 \equiv a^2$ ldots not quite working

    If a^{k+1} is a sqrt of a then a^{2k+2} should be a and a^{2k+1} should be 1. Well, a^{4k+2} is 1, by Fermat's little theorem, so a^{2k+1} is at least a sqrt of 1.

$(A \times C) \cup (B \times D) = (A \cup B) \times (C \cup D)$?

Remember, sets are **extensional**: two sets with the same mebers are the same sets. So it will be sufficient to check whether or not these two sets have the same members. We will need to arm ourselves with two functions `fst` and `snd`...

x belongs to the LHS iff

$x \in A \times C \ \vee \ x \in B \times D$

iff $(\texttt{fst}(x) \in A \wedge \texttt{snd}(x) \in C) \vee (\texttt{fst}(x) \in B \wedge \texttt{snd}(x) \in D)$

x belongs to the LHS iff

$\texttt{fst}(x) \in (A \cup B) \wedge \texttt{snd}(x) \in (C \cup D)$

Now we can do some abbreviations:

'$\texttt{fst}(x) \in A$ to 'a'

'$\texttt{fst}(x) \in B$ to 'b'

'$\texttt{snd}(x) \in C$ to 'c'

'$\texttt{snd}(x) \in D$ to 'd'

The two conditions reduce to $(a \wedge c) \vee (b \wedge d)$ and $(a \vee b) \wedge (c \vee d)$ which are clearly distinct.

Question 5 page 15 of the second set

For the first line to imply the second assume the first line, namely that there is $g : A/R \to B/S$ such that $g \circ p = q \circ f$, and assume that a_1 and a_2 satisfy the antecedent of the second line.

So $\langle a_1, a_2 \rangle \in R$. This is the same as saying that $p(a_1) = p(a_2)$. (consider the definition of p). So $g \circ p(a_1) = g \circ p(a_2)$. But $g \circ p = q \circ f$, so substituting $q \circ f$ for $g \circ p$ we get $q \circ f(a_1) = q \circ f(a_2)$ which is to say that $q(f(a_1)) = q(f(a_2))$ which—by definition of q—says that $\langle f(a_1), f(a_2) \rangle \in S$.

For the second line to imply the first line declare g by: $g[a]_R =: [f(a)]_S$. Line 2 tells us that it doesn't matter which element of an equivalence class we consider when trying to determine what g of that equivalence class is, so this definition is legitimate.

10.4.4 Some relevant ML code

```
Please find attached (and copied below) a better prime number builder (see
notes in code).  I'm intrigued as to exactly how this works and why it is
better.  If you have any time before the supervision I would be grateful if
you could have a look at it.

Andrew Rose

(*-----------------------------------------------------*)

(*  You may recall that I sent you some code last term
    which produced prime numbers.
    I thought of a different method using integer
    streams (ML Tick 6 & 6*).
    I asked both versions (old version included) to
    produce the first 887 prime numbers.
    The old version took 11.5 seconds.
    The new version took  1.5 seconds!
    Unfortunately, the new version seems to be very
    space inefficient.  Trying to generate more primes
    with the new version causes it to crash CML (windows
    closes it down!).
       *)


(*-------------Generic Stream Manipulations------------*)

datatype stream = Item of int * (unit->stream);
fun cons (x,xs) = Item(x,xs);
fun head (Item(i,xf)) = i;
fun tail (Item(i,xf)) = xf();
fun makeints n = cons(n, fn()=> makeints(n+1));
fun nth(s,n) = if n=1 then head(s) else nth(tail(s),n-1);

fun ton(n,s,xs) = if head(s)>n then
   xs
      else
   head(s)::ton(n,tail(s),xs);

fun filter f xs = if f(head(xs)) then
      cons(head(xs),fn()=>(filter(f) (tail(xs))))
    else
      filter (f) (tail(xs));
(*-----------------------------------------------------*)



(*-------------Prime Number Generation Code------------*)

fun notdiv n x =
 if (x mod n)=0 andalso n<>x then false else true;
```

```
fun sqr(x:int) = x*x;

fun makeprimes(n) =
  let fun xmakeprimes(n,acc,prms)=
    if n=acc then
      ton(sqr(nth(prms,n+1))-1,prms,[])
    else
      xmakeprimes(n,acc+1,(filter (notdiv (nth(prms,acc+1))) (prms)))
  in
    xmakeprimes(n,1,makeints 1)
  end;
(*-----------------------------------------------------*)




(*-----------Old Prime Number Generation Code----------*)

fun pIsPrime(n) =
 let
  fun xpIsPrime(n,sf) =
   if sf=n then
    true
   else
    if n mod sf = 0 then
     false
    else
     xpIsPrime(n,sf+1)
    (**)
   (**)
  (**)
 in
  xpIsPrime(n,2)
 end
;

fun pBuild(n) =
 let
  fun xpBuild(n,sf,acc) =
   if n=0 then
    acc
   else
    if pIsPrime(sf) then
     xpBuild(n-1,sf+1,sf::acc)
    else
     xpBuild(n,sf+1,acc)
    (**)
   (**)
  (**)
 in
  rev(xpBuild(n,2,[]))
 end
;
```

```
(*-------------------------------------------------*)

------=_NextPart_000_0006_01BD26BA.23FD0940
Content-Type: application/octet-stream;
        name="Primes.ml"
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment;
        filename="Primes.ml"

(*  You may recall that I sent you some code last term
    which produced prime number.
    I thought of a different method using integer
    streams.
    I asked both versions (old version included) to
    produce the first 887 prime number.
    The old version took 11.5 seconds.
    The new version took  1.5 seconds!
    Unfortunately, the new version seems to be very
    space inefficient.  Trying to generate more primes
    with the new version causes it to crash CML (windows
    closes it down!).
                                                        *)


(*-------------Generic Stream Manipulations------------*)

datatype stream = Item of int * (unit->stream);
fun cons (x,xs) = Item(x,xs);
fun head (Item(i,xf)) = i;
fun tail (Item(i,xf)) = xf();
fun makeints n = cons(n, fn()=> makeints(n+1));
fun nth(s,n) = if n=1 then head(s) else nth(tail(s),n-1);

fun ton(n,s,xs) = if head(s)>n then
                        xs
                    else
                        head(s)::ton(n,tail(s),xs);

fun filter f xs = if f(head(xs)) then
                    cons(head(xs),fn()=>(filter(f) (tail(xs))))
                  else
                    filter (f) (tail(xs));
(*-------------------------------------------------*)



(*-------------Prime Number Generation Code------------*)

fun notdiv n x =
        if (x mod n)=0 andalso n<>x then false else true;

fun sqr(x:int) = x*x;
```

```
fun makeprimes(n) =
  let fun xmakeprimes(n,acc,prms)=
    if n=acc then
      ton(sqr(nth(prms,n+1))-1,prms,[])
    else
      xmakeprimes(n,acc+1,(filter (notdiv (nth(prms,acc+1))) (prms)))
  in
    xmakeprimes(n,1,makeints 1)
  end;
(*-------------------------------------------------*)




(*-----------Old Prime Number Generation Code----------*)

fun pIsPrime(n) =
        let
                fun xpIsPrime(n,sf) =
                        if sf=n then
                                true
                        else
                                if n mod sf = 0 then
                                        false
                                else
                                        xpIsPrime(n,sf+1)
                                (**)
                        (**)
                (**)
        in
                xpIsPrime(n,2)
        end
;

fun pBuild(n) =
        let
                fun xpBuild(n,sf,acc) =
                        if n=0 then
                                acc
                        else
                                if pIsPrime(sf) then
                                        xpBuild(n-1,sf+1,sf::acc)
                                else
                                        xpBuild(n,sf+1,acc)
                                (**)
                        (**)
                (**)
        in
                rev(xpBuild(n,2,[]))
        end
;

(*-------------------------------------------------*)
------=_NextPart_000_0006_01BD26BA.23FD0940--
```

```
Joseph Marshall's prime finding and factorization program.

fun DivBy(m,[])=false
  | DivBy(m,l::ls)=if((m mod l)=0) then true
      else DivBy(m,ls);

fun PrimeList(n)=
  let fun PLCalc(n,m,ls)= if(m>n)then ls else
        if DivBy(m,ls) then PLCalc(n,m+1,ls)
                       else PLCalc(n,m+1,m::ls)
  in  PLCalc(n,2,[])
  end;


fun Factorize(n)=
  let val flist=PrimeList(n)
  in
    let fun FCalc(n,[],fl)=fl
      | FCalc(n,pf::pfs,fl)=if((n mod pf)=0) then FCalc(n div
pf,pf::pfs,pf::fl)
                              else FCalc(n,pfs,fl)
    in
      FCalc(n,flist,[])
    end
  end;




(*  ------------------------------------------------  *)
(*                     PRIME FACTORS                  *)
(*                      ANDREW ROSE                   *)
(*                       20/11/97                     *)
(*  ------------------------------------------------  *)

(*  The code below defines the following functions...

pIsPrime        - Tests to see if a number is prime
pBuild(n)       - Builds a list of the first n primes
pFact(n)        - Finds the prime factors of a list
pFactFromList(n)- Finds the prime factors of a list
                  given a list of primes.  If the list
                  of primes is exhausted before all the
                  factors have been foundd then the last
                  factors are found using pFact.

                                                        *)

fun pIsPrime(n) =
        let
                fun xpIsPrime(n,sf) =
                        if sf=n then
                                true
```

```
                        else
                                if n mod sf = 0 then
                                        false
                                else
                                        xpIsPrime(n,sf+1)
                                (**)
                        (**)
                (**)
        in
                xpIsPrime(n,2)
        end
;

fun pBuild(n) =
        let
                fun xpBuild(n,sf,acc) =
                        if n=0 then
                                acc
                        else
                                if pIsPrime(sf) then
                                        xpBuild(n-1,sf+1,sf::acc)
                                else
                                        xpBuild(n,sf+1,acc)
                                (**)
                        (**)
                (**)
        in
                rev(xpBuild(n,2,[]))
        end
;

fun pFact(n) =
        let
                fun xpFact(n,sf,acc) =
                        if sf > n then
                                acc
                        else
                                if (n mod sf) = 0 then
                                        xpFact(n div sf, sf, sf::acc)
                                else
                                        xpFact(n, sf+1, acc)
                                (**)
                        (**)
                (**)
        in
                xpFact(n,2,[])
        end
;

exception BotchUp

fun pFactFromList(n,pList) =
        let
                fun xpFact(n,sf,acc) =
```

```
                        if sf > n then
                                acc
                        else
                                if (n mod sf) = 0 then
                                        xpFact(n div sf, sf, sf::acc)
                                else
                                        xpFact(n, sf+1, acc)
                                (**)
                        (**)
                (**)
                fun xpFactFromList(n,[],acc) = raise BotchUp
                  | xpFactFromList(n,sf::[],acc) =
                        xpFact(n,sf,acc)
                  | xpFactFromList(n,pList,acc) =
                        if hd(pList) > n then
                                acc
                        else
                                if (n mod hd(pList)) = 0 then
                                        xpFactFromList(n div hd(pList), pList, hd(pList)::
                                else
                                        xpFactFromList(n, tl(pList), acc)
                                (**)
                        (**)
                (**)
        in
                xpFactFromList(n,pList,[])
        end
;

--------------4F9F38BD7423--
```

$(A \times C) \cup (B \times D) = (A \cup B) \times (C \cup D)$?

Sets are **extensional**: two sets with the same members are the same set. So let's see which things belong to the LHS and to the RHS. '`fst`' and '`snd`' are the obious operation for taking ordered pairs apart.

$x \in$ LHS iff

$x \in A \times C \vee x \in B \times D$ iff

$(\texttt{fst}\ x \in A \wedge \texttt{snd}\ x \in C) \vee (\texttt{fst}\ x \in B \wedge \texttt{snd}\ x \in D)$

and

$x \in$ RHS iff

$\texttt{fst}x \in (A \cup B) \wedge \texttt{snd}x \in (C \cup D)$ iff

$(\texttt{fst}x \in A \vee \texttt{fst}x \in B) \wedge (\texttt{snd}x \in C \vee \texttt{snd}x \in D)$.

We can introduce some abbreviations to make this legible: abbreviate

'$\texttt{fst}x \in A$ to 'a'

'$\texttt{fst}x \in B$ to 'b'

'$\texttt{snd}x \in C$ to 'c'

'$\texttt{snd}x \in D$ to 'd'

and then the LHS and the RHS become

$(a \wedge c) \vee (b \wedge d)$ and
$(a \vee b) \wedge (c \vee d)$

which are clearly inequivalent.