LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

EDINBURGH UNIVERSITY
POSTGRADUATE EXAMINATION QUESTIONS
IN
COMPUTATION THEORY

1978 - 1988

POSTGRADUATE EXAMINATION QUESTIONS

LFCS Report Series

ECS-LFCS-88-64
(also published as CSR-276-88)

SEPTEMBER 1988

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Copyright © 1988, LFCS

Edinburgh University

Postgraduate Examination Questions

in

Computation Theory

1978 – 1988

Preface

For the past eleven years an informal course of lectures and seminars in Computation Theory has been offered to postgraduate students at Edinburgh University. This course, designed to give first-year Ph.D. and M.Phil. students a suitable grounding for research in this area, was begun in 1977/78 by the Departments of Computer Science and Artificial Intelligence and has been continued since 1979/80 by the Department of Computer Science. In 1977/78 the course consisted of about 100 hours of lectures plus seminars and this has grown to over 170 hours in 1987/88.

The course is divided into three broad sections: Complexity, Programming Methodology and Semantics. Each of these sections contains between three to five topics. These topics vary from year to year, reflecting the interests of lecturers and visiting researchers. Some of the introductory material has in recent years been incorporated into the third- and fourth-year undergraduate syllabus but the more advanced material (about half of the course) is provided for postgraduate students only.

Each year in May there is an informal three-day take-home examination on the material taught in the course. The examination normally contains one question on each topic and students are asked to answer four questions of which at least one should be from each of the three sections of the course. The students are allowed to consult books and course notes but not to collaborate or discuss the questions with others. Each question is supposed to require on the order of six hours to solve. This report contains all of the questions which appeared on these examinations since the course began, organized more or less by topics.

Here is a list of the students who have endured the examination:

1977/78:	Con Bradley, David Rydeheard, Martin Seysen and Glynn Winskel
1978/79:	Luca Cardelli, Luis Damas, Mark Jerrum, Alan Mycroft, Michael Sanderson and Donald Sannella
1979/80:	Gordon Brebner, Li Wei, Brian Monahan and Kevin Mitchell
1980/81:	Rocco De Nicola, Peter Dybjer, Mark Millington and Carl Sturtivant
1981/82:	Flemming Nielson, Barbara Paech, Hanne Riis Nielson and Oliver Schoett
1982/83:	Ilaria Castellani, Andreas Goerdt, K.V.S. Prasad, Brian Ritchie and Allen Stoughton
1983/84:	Tatsuya Hagino, Kim Larsen and Alistair Sinclair
1984/85:	Marek Bednarczyk, David Berry, Martin Illsley, Eugenio Moggi and Mads Tofte
1985/86:	Simon Brown, Mads Dam, Claire Jones, Andreas Knobel, Faron Moller and Sun Yong
1986/87:	Jordi Farrés-Casals, James Harland, Paweł Pączkowski, David Pym, Simon Ambler, Luo Zhaohui and Chris Tofts
1987/88:	Jamie Andrews, Jo Blishen, Julian Bradfield, Carolyn Brown, Douglas Gurr and Leslie Henderson

The lecturers on the course who were responsible for setting examination questions included the following:

Lloyd Allison, Gordon Brebner, Rod Burstall, Gerardo Costa, Martin Feather, Michael Gordon, Robert Harper, Matthew Hennessy, Furio Honsell, Mark

Jerrum, Kyriakos Kalorkoti, Clemens Lautemann, David MacQueen, Ian Mason, Robin Milner, Kevin Mitchell, Eugenio Moggi, Alan Mycroft, Gordon Plotkin, Nick Rothwell, David Rydeheard, Donald Sannella, David Schmidt, Jerry Schwarz, John Scott, Mike Shields, Alistair Sinclair, Mike Smyth, Colin Stirling, Andrzej Tarlecki, Robert Tennent, Les Valiant, Chris Wadsworth and Glynn Winskel.

The course has been organized by a succession of people including:

Rod Burstall, Matthew Hennessy, Robin Milner, Donald Sannella and Les Valiant.

My apologies if anybody has inadvertently been omitted from these lists.

Special thanks to Peter Dybjer, Andreas Goerdt, Eleanor Kerse, Gordon Plotkin and Brian Ritchie for digging up old exam papers, to Robert Harper, Mark Jerrum, Kyriakos Kalorkoti, Robin Milner, Kevin Mitchell, Gordon Plotkin and Colin Stirling for helping with the editing, and to Karen McCall for typing.

Donald Sannella
September 9, 1988

1 Analysis of algorithms

Analysis of algorithms (1988)

Let n be a perfect square, X a set of cardinality n, and $\leq$ a total order on X. The elements of X are arranged into a $\sqrt{n} \times \sqrt{n}$ array (a_{ij}) in which the elements which form each row and column are correctly ordered according to $\leq$; that is to say, the conditions $a_{ij} \leq a_{i,j+1}$ and $a_{ij} \leq a_{i+1,j}$ hold for every legal choice of i and j. Note that the arrangement of elements of within the array gives *partial information* about the total order $\leq$. This prior knowledge may reduce the number of pairwise comparisons required to answer queries involving the total order $\leq$.

Derive good upper and lower bounds on the number of pairwise comparisons required, in the worst case, to perform the following operations.

(a) Sort the set X (i.e., completely determine the total order $\leq$). [Hint: you may need to use the so called 'hook formula', which can be found in a number of references including [Knu].]

(b) Given an element $x \in X$, locate x within the array (a_{ij}).

(c) Given an element x, determine the rank of x within X.

(d) Find the median of the set X. [Note: in this part, it may not be possible to obtain upper and lower bounds which are within a constant factor of each other.]

References

[Knu] D. E. Knuth, *The Art of Computer Programming* **2**: *Sorting and Searching.* Addison-Wesley, 1969.

Analysis of algorithms (1987)

Consider the following probabilistic algorithm for finding the kth largest of a finite set S of integers.

procedure *select*(k, S);
begin
  Choose a uniformly from S;
  $S_L := \{x \in S\colon\ x < a\}$;
  $S_G := \{x \in S\colon\ x > a\}$;
  [[Forming S_L, S_G requires $|S| - 1$ comparisons in total]]
  if $k \leq |S_G|$ **then** *select*(k, S_G)
  else if $k = |S_G| + 1$ **then** output a
  else *select*$(k - |S_G| - 1, S_L)$
end

Analyse, as accurately as you can, the expected number of comparisons performed by *select* when finding the median of a set of n elements. (An *exact* analysis may be impossible or infeasible).

Analysis of algorithms (1986)

Suppose $G = (V, E)$ is an undirected graph, and w a function which assigns a rational "weight" $w(e)$ to each edge e of G. Extend the function w to subgraphs of G by defining the weight of a subgraph H to be the product of the weights of the edges which compose H. Let $M(G, w)$ be the sum of the weights of all perfect matchings in G, and $C(G, w)$ be the sum of the weights of all closed subgraphs of G. (These quantities have a certain significance in statistical mechanics.)

Let $\Gamma = (U, A)$ be an undirected graph. The *prism* with base Γ and height h is the undirected graph with vertex set $U \times \{1, \ldots, h\}$ and edge set

$$\{\{\langle u, i\rangle, \langle v, i\rangle\} : 1 \leq i \leq h \text{ and } \{u, v\} \in A\} \cup \{\{\langle u, i\rangle, \langle u, i+1\rangle\} : 1 \leq i \leq h-1 \text{ and } u \in V\}$$

(a) Let Γ be a *fixed* undirected graph. Present a polynomial time algorithm for computing $M(G, w)$ when G is a prism with base Γ, and w is an arbitrary assignment of rational weights to the edges of G. (Assume any reasonable encoding for rational numbers. Note that the running time of the algorithm may be exponential in the size of Γ.)

(b) A polynomial-time algorithm is known which computes $M(G, w)$ when the input graph G is restricted to be planar. Using this algorithm as a subroutine, design a polynomial-time algorithm for computing $C(G, w)$ when G is planar.

Notes:

- A *perfect matching* is a spanning subgraph in which every vertex has degree 1.
- A *closed subgraph* is a spanning subgraph in which every vertex has even degree (possibly 0).

Analysis of algorithms (1985)

Let Σ be a finite alphabet. Define a relation R on Σ^* as follows. For $x, y \in \Sigma^*$, xRy if and only if y can be obtained from x by one of the following three operations.

i. Inserting a single letter into x
ii. Deleting a single letter from x
iii. Replacing a letter in x by a different letter

Also define $d(x, y) = min\{k \in N;\ \ xR^k y\}$. ($d$ is a metric on Σ^*).

Present an algorithm for computing $d(x, y)$ which runs in time $O(d(x, y) min\{|x|, |y|\})$.

[Hint: as a subgoal find an algorithm which runs in time $O(|x| \cdot |y|)$.]

Analysis of algorithms (1984)

Let n by a positive integer, $X = \{1, \ldots, n\}$ and $f_1, \ldots f_m : X \to X$ be a set of functions from X to itself. The functions $f_1, \ldots, f_m$ are *sparse*, that is to say they differ from the identity function on only a small subset of X. Accordingly, each function f_i, $1 \leq i \leq m$, is specified by listing (in any order) the elements of the set:

$$S_i = \{(x, f_i(x)) : x \in X, f_i(x) \neq x\}$$

Give an algorithm, running in time $O(n + \Sigma_{i=1}^{m} |S_i|)$, for finding the composition $g = f_1 \circ f_2 \circ \cdots \circ f_m$ of a set of functions $\{f_1, \ldots f_m\}$ presented in the above form.

[Note: This problem has a trivial solution, a fact which was not realised when the question was set.]

Analysis of algorithms (1983)

The following is a non-trivial result for the theory of graphs:

Theorem (Lipton and Tarjan) Let G be an undirected, planar graph on n vertices. Then the vertices of G can be partitioned into three sets, U, V and S, such that no edge of G has endpoints in both U and V, neither U nor V contains more than $2n/3$ vertices, and S contains not more than $2\sqrt{2}\sqrt{n}$ vertices. Moreover, an algorithm exists which finds a partition U, V, S satisfying the above in $O(n)$ time.

The theorem suggests a natural recursive divide and conquer approach to solving problems on planar graphs. Investigate this technique by applying it to the following two problems:

PLANAR GRAPH 3-COLOURABILITY

Input: Undirected planar graph G

Output: True if there is an assignment of three colours to the vertices of G such that no two adjacent vertices are assigned the same colour; false otherwise.

COUNTING MATCHINGS IN A PLANAR GRAPH

Input: Undirected planar graph G

Output: The *number* of matchings in G. (A subset M of the edges of G is a *matching* iff no two edges in M are incident at a common vertex.)

Obtain tight upper bounds on the run-time of your proposed recursive algorithms, and compare then with the corresponding bounds for algorithms employing naive "exhaustive search".

Graph algorithms (1982)

Consider the following generalisation of the notion of matching in a graph. Suppose D is a subset of the natural numbers and $G = (V, E)$ is an undirected graph with vertex set V and edge set E. A subset F of E is said to be a *D-factor* iff, for all vertices $v \in V$:

$$|\{e \in F \mid e \text{ incident at } v\}| \in D$$

Note that under this definition, perfect matching and $\{1\}$-factor are synonymous. The decision problem corresponding to this notion is:

D-FACTOR

Input: Undirected graph G.

Output: True iff G contains a D-factor.

Suppose (as in fact is the case) that there exists a subroutine for $\{1\}$-FACTOR which runs in time $O(n^{2.5})$, where n is the number of vertices in the input graph. Given such a subroutine, construct and justify polynomial time algorithms for each of the following specialisations of D-FACTOR:

(a) D is restricted to be of cardinality 1.

(b) D is of the form $\{d \mid d_0 \leq d \leq d_1\}$ for some natural numbers d_0, d_1.

(c) $D = \{1, 3, 5, 7, 9, \ldots\}$.

What are the time complexities of your proposed algorithms for (a)-(c)?

Analysis of algorithms (1982)

Saigon is a game similar to Hanoi, but with the tiresome restriction on the number of towers removed. In (n, t)-Saigon there are n discs of various sizes arranged in t towers. An arrangement of discs is *legal* if no disc rests immediately on top of a disc of smaller size. A *move* is a transition from one legal arrangement to another, which is effected by removing the topmost disc from one tower and placing it on top of another tower. A *game* is a sequence of moves which transforms the initial arrangement, in which all discs are in the first tower, to the final arrangement in which all discs are in the second tower. Let $S(n, t)$ be the minimum number of moves in a game of (n, t)-Saigon.

(a) Prove that $S(n, 4) < 2^{\alpha n^{1/2}}$ for some real number α

(b) Prove that $S(n, 5) < 2^{\alpha n^{1/3}}$ for some real number α

(c) Prove that $S(n, \lfloor \lg n \rfloor)$, for $n \geq 8$, is bounded above by a polynomial function of n.

Graph algorithms (1981)

Let $T = (V, A, r)$ be an undirected tree, rooted at r, with vertex set V and edge set A. Call a matching $M \subseteq A$ of T *proper* if, for all vertices $v \in V - \{r\}$, the following property holds:

either v is covered (saturated) by M
or there is an edge in M which is incident both at the father of v and at some brother of v.

(a) Prove that if M is a proper matching in T then M is a maximum (cardinality) matching in T.

(b) Show how to find a proper matching (and hence a maximum matching) in T in time $O(|V|)$, using depth first search.

(c) Let $G = (V, E)$ be an undirected graph and let $T = (V, A, r)$ be a depth first spanning tree of G. Let $M(G)$ be a maximum matching in G, and $M(T)$ be a proper matching in T. Prove that if r is covered by $M(T)$ then $|M(G)| \leq 2\,|M(T)| - 1$, and that otherwise $|M(G)| \leq 2\,|M(T)|$.
[Hint: Use induction on the number of vertices in G. Consider the subgraphs of G which are spanned by the subtrees of T rooted at the sons of r.]

(d) Use the result of (c) to construct an algorithm which takes as input an arbitrary graph $G = (V, E)$ and produces, in time $O(|E|)$ a matching in G whose cardinality is at least $\frac{1}{2}$ that of a maximum cardinality matching in G.

Analysis of algorithms (1980)

Show that both of the following problems have polynomial time algorithms:

(a) Satisfiability of CNF formulae with at most two literal in each conjunct.

(b) Given three strings $a_1 \ldots a_n$, $b_1 \ldots b_m$, $c_1 \ldots c_{n+m} \in \{0,1\}^*$ to determine whether the set $\{1, \ldots, n+m\}$ can be partitioned into two disjoint subsets X_1, X_2 such that the substring of $c_1 \ldots c_{n+m}$ indexed by X_1 is $a_1 \ldots a_n$, and that indexed by X_2 is $b_1 \ldots b_m$.

Analysis of algorithms (1978)

Derive a comparison algorithm for the problem of finding the median of five elements that has worst-case complexity equal to six.

2 Computational complexity

Computational complexity (1988)

After a particularly successful day at the office, you are convinced that you have designed a polynomially time bounded deterministic Turing machine with the following curious property: when presented with (a reasonable encoding of) a Boolean formula ϕ over $\{\wedge, \vee, \neg\}$ it outputs a list of ten natural numbers, one of which is the correct number of satisfying assignments of ϕ. (Unfortunately, there is no way of telling *which* one.) You inform your best friend of this result, whereupon she reflects for a moment and exclaims, "So P = NP — I knew it!"

Alas, before she is able to provide a proof of this preposterous claim she is spirited away by an evil sorcerer, never to be seen again. Ever eager to boost your list of publications, you are therefore obliged to construct a proof yourself, assuming the existence of the Turing machine described earlier.

[You may find the following ideas helpful:

(a) Consider how a pair ϕ_1, ϕ_2 of Boolean formulae may be combined into a third Boolean formula ψ such that the numbers of satisfying assignments of ϕ_1 and ϕ_2 may be deduced easily from the number of satisfying assignments of ψ.

(b) For a given Boolean formula ϕ, think about the binary tree of formulae formed by setting the values of successive variables in ϕ to True and False. Satisfiability of ϕ may be determined by searching this (rather large) tree: how might the information provided by your Turing machine be used to *prune* the tree to a manageable size?]

Computational complexity (1987)

Let f by a polynomial time transformation from quantified Boolean formulae to strings over $\{0,1\}$. Prove that if, for some polynomial p, the set $\{f(F) \mid F \text{ is true}\}$ contains at most $p(n)$ strings of length n for all n, then P = PSPACE.

Computational complexity (1986)

Find a *direct* polynomial transformation f which maps any conjunctive normal form expression E to some graph G and some integer k such that the number of satisfying assignments of E is equal to the number of non-isomorphic colourings of G using k colours.

Comment on the relevance of your transformation to the classification to the problem of counting graph colourings.

Computational complexity (1985)

Let $P^{NP[log]}$ be the class of polynomial time deterministic Turing Machines (TMs) which makes at most a logarithmic number of calls on an NP oracle during execution.

Show that, for any TM M in $P^{NP[log]}$, there is a polynomial time deterministic TM making *one* call on a #P oracle during execution which accepts its input string ω if an only if M accepts ω.

[Hint: Consider a non-deterministic TM M' which is the same as M, except when an oracle call is made. At each call, it non-deterministically chooses between:

(a) Simulate the oracle computation, then *halt.*

(b) Assume the oracle answers 'yes', and continus M's computation, but "repeated" non-deterministically many times.

(c) Assume the oracle answers 'no', and continue M's computation, but "repeated" non-deterministically many times.]

Computational complexity (1984)

Let EXPTIME [NEXPTIME] be the complexity class which contains precisely those languages which can be recognised by a deterministic [non-deterministic] Turing Machine in time at most exponential in the input size.

(a) Show that if EXPTIME $\neq$ NEXPTIME, then P $\neq$ NP

A language L is said to be *sparse* if, for all n, the number of strings in L of size at most n is bounded by a polynomial in n.

(b) Show that, if a sparse language L is contained in NP, but not P, then EXPTIME $\neq$ NEXPTIME.

Computational complexity (1984)

Given a fixed non-deterministic Turing Machine M which has polynomial run-time, devise a polynomial time transformation which maps each input w for M to a directed graph $G(w)$ such that M accepts w if and only if $G(w)$ has a Hamiltonian circuit. Your transformation should be *direct*, i.e. not involve any intermediate sub-transformations.

Complexity (1983)

It is well-known that the problem of determining whether an undirected graph has a Hamiltonian path is NP-complete. Indeed, the problem remains NP-complete even if the graph is restricted to being "cubic", that is exactly three edges are incident at each vertex.

An undirected graph $G = (V, E)$, with vertex set V and edge set E, is called an *edge graph* if there exists an undirected graph $H = (N, L)$ such that G is isomorphic to the graph with vertex set L and an edge set consisting of all pairs of edges that share a common vertex in H.

A theorem of Krausz states that $G = (V, E)$ is an edge graph if and only if there is a partition of E such that each subset in the partition forms a clique and every vertex in V has incident edges which are in at most two subsets in the partition.

Show that the problem of determining whether an edge graph has a Hamiltonian path is NP-complete.

Complexity (1983)

A Turing Machine (TM) may be augmented with an *oracle*. This allows it to test whether the string stored on a special "oracle" tape is contained in a particular language in one time step. By employing such TMs, new complexity classes may be defined. If X is a class of TMs operating within some resource bound, then X^A is the same as X except that the TM can recoginise the language A in one time step.

Consider the complexity class RP, which consists of all non-deterministic TMs in NP having the additional property that, if they have time complexity n^k for some constant k, then they have either zero or at least 2^{n^k-1} accepting computations. RP may be viewed as the class of polynomial time randomised algorithms.

Show that, for some language D, $\mathrm{P}^D \neq \mathrm{NP}^D = \mathrm{RP}^D$.

[Hint: Construct D (by induction on the size of strings contained in D, or otherwise) in such a way that:

(a) $\{x \mid \exists y.|y| = |x| \text{ and } y \in D\} \notin \mathrm{P}^D$ and

(b) The problem NCOMP (generalised to TMs with D oracle) $\in \mathrm{RP}^D$.]

Complexity (1982)

Consider the following pebble game, played on a directed acyclic graph (DAG), in which the aim is to place a pebble on every node exactly once and end with a pebble on every output node, subject to the rules:

1. A pebble may be placed on a node if either
   i. the node is an input node, or
   ii. all immediate ancestors of the node have pebbles on them.
2. A pebble may be removed from any node at any time.

Show that, given a DAG G and an integer k as input, the problem of determining the existence of a strategy using at most k pebbles is NP-complete.

[Hint: Recall that satisfiability of CNF expressions with exactly three literals per clause is a well-known NP-complete problem. An obvious non-deterministic algorithm is to select true/false values for each variable in turn, and then check the truth of each conjunct in turn. You may care to consider simulating this algorithm by playing the pebble game on an appropriate DAG.]

Complexity (1981)

(a) Give a *direct* polynomial time reduction from NCOMP to CLIQUES [N.B. Here NCOMP is the problem of determining whether a non-deterministic one tape TM M halts in y steps on input x (M, x, y arbitrary). You may not appeal to any known results.]

(b) Let LOGCLIQUES be the problem of determining for an arbitrary graph with (say) n nodes whether it has a clique of size $\lceil \log_2 n \rceil$. Would you expect this problem to be NP-complete?

Complexity (1981)

Let HC be the problem of determining for an arbitrary undirected graph whether it contains a Hamiltonian circuit. Describe an algorithm for this problem that runs in time K^n for all n-node graphs, where K is a fixed constant (independent of the graph and of n).

Now consider space efficient non-deterministic Turing machine algorithms for HC. What is the best non-deterministic space upper bound that you can give for this problem? Why would an upper bound of $o(\sqrt{n})$ surprise you?

Complexity (1980)

(a) Show that each of the following functions is time constructible and space constructible.

i. n^2

ii. 2^n

iii. $n!$

(b) What, if any, is the relationship between each of the following pairs of complexity classes?

i. DSPACE(n^2) and DSPACE(f(n)) where $f(n) = n$ for odd n, and $f(n) = n^3$ for even n.

ii. DTIME(2^n) and DTIME(3^n)

iii. NSPACE(2^n) and DSPACE(5^n)

iv. DSPACE(n) and DTIME($\lceil \log_2 m \rceil^n$)

Complexity (1980)

Consider the following three computational problems:

(a) K-colouring the nodes of a graph

(b) Satisfiability of Boolean formulae in conjunctive normal form and

(c) Hamiltonian circuits in directed graphs.

Give *direct* polynomial time reductions

1. from (a) to (b) and
2. from (b) to (c).

Complexity (1979)

Give a *direct* polynomial time transformation f

$$f\text{: Boolean CNF Formulae} \rightarrow \text{undirected graphs}$$

such that if F is any CNF formula over n variables, then for some k the number of distinct truth assignments that make F true is equal to the number of ways of colouring the graph $f(F)$ with k colours.

[N.B. Here sets of isomorphic colourings are counted as one.]

Complexity (1979)

For appropriately restricted functions f, g, h prove that

$$\text{DTAPE}(f(n)) \subseteq \text{DTIME}(g(n)) \Rightarrow \text{DTAPE}(f(h(n))) \subseteq \text{DTIME}(g(h(n)))$$

Using such arguments prove that for all rational $r > 1$,

(a) $\text{P} \neq \text{DTAPE}(x^r)$

(b) $\text{P} \neq \text{NTAPE}(x^r)$

(c) $\text{NP} \neq \text{DTAPE}(x^r)$

(d) $\text{NP} \neq \text{NTAPE}(x^r)$

Complexity (1978)

Consider the class of regular expressions over the alphabet $\{0,1\}$ which include the operation of squaring in addition to the usual concatenation, union and Kleene closure. (If E is a regular expression then the language denoted by E^2 is $\{w_1w_2 : w_1, w_2 \in L(E)\}$ where $L(E)$ is the language denoted by E.) Let TOT be the problem of deciding whether the language represented by such a regular expression is equal to $\{0,1\}^*$.

Describe in detail how you would obtain a number X for which the following statement is true: "Any Turing machine with tape alphabet {0,1} that

(a) works in space less than 2^{1000}, and

(b) solves TOT for all inputs of size X,

is of size at least 2^{1000}."

3 Algebraic complexity

Algebraic complexity (1988)

Let k be any field, X a finite set of indeterminates over k and F a finite subset of $k(X)$. For $Y \subseteq X$ put

$$L_Y(k,X) = \left\{ \sum_{x \in X-Y} \alpha_x x + g \;\middle|\; \alpha_x \in k, g \in k(Y) \right\}.$$

Define $f_1, \ldots, f_d \in k(X)$ to be *linearly independent modulo* $L_Y(k,X)$ if, for all $\alpha_1, \ldots, \alpha_d \in k$, we have that $\alpha_1 f_1 + \cdots + \alpha_d f_d \in L_Y(k,X)$ if and only if $\alpha_1, \ldots, \alpha_d = 0$. Put

$$\delta_Y(F,k,X) = \max\{d \mid \exists f_1, \ldots, f_d \in F \text{ which are l.i. modulo } L_Y(F,k,X)\}.$$

Let $A_Y(k,X)$ be the set of all isomorphisms $\sigma\colon k(X) \to k(X)$ which are the identity on $k(Y)$. Put

$$\mu_Y(F,k,X) = \min\{\delta_\emptyset(F^\sigma, k(X-Y), Y) \mid \sigma \in A_Y(k,X)\}.$$

Show that for all $Y \subseteq X$

$$L(F) \geq \delta_Y(F,k,X) + \mu_Y(F,k,X).$$

Algebraic complexity (1987, 1986)

The first part of this question is much harder than the second and carries twice the marks. The two parts are independent.

Let k be an infinite field and $p_0, \ldots, p_m, q_0, \ldots, q_n, t$ be indeterminates over k. Put

$$p = \sum_{i=0}^{m} p_i t^i,$$

$$q = \sum_{i=0}^{n} q_i t^i,$$

$$pq = \sum_{i=0}^{m+n} c_i t^i.$$

For $0 \leq l \leq u \leq m+n$ and $\Gamma \subseteq \{0, \ldots, m+n\}$ let $P(m,n,l,u,\Gamma)$ denote the non-scalar complexity of computing $c_l, \ldots, c_u$ given that each c_i, for $i \in \Gamma$, has already been computed and can be used without extra charge.

Let $x_0, x_1, \ldots$ be new indeterminates over k and put

$$X = \sum_{i=0}^{\infty} x_i t^i,$$

$$Y = X^{-1} = \sum_{i=0}^{\infty} y_i t^i.$$

(Regard X as a unit of $k(x_0, x_1, \ldots)[[t]]$ and take its inverse in this ring.) For $n \geq 1$ put

$$T_n = L(y_0, \ldots, y_{n-1}),$$

where computations are carried out with scalars k and "inputs" $x_0, \ldots, x_{n-1}$.

1) Show that

$$\begin{aligned} T_{4n} \leq T_{2n} &+ P(n-1, 2n-1, 0, 3n-2, \emptyset) \\ &+ P(4n-2, 4n-1, 2n, 4n-1, \{0, \ldots, 2n-1\}) \end{aligned}$$

2) Let q be as above and put

$$q^2 = \sum_{i=0}^{2n} d_i t^i,$$

$$S_r = L(d_0, \ldots, d_{r-1}),$$

where $1 \leq r \leq 2n+1$, and computations are carried out with scalars k and "inputs" $q_0, \ldots, q_n$. Show that

$$T_n \geq S_{n-2}.$$

(*Hint*: you will find it helpful to consider an optimal algorithm $\mathcal{A}$ for computing $y_0, \ldots, y_{n-1}$ and replace x_0 with some suitable scalar α. Alternatively treat x_0 as a scalar so that computations are carried out with scalars $k(x_0)$ and "inputs" $x_1, \ldots, x_{n-1}$.)

Algebraic and axiomatic complexity (1985)

Let k be an infinite field, $X = \{x_1, \ldots, x_n\}$ a set of indeterminates over k and $\mathcal{A}$ a straight line algorithm over $k(X)$ given k, X. For any $S \subseteq k(X)^n$ say that $\mathcal{A}$ *admits substitutions* s if for all $(s_1, \ldots s_n) \in S$ the substitution σ defined by $\sigma(x_i) = s_i$, $1 \leq i \leq n$ is suitable for $\mathcal{A}$ (i.e. replacing each "input" x_i of $\mathcal{A}$ with $\sigma(x_i)$ does not cause a division by zero). Define the *L^S-complexity*, $L^S(F)$, of $F \subseteq k(X)$ to be

$$\min\{L(\mathcal{A}) \mid \mathcal{A} \text{ computes } F \text{ and admits substitutions } S\}$$

if such an $\mathcal{A}$ exists and ∞ otherwise.

(a) State a necessary and sufficient condition for $L^S(F)$ to be finite. (The condition should depend only on the elements of F).

(b) Explain briefly why
$L^S(f, \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}) \leq 3L^S(F)$
holds for any $f \in k(X)$ and $S \subseteq k(X)^n$ (See chapter 4 of [Kal]).

From now on let

$$R = \{A \in k^{n^2} \mid A \text{ is an invertible matrix}\}$$

(identify $n \times n$ matrices with n^2-tuples by writing the rows of a matrix one after the other).

Let $I(n)$ be the L^R-complexity of computing the inverse of arbitrary invertible $n \times n$ matrices. Let $M(n)$ be the ordinary L-complexity of computing AB for arbitrary $n \times n$ matrices A, B.

(c) Find the inverse of the matrix

$$\begin{pmatrix} 0 & 0 & I \\ 0 & I & B \\ I & A & 0 \end{pmatrix}$$

where A, B are arbitrary $n \times n$ matrices and I is the $n \times n$ identity matrix. Deduce that

$$I(n) \geq c\, M(n)$$

for some constant $c > 0$.

The *trace* of an $n \times n$ matrix $A = (a_{ij})$ is given by

$$tr(A) = \sum_{i=1}^{n} a_{ii}$$

Let $T(n)$ be the L^R-complexity of computing $tr(A^{-1})$ of an arbitrary $n \times n$ matrix A.

(d) Show that

$$T(n) \geq d\, M(n)$$

for some constant $d > 0$.

[Hint: use part b and an "extended" version of part c.]

References

[Kal] K. Kalorkoti. Introduction to algebraic complexity (lecture notes).

4 ML

ML (1988)

The paper [Car] describes the polymorphic typechecking algorithm applied to a simple functional language, and gives an example implementation in an (obsolete) dialect of ML. Your task is to understand and re-implement the typechecker in Standard ML, using the modules facilities.

Break the implementation down into modules, one to provide each of:

- the library routines, utilities and timestamps given on page 12 of the paper;
- the type definitions for `Type` objects, and functions to manipulate them (pages 13–15);
- the environment management functions (page 15);
- the typechecker itself (page 16).

If any of these modules relies on any other, then it should be written as a *functor* with the other module as an argument. If a module does not rely on any other, then it can be implemented as a *structure*. Attach a signature to each module, to minimise interface dependency between them.

The top-level module should be an *abstraction*, to ensure that there are no accidental dependencies on types visible in sub-structures. The signature should look like:

```
signature TYPECHECKER =
   sig
      datatype Term = ...
      datatype Type = ...
         ...

      exception Undefined and Unify
      val AnalyzeTerm: Term -> Type
   end
```

(one or two other datatypes may be necessary so that you can construct terms and manipulate the types provided by this abstraction).

The function `AnalyzeTerm` should typecheck in an initial type environment containing the pervasive identifiers given on pages 2–3 — others (such as `times`) may also be needed. The `Term` datatype should be extended to allow integer constants.

Test your typechecker with the `factorial` function on page 2, the `length` function on page 3, and the specific examples *Ex1*, *Ex2* and *Ex3* on pages 5–6.

References

[Car] L. Cardelli. Basic polymorphic typechecking. *Polymorphism — The ML/LCF/HOPE Newsletter* 2,1 (1985).

ML (1987)

The problem is to write an interpreter for Horn clause logic programs in ML. While grossly inefficient solutions will be marked down, the emphasis should be on clarity and elegance. Your program should be written in terms of abstract types that you shall define; luxuries such as parsing, pretty–printing, and proper error reporting are to be neglected. The problem breaks down into two parts. The first is concerned with building the machinery of terms and substitutions, and the second with the interpreter proper.

Terms, atoms, substitutions

Fix a first–order language $\mathcal{L}$. A *term* t is either a variable v or an n–ary function symbol f applied to n terms $t_1, \ldots, t_n$, written $f(t_1, \ldots, t_n)$. An *atomic formula*, or *atom*, A is an n–ary relation symbol R applied to n terms $t_1, \ldots, t_n$, written $R(t_1, \ldots, t_n)$.

A *substitution* is a function σ from variables to terms which is almost everywhere the identity. The *domain* of a substitution σ is the set of variables x such that $\sigma(x) \neq x$. A substitution σ is extended to a function $\sigma^\sharp$ on terms as follows:

$$\begin{aligned} \sigma^\sharp(v) &= \sigma(v) \\ \sigma^\sharp(f(t_1, \ldots, t_n)) &= f(\sigma^\sharp(t_1), \ldots, \sigma^\sharp(t_n)) \end{aligned}$$

Substitutions are similarly extended to atoms. The superscript $\sharp$ is usually dropped.

Two terms t_1 and t_2 are said to be *unifiable* if there is a substitution σ with domain a subset of the free variables of t_1 and t_2 such that $\sigma(t_1) = \sigma(t_2)$; such a σ is called a *unifier* of t_1 and t_2. A key fact about unifiers is that if two terms have a unifier, then there is a unique (up to renaming of variables) *most general* unifier, written $\mathrm{mgu}(t_1, t_2)$. By "most general" we mean that if $\sigma = \mathrm{mgu}(t_1, t_2)$ and σ' is any other unifier of t_1 and t_2, then $\sigma' = \sigma'' \circ \sigma$ for some substitution σ''. There is a simple, but inefficient, algorithm to determine whether or not t_1 and t_2 are unifiable, and, if so, to return their most general unifier.

The first part of the problem is to define abstract types for terms, atoms, and substitutions, and to write an ML program in terms of these abstractions that implements the mgu algorithm. To keep things simple, use a fixed first–order language $\mathcal{L}$ with, say, a nullary, a unary, and a binary function symbol, and a unary and a binary relation symbol. If you are pressed for time, then you may simply specify, and not implement, the operations on terms, atoms, and substitutions, and concentrate on writing a unification algorithm.

Logic programming

A *program clause*, or *clause*, C, is an $n+1$ tuple of atoms ($n \geq 0$), written $A \leftarrow A_1, \ldots A_n$. A *logic program*, P, is a finite set of program clauses. A *goal*, or *query*, G, is an n–tuple of atoms ($n \geq 0$), written $?A_1, \ldots, A_n$. The empty goal is written $\Box$.

The informal meaning of a clause $A \leftarrow A_1, \ldots, A_n$ is $\forall x_1, \ldots x_k . A_1 \wedge \cdots \wedge A_n \supset A$, where all of the free variables of the clause are among the x_i's. For a fixed logic program P, the informal meaning of a goal $G = ?A_1, \ldots, A_n$ is the query "does there exist a substitution σ whose domain is the set of free variables in G such that $\sigma(A_1) \wedge \cdots \wedge \sigma(A_n)$ is a logical consequence of P?" Such a substitution is called an *answer substitution.*

Let P be the following logic program:

$\mathrm{ancestor}(a, b) \leftarrow \mathrm{parent}(a, b)$
$\mathrm{ancestor}(a, b) \leftarrow \mathrm{parent}(c, b), \mathrm{ancestor}(a, c)$

parent(Fred, Pebbles) $\leftarrow$
parent(Wilma, Pebbles) $\leftarrow$
parent(Barney, Wilma) $\leftarrow$
parent(Betty, Wilma) $\leftarrow$

The relation ancestor(a, b) means "a is an ancestor of b," which holds if a is either a parent of b or an ancestor of a parent of b.
The goal ? ancestor(Barney, Pebbles) is a consequence of P (with the empty answer substitution), and the goal ? ancestor$(a,$ Pebbles) is also a consequence of P, with substitutions $a \mapsto$ Fred and $b \mapsto$ Wilma. The goal ? ancestor(Bill, a) is not a consequence of P.

An interpreter for Horn clause logic programs is a program that, given a logic program P and a goal G, attempts to determine whether or not G is a logical consequence of P, and, if so, returns an answer substitution witnessing that fact. Note that no answer substitution need exist, and that it is not recursively decidable whether or not one does exist. Thus any logic programming interpreter may diverge if a given goal G is not a consequence of P.

If G is a consequence of P, then it is always possible to find an answer substitution by applying *SLD resolution*. Let $G = ?A_1, \ldots, A_n$ be a goal and let $C = B \leftarrow B_1, \ldots, B_m$ be a clause such that $\sigma = \text{mgu}(A_1, B)$ exists. The *resolvent* of G and C is defined to be the goal

$$?A_2, \ldots, A_n, \sigma(B_1), \ldots, \sigma(B_n).$$

If A_1 and B fail to unify, then C and G have no resolvent. An important theorem about logic programs is that a goal G is a logical consequence of a program P iff there is a sequence $G = G_1, \ldots, G_n = \square$ of goals, and a sequence $C_1, \ldots, C_{n-1}$ of clauses of P such that G_{i+1} is the resolvent of G_i and C_i for each $1 \leq i \leq n-1$. Such a sequence is called a *refutation* of G. Let $\sigma_1, \ldots, \sigma_{n-1}$ be the sequence of substitutions obtained at each resolution step of a refutation of G. Then the answer substitution determined by this refutation is the composition $\sigma_{n-1} \circ \cdots \circ \sigma_1$ restricted (for the sake of uniqueness) to the free variables of G.

From the point of view of the logic programming interpreter, the problem is to find a refutation of G whenever one exists. In the interest of efficiency many logic programming systems (notably PROLOG) do not have this property: the interpreter may diverge pursuing a sequence of resolutions that does not lead to $\square$. For example, the goal $P(a, c)$ is clearly a consequence of the following program, yet most PROLOG interpreters fail to find an answer substitution.

$P(a, b) \leftarrow$
$P(c, b) \leftarrow$
$P(x, z) \leftarrow P(x, y), P(y, z)$
$P(x, y) \leftarrow P(y, x)$

The problem is that the fourth clause is "starved" by the PROLOG interpeter since every atom that matches the fourth clause also matches the third, and so the fourth is never tried, and the search fails to terminate.

The second part of this problem is to construct an interpreter for Horn clause logic programs defined over the first–order language that you implemented in part one. Your solution must have the property that it will find an answer substitution whenever one exists.

ML (1986)

Design a package to support polynomial arithmetic with the following signature as an interface.

```
signature POLYNOMIAL =
  sig
    type polynomial
    and terms

    exception polynomial of string  (* In case of errors *)

    datatype term = term of { order: int,  coeff: coeff }
         and coeff = int of int | poly of polynomial

    (* Polynomial construction *)
    val polynomial: string * ((int * coeff) list) -> polynomial

    (* Printing *)
    val print: polynomial -> polynomial

    (* Polynomial decomposition *)
    val indeterminate: polynomial -> string
    val terms: polynomial -> terms

    (* Terms decomposition *)
    val first_term: terms -> term
    val rest_terms: terms -> terms
    val is_empty_terms: terms -> bool

    (* Polynomial arithmetic *)
    val + : polynomial * polynomial -> polynomial
    val * : polynomial * polynomial -> polynomial
    val / : polynomial * polynomial -> { quotient : polynomial,
                                         remainder: polynomial }
  end;
```

Design the package so that it uses an efficient representation for both sparse and dense polynomials. The package should be able to handle expressions such as

$$(x^3 + (y^2 + 2y + 3)x^2 + 3x + 7) + (3x^2 + 2x + 3)$$

Discuss how you would extend the package to cater for three or more representations of a polynomial.

ML (1985)

1. Design a function *samefringe* which, given two binary trees x and y, returns **true** if the same values occur in the tips of x and y in the same order, regardless of the

internal structure of x and y, and returns *false* otherwise. The solution should avoid *flattening* a huge tree when it is fringe unequal to the one with which it is being compared.

2. Given an integer n, enumerate without repetition all paraffin isomers with the empirical formula C_nH_{2n+2}.

3. Briefly indicate how your solution to the paraffin problem could be made more efficient.

ML (1984)

It is possible to remove bound variables from functions with curried arguments by the use of combinators. We introduce three combinators, S, K, I, defined by the equations

$$\begin{aligned} S\,f\,g\,x &= f\,x\,(g\,x) \\ K\,x\,y &= x \\ I\,x &= x \end{aligned}$$

We can 'abstract away' a bound variable from an expression E (written $[x]E$) as follows

$$\begin{aligned} [x](E_1\,E_2) &\Rightarrow S\,([x]\,E_1)\,([x]\,E_2) \\ [\mathrm{x}]\,x &\Rightarrow I \\ [\mathrm{x}]\,y &\Rightarrow K\,y \end{aligned}$$

where y is a constant or a variable other than x.

1. Create an abstract data type in ML for simple terms involving arithmetic operations and conditional expressions. Implement the abstraction function and use it to convert definitions of the form "**let** $f\,x = E$" to "$f = [x]\,E$".

2. Introduce the extra combinators B and C, defined by

$$\begin{aligned} B\,f\,g\,x &= f\,(g\,x) \\ C\,f\,g\,x &= f\,x\,g \end{aligned}$$

and the optimisations

$$\begin{aligned} S\,(K\,E_1)\,(K\,E_2) &\Rightarrow K\,(E_1\,E_2) \\ S\,(K\,E_1)\,I &\Rightarrow E_1 \\ S\,(K\,E_1)\,E_2 &\Rightarrow B\,E_1\,E_2 \text{ if no earlier rule applies} \\ S\,E_1\,(K\,E_2) &\Rightarrow C\,E_1\,E_2 \text{ if no earlier rule applies} \end{aligned}$$

For testing purposes, you should check that the term representing

let *fact* $n =$ **if** $0 = n$ **then** 1 **else** $n * \mathit{fact}(n-1)$

is translated to

$S(C(B$ `cond (eq 0))` $1)(S$ `times (B fact (C minus 1)))`

where the constants `cond`, `eq`, `times` and `minus` are the curried versions of the obvious functions.

3. Provide a reduction function for the resulting combinator terms and explain what type of reduction you are performing (e.g. normal order reduction, applicative order reduction, normal graph reduction etc.)

Include details of how to execute your program.

ML (1983)

1. Use the abstract data types `expr` (expressions) and `term` (terms, i.e. expressions with variables) in the program below to write a simple rewrite rule theorem prover to test whether two expressions are equal with respect to a given set of rewrite rules. Give documentation for your program.

2. Suggest, without writing ML code, how your program could be extended to do inferences using induction, e.g. about numbers or lists (data type definitions and types of functions would be helpful).

```
(* Print formatting for lists *)
fun format _ [] = []
  | format [bra, sep, ket] tkll =
    let fun insert sep [] = []
          | insert sep [h] = h
          | insert sep (h::t) = h @ (sep :: insert sep t)
     in bra :: insert sep tkll @ [ket] end;
    (* tkll is a list of list of strings, bra an open bracket, sep a
       separator and ket a close bracket, all strings, result is a list of
       strings e.g. format ["(",",",")"] [["a","b"],["c","d"],["e"]] =
       ["(","a","b",",","c","d",",","e",")"] *)

                          (*****************)
                          (*  Expressions  *)
                          (*****************)
type operator = string;

abstype expr = expr of operator * expr list
with
  fun CONST opr = expr(opr, [])
  fun APP(opr,exl) = expr(opr, exl)
  fun OPOF(expr(opr,_)) = opr
  fun ARGSOF(expr(_,exl)) = exl
  fun eqexpr(expr(opr1,exl1), expr(opr2,exl2)) =
      let fun eqexprs([],[]) = true
            | eqexprs((h1::t1), (h2::t2)) =
                eqexpr(h1,h2) andalso eqexprs(t1, t2)
       in opr1 = opr2 andalso eqexprs(exl1,exl2) end
  fun prexpr(expr(opr,exl)) = opr :: format ["(",",",")"] (map prexpr exl)
end;
```

```
                          (******************)
                          (*  Substitutions  *)
                          (******************)
type variable = string;

type substitution = (variable * expr) list;

exception Get and Put;

fun prsub subst =
    let fun prpair(v,ex) = format ["<",",",">"] [[v], prexpr ex]
     in format ["sub(",",",")"] (map prpair subst) end;

fun get(_, []: substitution) = raise Get
  | get(v, (v1,ex)::subst') = if v = v1 then ex else get(v,subst');
    (* gets the value corresponding to a variable v in a substitution
       e.g. get("b",[("a", ex1), ("b", ex2)]) = ex2 *)

fun put(v, ex, subst) =
    ( let val ex1 = get(v, subst)
       in if eqexpr(ex1, ex) then subst else raise Put end )
    handle Get => (v,ex) :: subst;
    (* puts the pair (v,y) into a substitution, first checking whether
       v is already defined.  Fails if V is differently defined.  *)

                          (***********)
                          (*  Terms  *)
                          (***********)

abstype term = var_t of variable | app_t of (operator * term list)
with
  fun var v = var_t v
  fun const opr = app_t(opr,[])
  fun app(opr,tml) = app_t(opr,tml)
  fun subst subs (var_t v) = get(v,subs)
    | subst subs (app_t(opr,tml)) = APP(opr, map (subst subs) tml)
      (* substitute in term to get an expression *)

  infix &;
  fun f & g = fn x => g(f x);
  fun I x = x;

  exception Match;
  fun match(tm, ex) = (* subst (match(tm, ex)) tm = ex,
                         if possible, if not raise Match  *)
      let fun matchl ((var_t v), ex) subs =
                      (* subst (matchl(tm, ex)) (subst sub tm) = ex,
                         if possible, if not raise Match  *)
              ( put(v,ex,subs) handle Put => raise Match )
```

```
            | matchl ((app_t(opr,tml)), ex) subs =
                ( if OPOF ex = opr then matchall(tml, ARGSOF ex) subs
                  else raise Match )
          and matchall([], exl) = I
            | matchall(h::t, eh::et) = matchl(h, eh) & (matchall(t,et))
       in matchl (tm, ex) [] end

  fun prterm (var_t x) = [x]
    | prterm (app_t(opr,tml)) =
        opr :: format ["(",",",")"] (map prterm tml)
      (* converts a term to a string list for printing *)
end;

                          (*************)
                          (*  Testing  *)
                          (*************)

val A = CONST "A" and B = CONST "b";
fun F(e1, e2) = APP("f", [e1, e2]) and G e = APP("g", [e]);
val FAGB = F(A, G B) and FGBGB = F(G B, G B);
implode(prexpr FAGB); implode(prexpr FGBGB);

val x = var "x" and y = var "y" and a = const "a" and b = const "b";
fun f(e1, e2) = app("f", [e1, e2]) and g e = app("g", [e]);
val fxy = f(x,y) and fgxa = f(g x, a) and fxx = f(x,x);
val fay = f(a, y) and fgby = f(g b, y);
implode(prterm fxy); implode(prterm fgxa); implode(prterm fxx);

implode(prexpr(subst [("x", FAGB)] fgxa));
implode(prsub(match(fxy, FAGB)));
implode(prsub(match(fxx, FAGB))) handle Match => "Fails on match";
implode(prsub(match(fxx, FGBGB)));
```

ML (1982)

You intend to write in ML an interpreter for a subset of ML (with parser and type-checker). Give a plan for your program indicating the main abstract data types you would use and what functions you would define. It is not necessary to specify these formally, but you should convey your intentions by giving good names to functions and giving the type of each function with a brief explanation.

Try to show the overall structure of your design. Don't actually write the program.

ML (1981)

Write a unification algorithm in ML. It need not be efficient, but it should be correct. Submit your program with test data and results. [Some credit will be given for a hand-written program if you are short of time.] Note that the most general unifier of a pair

of terms (if it exists) is the most general term which is a substitution instance of each of them; thus the unifier of $f(a, y)$ and $f(x, g(x))$ is $f(a, g(a))$.

Programming languages and concepts (1980)

Write and test in ML a simple editor for abstract syntax trees. The abstract syntax may be fixed, e.g. that of a trivial programming language, and it should be impossible to create ill-formed trees. Operations should include

- create a tree
- move pointer up or down
- find next occurrence of a sub-tree
- replace sub-tree at pointer by a given tree.

Please include comments and test run results. Give a short program description. [If time is short, write the description rather than finishing the programming.]

Programming languages (1979)

1. Consider the following problem specification:

   > Given a text file consisting of 'words' separated by blanks and newline characters, produce as output a new version of the file such that each line has length at most *LMAX*, no words have been split, and the total number of characters is unchanged.

   This specification is incomplete and vague on some points. Refine it to produce a precise and complete specification.

2. Write programs satisfying the revised specification in Pascal, ML and Prolog.

3. Explain briefly how the differences between languages affect the structure of the programs.

Programming languages (1978)

Write a program to find the longest ascending subsequence (not necessarily without gaps) in an array or list using ML.

Prove the resulting program correct.

Example: 1 0 2 1 3 2 3

[The program should not be grossly inefficient.]

Programming languages (1978)

1. Write a function in ML to solve the following problem:

   The input to the function is a sequence of commands, and the output is a sequence of items, where an item consists of a priority (which is an integer) and a string of characters. The sequences may either be presented externally (i.e. as data to be read and printed) or as internal data structures.

   The program maintains an internal multiset of items which will be referred to as S. (A multiset is like a set but permits duplication of items.) The commands may modify S and/or request an item be added to the output sequence.

   The possible commands are

   Reset: S is made empty.

   Add: An item (indicated in the command) is added to S.

   Min: An item with the smallest priority is output and deleted from S.

   Max: An item with the largest priority is output and deleted from S.

   The program will be marked on the following criteria:

   (a) readability and style,

   (b) correctness as determined by examination not by execution (a correct program which cannot be seen to be correct will lose marks),

   (c) efficiency.

   Machine output is not required. Arguments in support of (b) and (c) should be explicitly presented.

2. Discuss the following aspects of the program written in part (1):

   - What were the critical decisions made in the implementation?
   - What alternatives were there and how might the use of these alternatives have influenced the final result?
   - How did the programming language influence the organisation of the program?
   - What kinds of features which were not in ML (e.g. co-routines, objects, persistence) would have been useful? How would they have influenced the program?
   - What kinds of changes in the problem description could be easily handled by your modifications to your program? What kinds would present difficulty?

5 Algebraic specification

Algebraic specification (1988)

The Standard ML core language provides a powerful exception-raising and exception-handling mechanism.

The expression

`raise` *exn* `with` *exp*

raises exception *exn*, passing the value of the expression *exp*. This must occur within the scope of an exception declaration

`exception` *exn* `:` *ty*

which says that exception *exn* passes a value of type *ty*, and the expression *exp* must be of type *ty*. The expression

exp `handle` *exn* `with` *match*

yields the value of *exp* (which may be an exception), unless this is exception *exn* in which case the exception is trapped and the rules in *match* are used to determine the result from the value passed with the exception. See [HMM 86], [Har 86] and [HMT 87] for more information about exceptions in Standard ML.

Suppose that you wish to adopt this exception mechanism for your algebraic specification language. Here is a simple example of a specification you would like your language to handle:

```
val nth : int * 'a list -> 'a
exception nth : string
axiom length(l)<n => nth(n,l) = raise nth with "too high"
axiom n<1 => nth(n,l) = raise nth with "too low"
axiom nth(length(l)+1,l@(a::l')) = a
```

Your first step is to devise an appropriate institution. The expressions from which axioms are built should be allowed to contain both `raise` as in the example above and `handle`. Forget about polymorphic types, imperative side-effects, higher-order functions, and anything else you like (but mention what else you are forgetting and mention why you are forgetting it). You might like to try the simpler case of exceptions which do not pass values first, and then see how value passing can be added.

References

[HMM 86] R. Harper, D. MacQueen and R. Milner. Standard ML. LFCS report ECS-LFCS-86-2 (1986).

[Har 86] R. Harper. Introduction to Standard ML. LFCS report ECS-LFCS-86-14 (1986).

[HMT 87] R. Harper, R. Milner and M. Tofte. The semantics of Standard ML: Version 1. LFCS report ECS-LFCS-87-36 (1987).

Algebraic specification (1987)

Definition: Let SP, SP' be specifications and let $\sigma: Sig[SP] \to Sig[SP']$ be a signature morphism. SP' *implements* SP *via* σ, written $SP \overset{\sigma}{\rightsquigarrow} SP'$, if for every model A' of SP' there is a model A of SP, a subalgebra A'' of $U_\sigma(A')$ and a homomorphism $h: A'' \to A$.

1. Prove that implementations compose vertically, i.e. that $SP \overset{\sigma}{\rightsquigarrow} SP'$ and $SP' \overset{\sigma'}{\rightsquigarrow} SP''$ imples $SP \overset{\sigma' ; \sigma}{\rightsquigarrow} SP''$.

2. Let $\oplus$ be a binary specification-building operation which is monotonic on model classes (so $Mod[SP1'] \subseteq Mod[SP1]$ implies $Mod[SP1' \oplus SP2] \subseteq Mod[SP1 \oplus SP2]$ and $Mod[SP2'] \subseteq Mod[SP2]$ implies $Mod[SP1 \oplus SP2'] \subseteq Mod[SP1 \oplus SP2]$) such that $Sig[SP1 \oplus SP2] =_{\text{def}} Sig[SP1] \cup Sig[SP2]$.

   Suppose that $SP1 \overset{\sigma}{\rightsquigarrow} SP1'$ and $SP2 \overset{\sigma'}{\rightsquigarrow} SP2'$ are implementations; is it the case that $SP1 \oplus SP2 \overset{\sigma''}{\rightsquigarrow} SP1' \oplus SP2'$ for some σ'' (i.e. do implementations compose horizontally wrt $(\oplus)$?

   If yes: prove it.
   If no: give a counter-example, and try to think of some reasonable condition on the semantics of $\oplus$ and/or the specifications involved which would guarantee this desireable property.

Specification (1985, 1986)

This question has four parts.

1. Define an institution appropriate for specifying the *time complexity* of applicative ML programs over the natural numbers running on a VAX 780 (ignore features of ML like exceptions and assume that all programs terminate). Call this institution **TIME**. **TIME**-specifications will contain axioms of the form:

   $$x_1, \ldots, x_n.\ t1 \text{ requires at most } t2 \text{ CPU seconds}$$

   where $x_1, \ldots, x_n$ are variable names and $t1, t2$ are terms built using the functions and constants provided by the program and possibly containing $x_1, \ldots, x_n$. For example:

   $$\forall m, n.\ f(m, n, m) \text{ requires at most } 37 * n \text{ CPU seconds}$$
   $$\forall x, y.\ g(x, h(3, y)) \text{ requires at most } x * t(x, y) + 4 \text{ CPU seconds}$$

   It is difficult to give a completely formal answer to this question; try to be formal where possible but do not go overboard (e.g. do not give a formal definition of the VAX 780 instruction set!). Make any assumptions you think are reasonable.

2. Consider a specification-building operation with the following syntax:

   temporally constrain SP by AX

where SP is a **TIME**-specification and AX is a set of **TIME**-sentences over the signature of SP. Let the denotation $[\![SP]\!]$ of a **TIME**-specification SP be a **TIME**-signature Σ (i.e. an object in the category $\mathbf{Sign}_{\mathbf{TIME}}$) together with a class of Σ-models (objects in the category $\mathbf{Mod}_{\mathbf{TIME}}(\Sigma)$). Informally, the above operation produces a specification having the signature of SP and those models of SP which satisfy the time constraints given in AX. Give a formal semantics for this operation.

3. Consider another specification-building operation with the following syntax:

   expand SP **by opns** $OPNS$

   where SP is a **TIME**-specification and $OPNS$ is a set of operator declarations, e.g. {op1: nat × nat → nat, op2: → nat}. Informally, the above operation produces a specification having as signature the result of adding the operators $OPNS$ to the signature of SP. Give a formal semantics for this operation.

4. The institution **EQ** (equational logic) can be used to specify the functional behaviour of ML programs. Consider another specification-building operation with the following syntax:

   functionally constrain SP **by** AX

   where SP is a **TIME**-specification and AX is a set of **EQ**-sentences over the signature of SP. Informally, the result of the above operation is a **TIME**-specification having as models those models of SP which satisfy the sentences given in AX. Give a formal semantics for this operation.

Specifications (1984)

Suppose $T = \langle \Sigma, E \rangle$ and $T' = \langle \Sigma', E' \rangle$ are equational theories (i.e. E and E' are closed sets of equations) with $T \subseteq T'$; let $i: T \to T'$ be the inclusion and let $S = sorts(\Sigma)$. Let $\_|_i : \mathbf{Mod}(T') \to \mathbf{Mod}(T)$ be the i-reduct functor, with left adjoint $\mathbf{F}: \mathbf{Mod}(T) \to \mathbf{Mod}(T')$; let n and ϵ be the unit and co-unit respectively of the adjunction. We restrict consideration of algebras with countable carrier sets.

Recall that an algebra $A' \in |\mathbf{Mod}(T')|$ is called i-generated iff $\epsilon_{A'}: \mathbf{F}(A'|_i) \to A'$ is surjective.

Suppose $X = \{X_s\}_{s \in S}$ is an S-indexed family of countably infinite sets (of variables). An algebra $A' \in |\mathbf{Mod}(T')|$ is said to contain no junk wrt i iff there exists a map $\varphi: X \to |A'|$ such that $\varphi^\sharp: \mathrm{T}_{\Sigma'}(X) \to A'$ is surjective. That is, for every value $a \in |A'|$ there is a term t with variables of sorts in T only such that t has value a for some assignment of values in A' to variables.

1. Prove that an algebra $A' \in |\mathbf{Mod}(T')|$ is i-generated iff A' contains no junk wrt i.

2. Suppose s is a sort in $sorts(\Sigma') - sorts(\Sigma)$. Give a structural induction principle for s which is sound for i-generated models of T'.

A Σ-generating constraint is an inclusion $i: T \to T'$ such that $sig(T') \subseteq \Sigma$. A Σ-algebra A satisfies a Σ-generating constraint $i: T \to T'$ iff $A|_{sig(T')}$ is i-generated. A specification is a triple $\langle \Sigma, E, C \rangle$ where C is a set of Σ-generating constraints; a model of such a specification must satisfy the constraints as well as the equations.

Consider the specification

$$SP = \langle sig(\text{setUnion}), eqns(\text{SetUnion}), \{\emptyset \hookrightarrow \text{Bool}, \text{Elem} \hookrightarrow \text{Set}\}\rangle$$

where Φ is the empty specification and

Bool =
sorts bool
opns true, false: bool
not: bool → bool
and, or: bool, bool → bool
eqns not(true)=false not(false)=true
$\forall p$:bool. p and true = p $\forall p$:bool. p and false = false
$\forall p$:bool. p or true = true $\forall p$:bool. p or false = p

Elem =
Bool ∪
sorts elem
opns eq: elem, elem → bool
eqns $\forall x$:elem. eq(x, x)=true
$\forall x, y$:elem. eq(x, y)=eq(y, x)
$\forall x, y, z$:elem. eq(x, y) and eq(y, z) and not(eq(x, z))=false

Set =
Elem ∪
sorts set
opns $\emptyset$: set
add: elem, set→set
$\in$: elem, set→bool
eqns $\forall x$:elem. $x \in \emptyset$ = false
$\forall x, y$:elem, S:set. $x \in$ add(y, S)=eq(x, y) or $x \in S$

SetUnion = Set ∪
opns ∪: set, set→set
eqns $\forall S$: set. $\emptyset \cup S = S$
$\forall x$:elem,S, S': set. add$(x, S) \cup S' =$ add$(x, S \cup S')$

3. Using the induction principle in (2) and the usual rules of equational deduction (symmetry, reflexivity, transitivity, substitutivity) prove that all models of SP satisfy the equation $\forall x$:elem,S, S':set. $x \in (S \cup S') = x \in (S' \cup S)$.

Specifications (1983)

1. Explain how the concepts of initiality and freeness are useful in specification languages.

2. Give a specification in approximately the style of Sufrin's paper on editor specifications (in *Science of Computer Programming*, May 1982) for a program to play noughts and crosses (tic-tac-toe). Also define an n-move lookahead strategy for such a program.

Specifications (1982)

Specify a text formatter in CLEAR.

- The input is a sequence of words. Each word has length ≥ 0. A word of length 0 denotes the beginning of a paragraph.
- The output is a sequence of pages, each consisting of *pagelength* lines, where each line has length *linelength*. Lines should be right-justified (i.e. the right margin should be straight, as in a book, not ragged) and as many words as possible should be packed into each line. A new paragraph should begin a new line (the last line of the old paragraph need not be right-justified) and should be indented *indentlength* spaces.

The final specification (and the smaller specifications which make it up) should be as parametric and general as possible and make suitable use of modularisation. Note that *pagelength, linelength* and *indentlength* are parameters of the final specification. Also note that (e.g.) the informal specification does not state that a word is a sequence of characters.

This informal specification is somewhat vague; indicate any design decisions you make.

Try to write a specification, not an algorithm, and to leave some freedom to the implementor (e.g. the informal specification does not say that words in a justified line must be spread out evenly). Assume that theories of natural numbers, sequences etc. with appropriate operators are predefined.

Program Development (1981)

A finite directed graph G (without parallel edges) with a finite set of nodes $node_G$ can be represented by its adjacency function

$$neighbours_G : node_G \rightarrow set\ node_G$$

Assume $node_G$, $neighbours_G$ and the specification ARRAY (of nodes, see below) to be given.

1. Specify the boolean function *path*: $array \rightarrow bool$ by a universally quantified formula using the function $neighbours_G$. $path(p)$ should check whether p is a path of G; i.e. if $\langle get(p,i),\ get(p,i+1)\rangle$ is an edge of G for all i.

2. Specify the function *reach*: $node_G \rightarrow set\ node_G$ by set comprehension using the function *path*. $reach(x)$ should be the set of all nodes which are reachable from node x by some path.

3. (a) Specify the function *embed*: $set\ node_G,\ set\ node_G \rightarrow set\ node_G$ by set comprehension using the function *path*. $embed(S1, S2)$ should be the set of all nodes which are either in $S1$ or which are reachable from a node of $S2$ by a path not containing any node of $S1$.

(b) Show that $embed(\emptyset, \{x\}) = reach(x)$, i.e. *embed* is an embedding of *reach*.

4. (a) Derive equations for $embed(S1, \emptyset)$ and $embed(S1, S2 \cup \{x\})$.
   [Hint: Distinguish $x \in S1$ and $x \notin S1$].

   (b) Use these equations to give a recursive definition of *embed* (in an iterative form) and prove the termination of this algorithm.
   [Hint: Use the choice-operator *some* to choose an element of $S2$].

5. (a) Prove $embed(S1, S2 \cup S3) = embed(embed(S1, S2), S3)$

   (b) Derive two different (non-iterative) recursive algorithms for *embed* corresponding to a "depth first search" and a "breadth first search".
   [Hint: Use (5(a)) and $S2 \cup S3 = S3 \cup S2$]

The specification ARRAY is as follows:

spec ARRAY

based on	NODE_G, NAT
type	*array*
functions	*empty*: *array*
	conc: $node_G$, *array* $\rightarrow$ *array*
	length: *array* $\rightarrow$ *nat*
	get: *array*, *nat* $\rightarrow$ $node_G \cup \{error\}$
laws	$\forall a$: *array* $\forall x$: $node_G$ $\forall n$: *nat*.
	$length(empty) = 0$
	$length(conc(x, a)) = length(a) + 1$
	$get(conc(x, a), 0) = x$
	$get(conc(x, a), n + 1) = get(a, n)$
	$get(empty, n) = error$

end of spec

where *error* is a special element not in $node_G$.

6 Program logics and type theory

Program logics and type theory (1988)

Here are 5 questions about the so-called "intersection type discipline".

1 exam question amounts to 2 of these 5.
2 exam questions amount to 4 out of these 5.

The intersection type discipline is a system S for assigning types to terms of the lambda calculus. S is a system for deriving assertions of the form $\Gamma \vdash M{:}\sigma$.

M, N range over ordinary terms of the lambda calculus. σ, τ range over T, the set of type symbols, Γ ranges over type environments, i.e. sets of pairs consisting of a variable and a type ($\Gamma ::= \emptyset \mid \Gamma, x{:}\sigma$). The set T is defined inductively as follows: let At be a given set of atomic type symbols,

i. $At \subseteq T$
ii. if $\sigma, \tau \in T$ then $\sigma \to \tau, \sigma \cap \tau \in T$

The rules of the system S are:

1. $\dfrac{}{\Gamma \vdash x{:}\sigma}$ provided $x{:}\sigma \in T$ and all the variables in Γ are distinct.

2. (a) $\dfrac{\Gamma, x{:}\sigma \vdash M{:}\tau}{\Gamma \vdash \lambda x.M{:}\sigma \to \tau}$

   (b) $\dfrac{\Gamma \vdash M{:}\sigma \to \tau \quad \Gamma \vdash N{:}\sigma}{\Gamma \vdash MN{:}\tau}$

3. (a) $\dfrac{\Gamma \vdash M{:}\sigma \quad \Gamma \vdash M{:}\tau}{\Gamma \vdash M{:}\sigma \cap \tau}$

   (b) $\dfrac{\Gamma \vdash M{:}\sigma \cap \tau}{\Gamma \vdash M{:}\sigma}$

   (c) $\dfrac{\Gamma \vdash M{:}\sigma \cap \tau}{\Gamma \vdash M{:}\tau}$

The following result holds:

$$(\Gamma \vdash M{:}\sigma \wedge M \to_\beta N) \Rightarrow \Gamma \vdash N{:}\sigma$$

Question 1

Show that N is strongly normalizing iff $\exists \Gamma, \tau$ such that $\Gamma \vdash N{:}\tau$.

[Hint:
$\Rightarrow$ Induction on the structure of normal forms and on the maximum length of a reduction sequence.
$\Leftarrow$ Use the computability method].

Question 2

Encode a natural deduction version of the above formal system S in the logical framework LF. State formally the faithfulness property you conjecture your encoding satisfies.

[Hint: Do not encode object language type environments explicitly.]

Question 3

A number of decidability questions concerning S can be raised.

(a) given M does there exist Γ, τ such that $\Gamma \vdash M{:}\tau$?

(b) given Γ and σ does there exist M such that $\Gamma \vdash M{:}\sigma$?

(c) given Γ, σ, M does $\Gamma \vdash M{:}\sigma$ hold?

Decide at least 2 of the above.

[Hint: One of the above is still an open problem, as far as I know.]

Question 4

Show that if a type is inhabited by a closed term then, interpreting $\cap$ as logical conjunction, the formula corresponding to the type is a theorem of minimal logic.

Is the converse true? If not, how would you extend the calculus and the system in order to achieve the inhabitability of all minimal logic implicational and conjunctive theorems?

Question 5

Consider the system S' consisting of the rules 1, 2(a), 3(a) of the system S and the following:

$$2(\mathrm{b})' \quad \frac{\Gamma, y{:}\sigma \to \tau, x{:}\tau \vdash N{:}\rho \quad \Gamma \vdash M{:}\sigma}{\Gamma, y{:}\sigma \to \tau \vdash N[yM/x]{:}\rho}$$

$$3(\mathrm{b})' \quad \frac{\Gamma, x{:}\sigma, y{:}\tau \vdash M{:}\rho}{\Gamma, z{:}\sigma \cap \tau \vdash M[z/x, z/y]{:}\rho} \quad \text{provided } z \notin dom\ \Gamma$$

$$(\mathrm{cut}) \quad \frac{\Gamma \vdash M{:}\sigma \quad \Gamma, x{:}\sigma \vdash N{:}\tau}{\Gamma \vdash N[M/x]{:}\tau}$$

Compare the system S to the system S', and show that

$$\Gamma \vdash_{S'} M{:}\sigma \Rightarrow \exists N.\Gamma \vdash_{S'-\{\mathrm{cut}\}} N{:}\sigma$$

1. Let τ be the type assignment system for λ-terms, where types are defined by the grammar

$$\sigma ::= \kappa^0 \mid \kappa^1 \mid \sigma \to \sigma'$$

where κ^0 and κ^1 are the only basic constants. Suppose that the congruence[1] relation $\sim$ generated by $\kappa^0 \sim \kappa^1 \to \kappa^0$ and $\kappa^1 \sim \kappa^0 \to \kappa^1$ is defined over types. The rules of the natural deduction presentation of τ are:

   (a) $x:\sigma$
   All assumptions have this form.

   (b)
$$\frac{\begin{array}{c}[\cancel{x:\sigma}] \\ \vdots \\ M:\sigma'\end{array}}{\lambda x.M:\sigma \to \sigma'}$$

   provided all undischarged assumptions of the form $x:\sigma$ in the derivation of $M:\tau$ are discharged by an application of this rule.

   (c)
$$\frac{M:\sigma \to \sigma' \quad N:\sigma}{MN:\sigma'}$$

   provided undischarged assumptions in the derivation of the premises which have the same subject also have the same predicate.

   (d)
$$\frac{M:\sigma \quad \sigma \sim \sigma'}{M:\sigma'}$$

   Prove that if $\Gamma \overset{\tau}{\vdash} M:\sigma$ then M is strongly normalizing.

2. Generalize the above result, studying the issue of strong normalization for recursive type assignment systems. Namely, consider congruence relations $\sim$ generated by exactly one equivalence of the form $\kappa_i \sim \sigma_i$ for each basic constant κ_i $(0 \leq i \leq n)$; try to give a necessary and sufficient condition on the shape of the equivalences generating $\sim$ for the following to hold:

$$\Gamma \overset{\tau}{\vdash} M:\sigma \Rightarrow M \text{ is strongly normalizing}$$

   (There is no need to solve this completely; significant sufficient conditions are enough.)

3. List a number of conjectures you would like to prove or problems you would raise in order to understand the relation between recursive type assignment systems and the Curry type assignment system.

[1] A congruence relation is an equivalence relation compatible with the constructor $\to$, i.e. satisfying the property $(\sigma \sim \sigma', \sigma'' \sim \sigma''') \Rightarrow \sigma \to \sigma'' \sim \sigma' \to \sigma'''$

Intuitionistic type theory (1987)

The purpose of this exercise is to investigate the problem of adding second–order quantification to a Martin–Löf–like type theory. Recall that in the second–order λ–calculus, the type $\forall t.\tau(t)$ is the type of functions that, given a type σ, return a value of type $\tau(\sigma)$. We shall investigate adding such a type constructor to a type theory based on operational semantics.

Define $\forall$ to be a constructor of arity (0)0, and take $\forall(F)$ to be a value for any F. Extend the inference rules of type theory to include: The inference rules for type theory are extended to include:

$$\forall\text{F} \qquad \frac{(t \text{ type})\ F(t) = F'(t)}{\forall(F) = \forall(F')} \quad (t \textit{ new})$$

$$\forall\text{I} \qquad \frac{(t \text{ type})\ a = a' \in F(t)}{a = a' \in \forall(F)} \quad (t \textit{ new})$$

$$\forall\text{E} \qquad \frac{a = a' \in \forall(F)\ A \text{ type}}{a = a' \in F(A)}$$

In keeping with the type–free nature of terms and evaluation type abstraction and type application are implicit. Hypothetical judgements of the form (t type) J are new, and hence must be given semantics as part of this exercise.

The problem breaks down into two parts. First, the methods used to define a type system over an operational semantics must be extended to provide semantics for the $\forall$ type constructor. The principal problem is to overcome the impredicativity of the second–order quantifier in the construction of a type system. Second, semantics must be given to the new hypothetical judgement form, and the above rules must be proved sound with respect to the type system constructed in the first part of the exercise.

Building a Type System with $\forall$ Types

The problem is to construct a type system that gives semantics to the second–order quantifier. The method used in the notes is to define an operator on type systems such that the transfinite iterates form an increasing sequence of type systems. The type system defined by the operator is defined to be the limit of this chain, a fixed point of the operator. Type systems are partially ordered by a relation that expresses the idea that one type system is greater than another if it increases the set of types (by extending the type equality relation and assigning an equality relation to each of the new types) while leaving the equality relation assigned to existing types fixed (see [Har] for the technical details). The latter property is crucial to the construction, for if the equality relation to an existing type were extended, then the operator would fail to be monotone. But since the second–order quantifier is defined by abstraction over *all* types, and since the collection of types is increased at each stage of the iteration of the operator, monotonicity fails, and it is no longer clear that a fixed point exists.

Thus the purpose of this exercise is to devise a type system that gives meaning to the second–order quantifier in such a way that the above rules come out true. The key is to account for the quantification over all types without referring to the set of types available

at any one stage of the iteration. For the sake of simplicity, you may neglect explicit treatment of the other type constructors of type theory, but your solution must be adequate for systems with value–dependent types such as the Π type constructor. Therefore you may not treat the type equality relation separately from the membership relation (as was done in the case of simple type theory.) In order to give an account of the second–order quantifier you shall need to extend the set of values by an infinite set of type variables (so that they evaluate to themselves, thus behaving like base types). Type variables are accounted for in the semantics by parameterizing type systems by type environments that assign p.e.r.'s to type variables; type variables are related only to themselves, and the p.e.r. assigned to a type variable is given by the environment.

Proving the Soundness of the Rules

The second part of the problem is to specify the semantics of the new hypothetical judgement forms $(t \text{ type})A = B$ and $(t \text{ type})a = b \in A$. Since these judgement forms have no prior meaning in type theory, you are free to choose them however you see fit for the soundness of the above rules, provided that the following rule of substitution is validated:

S $$\frac{(t \text{ type})\ F(t) = F'(t) \quad A = A'}{F(A) = F(A')}$$

You may find it useful to contemplate choices for the semantics of the hypothetical judgement form before you tackle the definition of a type system in the previous part.

References

[Har] R. Harper. Constructing type systems over an operational semantics. LFCS report ECS-LFCS-88-59 (1988).

Program logics (1986)

Here is a partition problem. Initially, there are two non-empty sets of integers S_0, L_0. The problem is to partition them into sets S and L so that

i. the maximum of S is less than the minimum of L

ii. S has the same size as S_0 and similarly for L and L_0

iii. the original elements are maintained: $S \cup L = S_0 \cup L_0$.

(a) Design an algorithm for solving this problem which employs two parallel processes SMALL and LARGE: SMALL is to be responsible for the changes from S_0 to S and LARGE for the changes from L_0 to L.

(b) Make explicit any synchronization assumptions between SMALL and LARGE you implicitly appeal to.

Can you think of a good way of formalizing these assumptions?

Constructivism and type theory (1986)

Let RE be the submodel of the graph model given by restricting attention to r.e. sets of natural numbers (cf. [Sco]).

1. Show that a version of Kleene "number realizability" for arithmetic can be defined over RE. [Hint: Take the "probjects" to be members of RE, interpret "probject application" to be application as understood in RE, etc.]
2. Show that Heyting arithmetic is sound with respect to this interpretation.
3. Indicate whether or not CT is true on the interpretation. If not, can you suggest a principle which is slightly weaker than CT and which is true on the interpretation but which is still a plausible expression of Church's thesis?

References

[Sco] D.S. Scott. Data types as lattices. *SIAM J. Computing* 5 (1976).

Program logics (1985)

The following program computes the first component of an array $a[k]$, $1 \leq k \leq m$, which is greater than 0 (if there is one) using two parallel processes.

```
i:=2; j:=1; even:=m+1; odd:=m+1;
(esearch||osearch);
k:=min(even,odd);
```

where

```
esearch = while i<min(odd,even) do
            if a[i]>0 then even:=i else i:=i+2
osearch = while j<min(odd,even) do
            if a[j]>0 then odd:=j else j:=j+2
```

(a) Prove that the program is partially correct (assuming that assignments and evaluation of boolean conditions are atomic actions).

(b) Devise a proof to show that the program is totally correct.

(c) What issues arise when considering total correctness of parallel **while** programs with **await** statements?

Program logics (1984)

(a) Suppose L is a first-order language for arithmetic whose set of terms includes (one-dimensional) array terms of the form $a[t]$ where t is a term. Let PL be the set of **while** programs defined with respect to L which also includes assignments of the form $a[t] := s$ where s, t are terms. Design and prove totally correct a PL algorithm which sorts an array a of arbitrary size $n+1$ (of integers) such that $a[i-1] \leq a[i]$ for $1 \leq i \leq n$. (Assume any truths of arithmetic in your proof.)

(b) Suppose the set of **while** programs defined with respect to some first-order language is augmented by a parallel operator:

$$\alpha \,||\, \beta \text{ is a program whenever } \alpha, \beta \text{ are programs}$$

Given that the intended meaning of || is arbitrary interleaving of atomic actions — where an atomic action is an assignment or an evaluation of a boolean — then what problems do you find when trying to design Hoare rules for || of the following form?

$$\frac{\{A\}\alpha\{B\} \quad \{C\}\beta\{D\}}{\{?\}\alpha \,||\, \beta\{?\}}$$

Program logics (1983)

Suppose L is a first-order language for arithmetic, AL is a standard axiomatization of arithmetic (on L), and PL is the set of **while** programs defined on L. Let HAL be the Hoare logic relative to AL and PL whose set of rules is:

Assig $\quad \{A[t/x]\}x := t\{A\}$

Comp $\quad \dfrac{\{A\}\alpha\{C\} \quad \{C\}\beta\{D\}}{\{A\}\alpha;\beta\{D\}}$

If $\quad \dfrac{\{A \wedge D\}\alpha\{C\} \quad \{A \wedge \neg D\}\beta\{C\}}{\{A\}\textbf{if } D \textbf{ then } \alpha \textbf{ else } \beta \textbf{ fi}\{C\}}$ $\quad D$ is quantifier free

While $\quad \dfrac{\{A \wedge D\}\alpha\{A\}}{\{A\}\textbf{while } D \textbf{ do } \alpha \textbf{ od}\{A \wedge \neg D\}}$ $\quad D$ is quantifier free

Con $\quad \dfrac{\vdash_{AL} A \Rightarrow B \quad \{B\}\alpha\{C\} \quad \vdash_{AL} C \Rightarrow D}{\{A\}\alpha\{D\}}$

Let α be the program:

```
if y=0 then a:=1
else a:=x; b:=y;
     while b>1 do a:=a*x; b:=b-1 od
fi
```

(a) Assuming any theorems of AL and standard definitions of substitution prove that:

i. $\{x \geq 0 \wedge y \geq 0\}\alpha\{a = x^y\}$ is a theorem of HAL.

ii. α is totally correct with respect to (input) $x \geq 0 \wedge y \geq 0$ and (output) $a = x^y$.

(b) Consider ways that one might attempt to define a proof-theoretic notion of soundness (consistency) for HAL. How successful are these? (A proof-theoretic notion of soundness does not appeal to a semantics.)

(c) Let PL be extended to PL' by inclusion of the following clause for variable declarations:

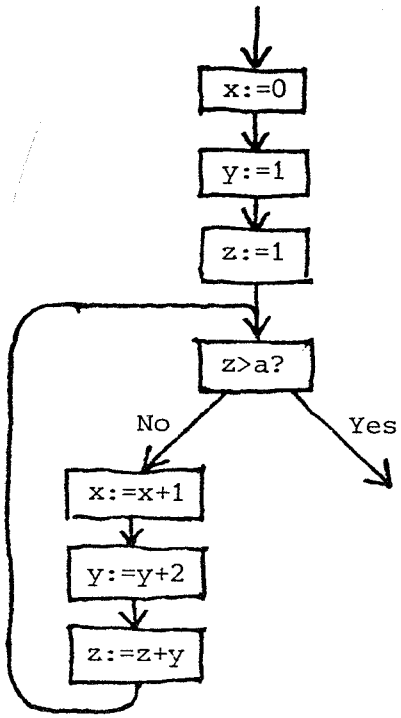
$$\textbf{begin var } x; \alpha \textbf{ end} \in PL' \text{ whenever } \alpha \in PL'$$

i. Add to HAL new proof rules to take account of variable declarations. Call the resulting system HAL'.

ii. Extend the standard valuation semantics of PL for PL'. Using these and standard semantics for AL give a sketch proof that HAL' is (semantically) sound.

Program logics (1982)

1. The following flowchart program sets x to the integer part of the square root of some value a. (Assume all values are natural numbers.)



Using Floyd's method show that the program is partially correct with respect to the input assertion True and the output assertion $x^2 \leq a \wedge (x+1)^2 > a$. [Hint: The variables y and z are used to ensure that $z = (x+1)^2$ always holds at the head of the loop.]

The following applicative program is derived from the flowchart program above, where f and g are functions of type $Nat^+ \times Nat^+ \times Nat^+ \rightarrow Nat^+ \times Nat^+ \times Nat^+$.

```
f(x,y,z) <= g(0,1,1)
g(x,y,z) <= case z>a of
              true:  (x,y,z),
              false: g(x+1,y+2,z+y+2)
              end
```

[Note: If $x0$, $y0$ and $z0 \neq \perp$, $f(x0, y0, z0)$ = the values of the variables after executing the flowchart program starting at the beginning of the program with the variables initially containing $x0, y0, z0$; and $g(x0, y0, z0)$ = the values after execution starting at the head of the loop.] Using computational induction, prove that the 3-tuple $f(x, y, z)$ satisfies the following predicate:

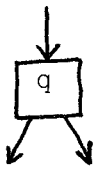
$$\lambda(x, y, z). \quad \text{either } x = y = z = \perp$$
$$\text{or } (x \neq \perp \wedge y \neq \perp \wedge z \neq \perp \wedge x^2 \leq a \wedge (x+1)^2 > a)$$

[Hint: Base your proof on the Floyd proof of the flowchart program.]

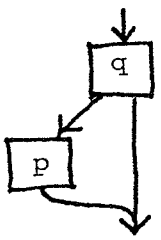
Describe briefly how an Floyd proof of partial correctness of a flowchart program can be transformed into a computational induction proof of a corresponding applicative program.

2. The following is a deduction rule which may be used to prove properties of flowchart programs. Show that the rule is valid for a large class of properties (including the property R in the example below).

Deduction Rule: Let



be any flowchart program with two exits. Given any 1-exit program p, let $F(p)$ denote the composite program

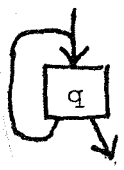


Let R be any property of 1-exit programs. Suppose we can prove the following:

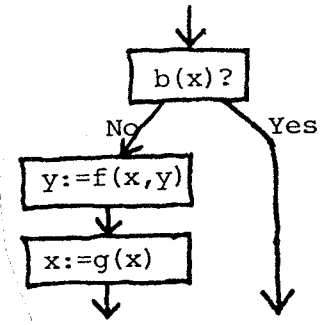
A1 R holds for the program 'loop' which has the property that no execution ever terminates.

A2 Given any 1-exit program p, if R holds for p then it holds for $F(p)$.

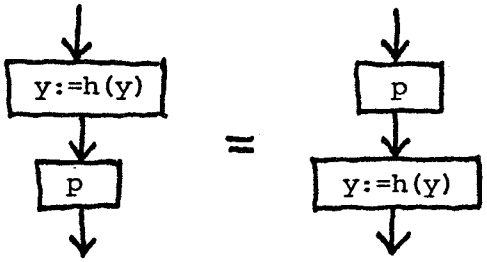
Then we can deduce that R holds for the program



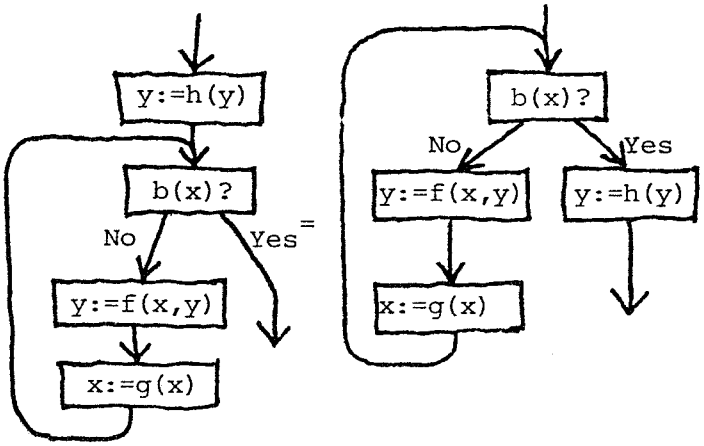
For example, let q be the program



where b is any predicate and f and g are any functions. Let R be the property which holds for a program p if



(where '=' means that, given the same initial values for their variables, either the two programs will both fail to halt, or they will both halt with their variables having the same values). Then it is clear that conditions A1 and A2 are satisfied provided that $f(x, h(y)) = h(f(x, y))$. In this case the rule allows us to deduce that



Program logics (1981)

Consider the problem of a Hoare-like logic for a programming language whose commands are

$$A ::= \textbf{skip} \mid \textbf{break} \mid x := E \mid (A; A) \mid \textbf{if } B \textbf{ then } A \textbf{ else } A \mid \textbf{loop } A$$

The **break** command causes exit from the smallest enclosing **loop** command, which otherwise loops indefinitely. Note that in general a program may therefore terminate in two possible ways: either normally, or via a **break**. (Termination of a **loop** is always normal.)

(a) Formulate an inference system for partial correctness proofs for this language, in terms of sentences

$$\{P\}A\{Q\}\{R\}$$

whose informal interpretation is "if P holds at the start, then if A terminates normally Q will hold, while if A terminates via a **break** then R will hold".

(b) Formulate a soundness theorem for this proof system, in terms of a suitable semantic function $\mathcal{M}$; show in particular that your inference rule for **loop** is sound. (Do not prove the whole theorem.)

(c) Prove, in your system, partial correctness of the following program for checking whether two arrays $X[1..n], Y[1..n]$ are equal:

```
equal:=true; i:=0;
loop (if i<n then i:=i+1 else break;
      if X[i]=Y[i] then skip else (equal:=false; break))
```

(d) Derive the best rule you can for the derived iteration construct

$$\textbf{while } B \textbf{ do } A \stackrel{def}{=} \textbf{loop (if } B \textbf{ then } A \textbf{ else break)}$$

(You should consider this derivation as a preliminary test that your proof system is powerful enough.)

Total correctness of simple programs (1980)

Let L be a many-sorted first-order language including a sort ω and binary relation symbols $\prec$ and $=$ over ω (i.e. of sort $\omega \times \omega$). Let

- x range over variables
- w range over variables of sort ω
- t range over L-terms
- b range over quantifier-free L-formulae
- p, q, r range over L-formulae
- s range over the usual set of simple programs, as given by the grammar:

$$s::=(x:=t) \mid (s;s) \mid (\textbf{if } b \textbf{ then } s \textbf{ else s}) \mid (\textbf{while } b \textbf{ do } s)$$

but restricted so that in assignments "$x:=t$" the sorts of x and t are the same.

Let X be a set of closed L-formulae and consider the following axiomatic system which is designed for proofs of total correctness:

i. p (if p is in X).

ii. $$\frac{p \supset q[t/x]}{\{p\}(x:=t)\{q\}}$$

iii. $$\frac{\{p\}s_1\{r\} \quad \{r\}s_2\{q\}}{\{p\}(s_1; s_2)\{q\}}$$

iv. $$\frac{\{p \wedge b\}s_1\{q\} \quad \{p \wedge \neg b\}s_2\{q\}}{\{p\} \textbf{ if } b \textbf{ then } s_1 \textbf{ else } s_2\{q\}}$$

v. $$\frac{p \supset r \quad \{r \wedge b \wedge (t = w)\}s\{r \wedge (t \prec w)\} \quad (r \wedge \neg b) \supset q}{\{p\} \textbf{ while } b \textbf{ do } s\{q\}}$$

(if w is not free in p, q, r, b, s or t)

Now let $\langle W, \prec_W \rangle$ be a well-founded set (i.e. W is a set, $\prec_W$ is a binary relation over W and there is no infinite descending sequence $w_0 \succ_W w_1 \succ_W \ldots$ of elements of W). A *W-interpretation* I of L is a first-order interpretation I of L where the set of elements of sort ω is W, the equality symbol is interpreted as equality and $\prec$ is interpreted as $\prec_W$.

1. Using a standard denotational or operational semantics, define the *I-total correctness* relation according to the following intuition:

   ($|\!\models_I \{p\}s\{q\}$) iff $\forall$ I-states σ.(if $\sigma \models_I p$ then when started in σ, the program s terminates in a state σ' such that $\sigma' \models_I q$)

   Prove that if $\models_I p$ for every p in X then if $\{p\}s\{q\}$ can be proved in the above system, it follows that $|\!\models_I \{p\}s\{q\}$. (In various subcases you need only consider V and one of (ii)–(iv) as given above).

2. In the following examples, a, b, x, y, i, j, k are variables of sort non-negative integer. By suitable choice of $L, X, W, \prec_W, I$ in each example show that:

   (a) $|\!\models_I \{X \geq 0 \wedge y > 0\}$

   ```
   a:= 0; b:= X;
   while b>=y do b:=b-y; a:=a+1
   ```

   $\{ay + b = x \wedge 0 \leq b < y\}$

   (b) $|\!\models_I \{i \geq 0 \wedge j \geq 0 \wedge k \geq 0\}$

   ```
   while i<>0 or j=0
   do if j=0 then i:=i-1; j:=k; k:=5*k
      else j:=j-1
   ```

   $\{i = 0 \wedge j = 0\}$

Program transformation (1980)

Backus has suggested that second-order functions, such as *maplist* or Iverson's / (called * and ** in HOPE) would be useful for program transformation. Discuss this with examples.

Program proving (1979)

Prove, using intermittent assertions, that the following program computes $x - 10$ if $x > 100$ otherwise 91, and leaves it in z. You need not give a complete formal proof; but state each lemma needed and indicate how it is to be proved, proving one lemma in detail as an example.

```
begin integer a,b;
    a:=x; b:=1;
 L: if a>100 then
       M: if b=1 then z:=a-10; goto HALT fi;
       N: a:=a-10;
          b:=b-1
       else a:= a+11;
            b:= b+1 fi;
    goto L;
  HALT
end
```

Program proving (1979)

The following program sorts a list of items into ascending order using $\leq$:

```
sort: list item -> list item
sort(nil)  -> nil
sort(A::L) -> bubble(A,sort(L))

bubble: item * list item -> list item

bubble(A,nil)  -> A::nil
bubble(A,B::L) -> B::bubble(A,L)  if B<=A
                  A::(B::L)       ifnot
```

Notation: nil represents the empty list and :: represents infix "cons", so $A::L$ represents a non-null list with head A and tail L.

Define a predicate *ordered*,

```
ordered: list item -> truthvalue
```

such that $ordered(L)$ is true iff L is in ascending order w.r.t. $\leq$.

1. Prove that for all lists of items L,

$$ordered(sort(L)) = true$$

Define a function *count*

```
count: item * list item -> number
```

such that $count(A, L)$ counts the number of occurrences of A in the list L.

2. Prove that for all items A and lists of items L,

$$count(A, sort(L)) = count(A, L)$$

The following program also sorts a list of items:

```
qsort: list item -> list item

qsort(nil)  -> nil
qsort(A::L) -> qsort(lessish(A,L)) <> (A::qsort(greater(A,L)))

lessish: item * list item -> list item

lessish(A,nil)  -> nil
lessish(A,B::L) -> B::lessish(A,L) if B<=A
                   lessish(A,L)    ifnot

greater: item * list item -> list item

greater(A,nil)  -> nil
greater(A,B::L) -> greater(A,L)    if B<=A
                   B::greater(A,L) ifnot

<>: list item * list item -> list item

nil <> L2    -> L2
(A::L) <> L2 -> A::(L1<>L2)
```

3. Prove that for all lists of items L,

$$ordered(qsort(L)) = true$$

7 Concurrency

Communicating Systems (1988)

In answering the several questions below, give a rigorous argument in each case. If a counter-example is appropriate, then find the simplest that you can.

1. Let $A \stackrel{\text{def}}{=} a.p.v.b.A$, $S \stackrel{\text{def}}{=} \overline{p}.\overline{v}.S$, and $A_i \stackrel{\text{def}}{=} A[a_i/a, b_i/b]$, $(i = 1, 2)$. Define

$$Sys \stackrel{\text{def}}{=} (A_1 \mid S \mid A_2)\backslash\{p, v\}$$

   (Think of A_1 and A_2 sharing the semaphore S). Further, let $B \stackrel{\text{def}}{=} a.b.B$ and $B_i \stackrel{\text{def}}{=} B[a_i/a, b_i/b]$, $(i = 1, 2)$. Then is it true that $Sys \approx B_1 \mid B_2$? (Recall that $\approx$ stands for weak bisimilarity.)

2. Now re-define A with b and v swapped, i.e. $A \stackrel{\text{def}}{=} a.p.b.v.A$, with no other changes. Is it now true that $Sys \approx B_1 \mid B_2$?

3. Let $\mathcal{S}$ be any binary relation over agents. $\mathcal{S}$ is said to be a *simulation* if $P\mathcal{S}Q$ implies that, for all $\mu \in Act$,

   Whenever $P \xrightarrow{\mu} P'$ then, for some Q', $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $P'\mathcal{S}Q'$.

   Analagous to weak bisimulation we then define the largest simulation, $\prec$, to be the union of all simulations. Prove that $\prec$ is a *pre-order*, i.e. transitive and reflexive.

4. Define the equivalence $\asymp \stackrel{\text{def}}{=} \prec \cap \succ$; that is, $P \asymp Q$ iff $P \prec Q$ and $Q \prec P$. Is $\asymp$ stronger than $\approx$, or weaker, or are they incomparable?

5. Does $P \asymp Q$ imply $P + R \asymp Q + R$?

6. Let $\approx_1$ be *trace-equivalence*; that is, $P \approx_1 Q$ means that for all $s \in \Lambda^*$, $P \stackrel{s}{\Rightarrow}$ iff $Q \stackrel{s}{\Rightarrow}$. What is the relationship between $\asymp$ and $\approx_1$?

7. Going back to (1) and (2) above, does $Sys \asymp B_1 \mid B_2$ hold in each case?

8. Give one reason in favour of $\approx$ over $\asymp$ for practical use.

Theory of communicating systems (1987)

A strong bisimulation S over process agents P may be defined as a relation which satisfies $\mathcal{S} \subseteq \mathcal{F}(\mathcal{S})$, where, for any relation R, $\mathcal{F}(\mathcal{R})$ is defined as follows: $(P, Q) \in \mathcal{F}(\mathcal{R})$ iff, for all $s \in \text{Act}^*$,

i. Whenever $P \xrightarrow{s} P'$ then, for some Q, $Q \xrightarrow{s} Q'$ and $(P', Q') \in \mathcal{R}$

ii. Whenever $Q \xrightarrow{s} Q'$ then, for some P', $P \xrightarrow{s} P'$ and $(P', Q') \in \mathcal{R}$

Note that s ranges over Act^*, so this definition differs from that in Chapter 4 of [Mil].

1. Show that every strong bisimulation S according to this definition is also a strong bisimulation according to the definition in Chapter 4 (Def 3), and conversely.

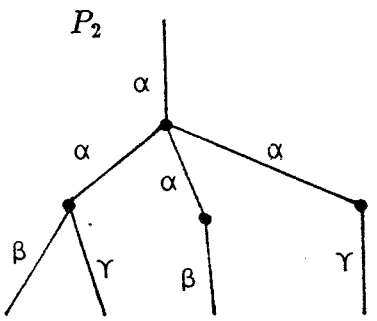
2. Let $\sim_0, \sim_1, \ldots$ be the sequence of relations given by $\sim_0 = \mathcal{P} \times \mathcal{P}$ and $\sim_{i+1} = \mathcal{F}(\sim_i)$ (using the above definition of F). Show that $\sim_{i+1} \subseteq \sim_i$ for each i, and that each $\sim_i$ is an equivalence relation.

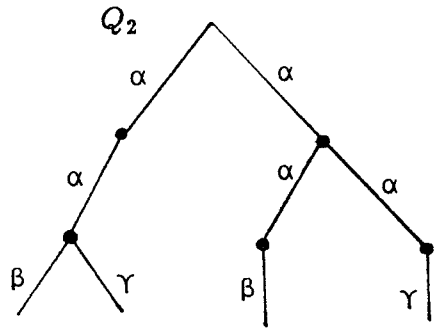
3. Let $\sim_\omega = \bigcap_{i \in \omega} \sim_i$, and let $\sim_{\omega+1} = \mathcal{F}(\sim_\omega)$. It is not true in general that $\sim_{\omega+1} = \sim_\omega$, but it is true provided that each action relation $\xrightarrow{\mu}$ ($\mu \in Act$) is image finite [**Definition:** a relation R is *image finite* if, for each x, $\{y \mid (x,y) \in \mathcal{R}\}$ is finite.] Prove this fact. Also, find an example of a pair (P, Q) of agents such that $P \sim_\omega Q$ but $P \not\sim_{\omega+1} Q$. (Give P and Q as derivation trees).

4. Let the pairs (P_i, Q_i) ($i \in \omega$) be given as follows:

$$P_0 = \beta.NIL \qquad Q_0 = \gamma.NIL$$
$$P_{i+1} = \alpha.(P_i + Q_i) \qquad Q_{i+1} = \alpha.P_i + \alpha.Q_i$$

The derivation trees of P_2 and Q_2, for example, are





By an inductive argument, prove that $P_i \sim_i Q_i$ but $P_i \not\sim_{i+1} Q_i$, for all $i \in \omega$. [**Hint:** you will need a stronger inductive formula than this.]

References

[Mil] R. Milner. *Calculus for Communication and Concurrency.* Prentice Hall (to appear in 1989).

Theory of communicating systems (1986)

1. Describe in detail the significant features of the following three concurrency models: CCS, SCCS, CIRCAL.

2. Explain in detail the differences and similarities of *composition* and *abstraction* in the three models.

3. All three models may be used to describe gate level hardware. Explain the approach adopted in each model to interpret the inherently continuously changing voltages through time.

4. Model an AND gate constructed out of a NAND and inverter in series using each of the three models. Say what assumptions you have used with particular reference to time and propagation delays.

5. Discuss how time is represented in each of the three models. Outline how a description at a fine grain of time may be made "coarser" using a larger time grain in CIRCAL. Can this approach be adopted in CCS and SCCS? Explain.

You will probably need to check through references for CCS [Mil1], SCCS [Mil2] and CIRCAL [Milne] to answer some of these questions.

References

[Mil1] R. Milner. *A Calculus of Communicating Systems.* Springer LNCS, Vol. 92 (1980).

[Mil2] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science* 25 (1983).

[Milne] G. Milne. CIRCAL and the representation of communication, concurrency and time. *ACM Trans. on Programming Languages and Systems* 7, 2 (1985).

Concurrency (1985)

This question is based on [Hen]. It is divided into three parts, each part based on a model for the language T_{Σ^2}.

1. Let SFNT be the Σ^2-po $\langle \text{FNT}, <_s \rangle$ where

   $$t <_s t' \text{ if for every } s \in L(t'),\ \mathcal{A}(t'(s)) \sqsubseteq \mathcal{A}(t(s))$$

   Prove SFNT is initial in the class $\mathcal{C}(SA_2)$ where SA_2 is the set of equations A_2 together with the equation $x \oplus y < x$.

2. **Refusal Sets** (cf. [BHR])
   A finite refusal set R is a finite set of pairs of the form $\langle s, X \rangle$ where $s \in Act^*$ and $X \subseteq Act$ is finite, which satisfies

   i. $\langle \epsilon, \emptyset \rangle \in R$
   ii. $\langle st, \emptyset \rangle \in R$ implies $\langle s, \emptyset \rangle \in R$
   iii. $X \subseteq Y$ and $\langle s, Y \rangle \in R$ implies $\langle s, X \rangle \in R$
   iv. $\langle s, X \rangle \in R$ and $\langle sa, \emptyset \rangle \notin R$ implies $\langle s, X \cup \{a\} \rangle \in R$.

   Let FRef be the set of finite refusal sets.

   (a) Prove $\langle \text{FRef}, \supseteq \rangle$ and SFNT are isomorphic as partial orders.

   (b) Extend FRef to a Σ^2-interpretation so that this isomorphism is also a Σ^2-isomorphism.

3. **Sets of Deterministic Processes**

A deterministic process is a non-empty, finite, prefix-closed subset of Act^* and we use D to denote the set of all deterministic processes. Let NP, the set of non-deterministic processes, be the collection of subsets n of D which satisfy

ci. $d, d' \in n$ implies $d \cup d' \in n$

cii. $d, d' \in n$, $d \subseteq e \subseteq d'$ implies $e \in n$.

NP can be made into a Σ^2-interpretation by defining

i. $\mathrm{NIL}_{\mathrm{NP}} = \{\{\epsilon\}\}$

ii. $a_{\mathrm{NP}}(n) = \{ad \mid d \in n\}$ where $ad = \{as \mid s \in d\} \cup \{\epsilon\}$

iii. $n +_{\mathrm{NP}} n' = \{d \cup d' \mid d \in n, d' \in n'\}$

iv. $n \oplus_{\mathrm{NP}} n' = c(n \cup n')$, where $c(m)$ is the least set containing m which satisfies the closure conditions (ci) and (cii) above.

(a) Prove $\langle \mathrm{NP}, \supseteq, \Sigma^2_{\mathrm{NP}} \rangle$ is a Σ^2-po.

(b) Prove NP is *not* fully abstract with respect to $\sqsubseteq_{\mathrm{MUST}}$ over T_{Σ^1}.

(c) Which equations in Figure 2.4 does NP satisfy?

(d) Find a set of equations E such that NP is initial in $\mathcal{C}(E)$.

References

[BHR] S.D. Brookes, C.A.R. Hoare and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM* 31 (1984).

[Hen] M. Hennessy. *An Algebraic Theory of Processes.* MIT Press (1988).

Concurrency (1984)

A "self-sorting" memory can be designed in terms of individual cells as follows:

- each cell contains at most one value
- the cells are connected together in a chain so that the left-hand neighbour always stores a smaller value.

(a) Give an expression in CCS for such a memory M which

i. accepts values to be stored

ii. will say whether or not it contains a given integer

iii. has bounded capacity

(b) Give an expression S in CCS which captures the overall external behaviour of M, its specification.

(c) Give an *outline* of a proof that M is a correct implementation of S.

(d) Amend the definition of the memory so that in addition it will accept a value which, if it is in the memory, will be deleted.

Concurrency (1983)

Let $\approx$ denote the bisimulation equivalence over CCS terms obtained by using the relations $\stackrel{s}{\Rightarrow}$, defined on page 41 of [Mil].

1. Show that for all terms P, Q,

   $$(\tau P) \mid Q \approx \tau(P \mid Q)$$

   You must prove any non-trivial property of $\approx$ which you use.

2. The dining philosphers problem.

   There are five philosophers each with the same characteristics: they either eat or think. They think standing up and eat sitting down. If they wish to eat they have a designated place at a table: each place has before it a plate and two forks (which are shared with adjacent neighbours). To eat a philospher picks up the fork to the left, then the fork to the right and uses them on the spaghettti from the bowl in the middle of the table.

   (a) Describe this system in CCS. Show that it has a deadlock.

   (b) Describe (in CCS) any of the well-known methods which avoids this deadlock.

   (c) Give a CCS term, SPEC, which describes the overall behaviour of the modified system in (b) (in terms of eating and thinking). Give the outline of a proof that your modified system is equivalent, w.r.t. bisimulations, to SPEC.

   In your modified system it is not necessary to ensure that every philosopher eventually eats.

References

[Mil] R. Milner. *A Calculus of Communicating Systems.* Springer LNCS, Vol. 92 (1980).

Models of parallelism (1983)

Let A be a non-empty set and let $\alpha: I \to 2^A$ be an indexed cover for A. Recall that

$ind(x, y)$ iff $\forall i \epsilon I: x(i) > \epsilon \Rightarrow y(i) = \epsilon$

1. Let $V \subseteq A_\alpha^*$. We shall call V an *abstract machine* iff

   (a) V is left closed in A_α^*.

   (b) $\forall x \in V: \forall a, b \in A: x.a, x.b \in V \wedge ind(a, b) \Rightarrow x.a.b \in V$

   Show that if $V = \&_{i \in I} V_i$, where the V_i are left-closed subsets of the $\alpha(i)^*$, then V is an abstract machine.

2. V is *determinate* iff

   $$\forall x \in V: \forall a, b \in A: x.a, x.b \in V \Rightarrow a = b \vee ind(a, b).$$

If $x \in V$ let V_x denote the set $\{y \in A_\alpha^* \mid x.y \in V\}$.

Show that if V is a determinate abstract machine, then the set of "expressions":

$$V_x \Leftarrow a.V_{x.a} \qquad (x.a \in V)$$

satisfy the conditions of strong confluence and determinacy in CCS.

3. Show that if $ind(u,v)$, then $u.v = v.u$ and that for any x, $x.u.v$ is the least upper bound in A_α^* of $x.u, x.v$. [**Hint**: Examine the coordinates.]

   Show by induction on length(v) that, for V determinate,

   $$x.a.u, x.v \in V \Rightarrow a < u \vee ind(a,u)$$

   where $a \in A$ and $u, v \in A_\alpha^*$. Conclude that if $x, y \in V$, for V a determinate abstract machine, then $x.y \in V$.

   Using the order-theoretic properties of behaviour systems and the representation by vector languages (or otherwise), explain briefly why any abstract machine contains greatest lower bounds for all non-empty sets.

   Complete the proof that:

   Theorem: V is a determinate abstract machine iff V is a left-closed lattice.

4. Let V be determinate and define

   $$V_i = \{x(i) \mid x \in V\}.$$

   Show that the V_i are totally ordered string languages and that

   $$V = \&_{i \in I} V_i$$

   Complete the proof that V is a determinate abstract machine iff V is the vector language of a system of left-closed, totally ordered coinciding sequential processes.

5. What has all this got to do with:

   (a) Marked Graphs[2]

   (b) GE_0-paths

[2] A Petri Net is a Marked Graph if no place in the net has more than one output or input transition, i.e. there is no conflict.

Models of parallelism (1982)

The following program which is taken from [CHP] is designed as a solution to the following problem in process synchronisation.

We have two sets of sequential processes, readers and writers. We also have a critical section which any of these processes may wish to access. The processes must be designed in such a way that:

1. At most one writer may ever be in the critical section at any one time.
2. Any number of readers may be in the critical section at any one time but at no time may a reader *and* a writer be in the critical section at the same time.
3. Writers have priority over readers; once a writer is ready to write, he will be able to do so as soon as possible. In particular, writers waiting to write will be allowed access to the critical section as soon as it is free and no reader will be allowed to use the section thereafter until all writers have finished.

integer readcount, writecount; (initial value = 0)
semaphore mutex1, mutex2, mutex3, w,r; (initial value = 1)

P(mutex3);	P(mutex2);
P(r);	writecount:= writecount + 1;
P(mutex1);	**if** writecount = 1 **then** P(r);
readcount := readcount + 1;	V(mutex2);
if readcount = 1 **then** P(w);	P(w);
V(r);	
V(mutex3);	
⟨reading⟩	⟨writing⟩
P(mutex1);	V(w);
readcount := readcount − 1;	P(mutex2);
if readcount = 0 **then** V(w);	writecount := writecount − 1;
V(mutex1);	**if** writecount = 0 **then** V(r);
	V(mutex2);

For a given fixed number R of readers and W of writers:-

1. Describe a suitable 'state space' for the above program and use it to describe the construction of a transition system which expresses the interleaved behaviour of the system.
2. Devise a predicate expressing the desired mutual exclusive and fixed priority properties of the system such that the predicate is q_0-inductive, where q_0 denotes the initial state of the system. Use this to prove the correctness of the solution. What assumptions must be made about the behaviour of semaphores?
3. Is the program still correct if the processes run in a maximally parallel manner (i.e. the only sequentialisation derives from the sequential nature of the processes and semaphores)? Justify your answer formally in terms of the trace or vector language determined by the transition system together with the independence relation determined by the assumption of maximal parallelism.

References

[CHP] P.J. Courtois, F. Heymans and D.L. Parnas. Concurrent control with "readers" and "writers". *Comm. of the ACM* 14 (1971).

Concurrency (1982)

Let A be a set of actions ranged over by a. Let Σ_0 be the set of operators $\{\mathrm{NIL}, \oplus, a{:}\}$. For each $s \in A^*$ we define $\stackrel{s}{\Rightarrow}$ to be the least relation over finite Σ_0-terms which satisfies

i. $t \stackrel{\epsilon}{\Rightarrow} t$
ii. $a{:}t \stackrel{a}{\Rightarrow} t$
iii. $t \oplus u \stackrel{\epsilon}{\Rightarrow} t$
iv. $u \oplus t \stackrel{\epsilon}{\Rightarrow} t$
v. $t \stackrel{s}{\Rightarrow} t_1, t_1 \stackrel{s'}{\Rightarrow} t_2$ implies $t \stackrel{s \cdot s'}{\Rightarrow} t_2$

Here ϵ is the empty word and $s \cdot s'$ denotes the concatenation of s with s'.

Using these relations we may define a notion of observational equivalence as on page 42 of [Mil]:

i. $t \cong_0 u$ for every t, u
ii. $t \cong_{k+1} u$ if for every $s \epsilon A^*$
  (a) $t \stackrel{s}{\Rightarrow} t'$ implies $u \stackrel{s}{\Rightarrow} u'$ for some u' such that $t' \cong_k u'$
  (b) $u \stackrel{s}{\Rightarrow} u'$ implies $t \stackrel{s}{\Rightarrow} t'$ for some t' such that $t' \cong_k u'$
iii. $\cong \; = \bigcap_{k \geq 0} (\cong_k)$
iv. $t \cong_c u$ if for every Σ_0-context $C[\ ], C[t] \cong C[u]$.

1. (a) Is $\cong$ a Σ_0-congruence, i.e. is it true that $t \cong_c u$ iff $t \cong u$?
   (b) State as many independent axioms as you can for $\cong_c$.
   (c) Give a translation, tr, into CCS such that

   $$t \cong_c u \text{ iff } tr(C[t]) \approx tr(C[u]) \text{ for every } \Sigma_0\text{-context } C[\ ].$$

   (d) For your translation is it true that $t \cong_c u$ iff $C[tr(t)] \approx C[tr(u)]$ for every CCS-context $C[\ ]$?

   (Here $\approx$ denotes observational equivalence as defined on page 43 of [Mil]).

2. An alternative equivalence may be defined as follows:

   $p \simeq q$ if $\forall s \in A^*$

   i. $p \stackrel{s}{\Rightarrow} p' \in \mathrm{Dead}$ implies $q \stackrel{s}{\Rightarrow} q'$ for some $q' \in \mathrm{Dead}$.
   ii. $q \stackrel{s}{\Rightarrow} q' \in \mathrm{Dead}$ implies $p \stackrel{s}{\Rightarrow} p'$ for some $p' \in \mathrm{Dead}$.

   Here $\mathrm{Dead} = \{p \mid p \stackrel{\epsilon}{\Rightarrow} p' \text{ implies } p \sim p'\}$

   (a) Is $\simeq$ a Σ_0-congruence?
   (b) State as many independent axioms as you can for $\simeq$.

(c) Discuss the difference between $\simeq$ and $\cong_c$

3. Extend the alphabet to Σ_1 by adding the binary operator $+$ and extend the semantics by the rule

   vi. $t \stackrel{s}{\Rightarrow} t'$ and $t \neq t'$ implies $t + u \stackrel{s}{\Rightarrow} t'$ and $u + t \stackrel{s}{\Rightarrow} t'$

   (a) Are $\cong$ and $\simeq$ both Σ_1-congruences?

   (b) How does the addition of $+$ affect your axioms? Are they still true? Are there any new axioms?

   (c) State informally the differences between the operators $+$ and $\oplus$.

References

[Mil] R. Milner. *A Calculus of Communicating Systems.* Springer LNCS, Vol. 92 (1980).

Calculus of communicating systems (1981)

1. In CCS, sequential composition of two agents may be defined by

   $$B_1; B_2 \stackrel{def}{=} (B_1[\delta'/\delta] \mid \delta'.B_2)\backslash\delta'$$

   where δ' is a new name. It acts as sequential composition when $\bar{\delta}$ occurs as the last action of B_1 (cf. [Mil] p. 128). Prove from the laws of strong equivalence that ";" is associative, i.e., $B_1; (B_2; B_3) \sim (B_1; B_2); B_3$.

2. Adapt the construction of [Mil] §8.4 to build a priority queue, in which the items stored are pairs (i, v) where i is an integer (the priority) and v a data value. The item at the head of the queue at any time is the item of highest priority which has been longest in the queue. Prove that your construction satisfies a specification analogous to §8.4 equation (3).

   [Note: part (2) carries greater weight in this question.]

References

[Mil] R. Milner. *A Calculus of Communicating Systems.* Springer LNCS, Vol. 92 (1980).

Models of parallelism (1981)

Suppose $\mathcal{N}$ is a net (S, T, F) and $M \subseteq S$ designates a marking (case) of $\mathcal{N}$. A *causal* net $\mathcal{N}' = (S', T', F')$ represents a non-sequential behaviour of $\mathcal{N}$ from M if there is a mapping $f: S' \cup T' \to S \cup T$ with

1. $\forall t \in T': f({}^\circ t) = {}^\circ f(t) \wedge f(t^\circ) = f(t)^\circ$ (t is an occurrence of $f(t)$)
2. $\forall s \in S': {}^\circ s = \emptyset \Rightarrow f(s) \in M$
3. $\forall x, x' \in S' \cup T': f(x) = f(x') \Rightarrow xF'^+x' \vee x'F^+x \vee x = x'$

Characterise (to within isomorphism) the set of all causal nets representing nonsequential behaviour of $\mathcal{N}$ from M, if $\mathcal{N}$ is a connected marked graph which is live and 1-safe w.r.t. the marking M.

Hint: Consider the causal net $\mathcal{C} = (S \cup M', T, F')$ where $M' = \{s' \mid s \in M\}, s \mapsto s'$ is a bijection and

$$\forall s \in S, t \in T: (s,t) \in F \Leftrightarrow (s,t) \in F'$$
$$\forall s \in M, t \in T: (t,s) \notin F'$$
$$\forall s' \in M', t \in T: (t,s') \in F' \Leftrightarrow (t,s) \in F$$

8 Algebras and categories

Algebras and categories (1988)

Definition: Let **T** be an arbitrary category. The category **Into(T)** of functors into **T** is defined as follows:

- objects are functors **F**: $\mathbf{I} \to \mathbf{T}$
- a morphism from **F1**: $\mathbf{I1} \to \mathbf{T}$ to **F2**: $\mathbf{I2} \to \mathbf{T}$ is a pair $\mu = \langle \mathbf{G}, \eta \rangle$, where **G**: $\mathbf{I1} \to \mathbf{I2}$ is a functor and η: $\mathbf{G};\mathbf{F2} \to \mathbf{F1}$ is a natural transformation
- the composition of morphisms is defined componentwise, i.e. for any two morphisms $\mu 1 = \langle \mathbf{G1}, \eta 1 \rangle$ and $\mu 2 = \langle \mathbf{G2}, \eta 2 \rangle$ such that the target of $\mu 1$ coincides with the source of $\mu 2$, their composition is $\mu 1;\mu 2 =_{\text{def}} \langle \mathbf{G1};\mathbf{G2}, \eta \rangle$, where $\eta_A = \eta 2_{\mathbf{G1}(A)};\eta 1_A$ for any $A \in |\mathbf{I1}|$.

Prove that if the category **T** is cocomplete then the category **Into(T)** is complete. (This result was mentioned without proof in [Gog 71].)

References

[Gog 71] J.A. Goguen. Mathematical representation of hierarchically organized systems. In *Global System Dynamics* (E. Attinger, ed.), pp. 112-128. S. Karger (1971).

Algebras and categories (1987)

Definition: An *indexed category* **C** over a category **Ind** (the *index category*) is a functor from $\mathbf{Ind}^{\mathrm{op}}$ to **Cat**. Thus, for each index $i \in |\mathbf{Ind}|$ there is a category $\mathbf{C}(i)$ and for every index morphism σ: $i \to j$ in **Ind** there is a functor $\mathbf{C}(\sigma)$: $\mathbf{C}(j) \to \mathbf{C}(i)$.

Examples of indexed categories are: many-sorted sets (the index category is **Set** and the functor takes S to $\mathbf{Set}^S$ and f: $S \to S'$ to the forgetful functor from $\mathbf{Set}^{S'}$ to $\mathbf{Set}^S$) and many-sorted algebras (the index category is **Sign** and the functor takes $\Sigma \in |\mathbf{Sign}|$ to $\mathbf{Alg}(\Sigma)$ and σ: $\Sigma \to \Sigma'$ to the functor $\mathbf{U}_\sigma$: $\mathbf{Alg}(\Sigma') \to \mathbf{Alg}(\Sigma)$).

Definition: For any indexed category **C**: $\mathbf{Ind}^{\mathrm{op}} \to \mathbf{Cat}$, *Flat*(**C**) is a category with:

Objects: pairs $\langle i, a \rangle$ where $i \in |\mathbf{Ind}|$ and $a \in \mathbf{C}(i)$

Morphisms from $\langle i, a \rangle$ to $\langle j, b \rangle$: pairs $\langle \sigma, f \rangle$ where σ: $i \to j$ (in **Ind**) and f: $a \to \mathbf{C}(\sigma)(b)$ (in $\mathbf{C}(i)$).

Composition: if $\langle \sigma, f \rangle$: $\langle i, a \rangle \to \langle j, b \rangle$ and $\langle \rho, g \rangle$: $\langle j, b \rangle \to \langle k, c \rangle$ then $\langle \sigma, f \rangle;\langle \rho, g \rangle = \langle \sigma;\rho, f;\mathbf{C}(\sigma)(g) \rangle$: $\langle i, a \rangle \to \langle k, c \rangle$.

a) Let **C**: $\mathbf{Ind}^{\mathrm{op}} \to \mathbf{Cat}$ be an indexed category such that:

1. **Ind** is complete;
2. for all indices $i \in |\mathbf{Ind}|$, $\mathbf{C}(i)$ is complete; and

3. for all index morphisms σ: $i \to j$, the functor $\mathbf{C}(\sigma)$: $\mathbf{C}(j) \to \mathbf{C}(i)$ is continuous (i.e. translates limits in $\mathbf{C}(j)$ to limits in $\mathbf{C}(i)$).

Prove that the category $Flat(\mathbf{C})$ is complete. (Hint: it is sufficient to prove that $Flat(\mathbf{C})$ has all products and equalisers.)

Definition: An indexed category $\mathbf{C}$: $\mathbf{Ind}^{\mathrm{op}} \to \mathbf{Cat}$ is *locally reversible* if for every index morphism σ: $i \to j$ in $\mathbf{Ind}$ the translation functor $\mathbf{C}(\sigma)$: $\mathbf{C}(j) \to \mathbf{C}(i)$ has a left adjoint $\mathbf{F}(\sigma)$: $\mathbf{C}(i) \to \mathbf{C}(j)$ (let η^{σ}: $id_{\mathbf{C}(i)} \to \mathbf{F}(\sigma);\mathbf{C}(\sigma)$ denote the unit).

b) Let $\mathbf{C}$: $\mathbf{Ind}^{\mathrm{op}} \to \mathbf{Cat}$ be an indexed category such that:

1. $\mathbf{Ind}$ is cocomplete;
2. for all indices $i \in |\mathbf{Ind}|$, $\mathbf{C}(i)$ is cocomplete; and
3. $\mathbf{C}$ is locally reversible.

Prove that the category $Flat(\mathbf{C})$ is cocomplete. (Hint: it is sufficient to prove that $Flat(\mathbf{C})$ has all coproducts and coequalisers; coproducts are the easier of the two.)

Some of the following definitions and facts may be useful (you may use the facts without proving them):

Fact: Let $\mathbf{C}$: $\mathbf{Ind}^{\mathrm{op}} \to \mathbf{Cat}$ be a locally reversible indexed category. For any index morphisms σ: $i \to j$ and ρ: $j \to k$ in $\mathbf{Ind}$, there is a natural isomorphism $i_{\sigma,\rho}$: $\mathbf{F}(\sigma;\rho) \to \mathbf{F}(\sigma);\mathbf{F}(\rho)$.

Definition: Let $\mathbf{C}$: $\mathbf{Ind}^{\mathrm{op}} \to \mathbf{Cat}$ be a locally reversible indexed category. For any index morphism ρ: $i \to j$, we can "lift along ρ" any morphism $\langle \sigma, g \rangle$: $\langle k, a \rangle \to \langle i, b \rangle$ in $Flat(\mathbf{C})$ to a morphism in C(j):

$$L_{\rho}(\langle \sigma, g \rangle) = i_{\sigma,\rho}(a);\mathbf{F}(\rho)(g^{\#})\text{: } \mathbf{F}(\sigma;\rho)(a) \to \mathbf{F}(\rho)(b)$$

Fact: Under the notation and assumptions of the above definition, for any index morphism θ: $j \to m$ in $\mathbf{Ind}$ and morphism $\langle \rho;\theta, f \rangle$: $\langle i, b \rangle \to \langle m, c \rangle$ in $Flat(\mathbf{C})$, $f^{\#}$: $\mathbf{F}(\sigma)(b) \to \mathbf{C}(\theta)(c)$ is a morphism in $\mathbf{C}(j)$ such that

$$\langle \sigma;\rho, \eta^{\sigma;\rho}(a) \rangle;\langle \theta, L_{\rho}(\langle \sigma, g \rangle);f^{\#} \rangle = \langle \sigma, g \rangle;\langle \rho;\theta, f \rangle\text{: } \langle k, a \rangle \to \langle m, c \rangle$$

in $Flat(\mathbf{C})$.

Corollary: Under the notation and assumptions of the previous definition,

$$\eta^{\sigma;\rho}(a);\mathbf{C}(L_{\rho}(\langle \sigma, g \rangle)) = g;\mathbf{C}(\eta^{\rho}(b))$$

Algebras and categories (1986)

Let $\mathbf{K}$ be an arbitrary category.

Definition: A *factorization system* for $\mathbf{K}$ is a pair $\langle E, M \rangle$ such that:

(a) E is a class of epimorphisms in $\mathbf{K}$ and M is a class of monomorphisms in $\mathbf{K}$

(b) both E and M are closed under composition and contain all isomorphisms in $\mathbf{K}$

(c) every morphism in $\mathbf{K}$ has an $\langle E, M\rangle$-factorization, i.e. for any morphism f there are $e \in E$ and $m \in M$ such that $f = e; m$

(d) $\langle E, M\rangle$-factorizations are unique up to isomorphisms, i.e. for any $e1, e2 \in E$ and $m1, m2 \in M$, if $e1; m1 = e2; m2$ then there is an isomorphism i such that $e1; i = e2$ and $i; m2 = m1$.

Let $\langle E, M\rangle$ be a factorization system for $\mathbf{K}$.
Prove the following theorems:

1. For any morphisms $f1, f2, e, m$ in $\mathbf{K}$, if $f1; m = e; f2, e \in E$ and $m \in M$, then there is a morphism g such that $e; g = f1$ and $g; m = f2$.

2. If $e \in E$ and f is a morphism such that $e; f \in M$, then e is an isomorphism. If $m \in M$ and g is a morphism such that $g; m \in E$, then m is an isomorphism.

Definition: For any $A \in |\mathbf{K}|$, a *sub-object* of A is any $B \in |\mathbf{K}|$ together with a morphism $m: B \to A$ such that $m \in M$. A *quotient* of A is any $B \in |\mathbf{K}|$ together with a morphism $e: A \to B$ such that $e \in E$. An object $A \in |\mathbf{K}|$ is *reachable* if it has no proper sub-object, i.e. every morphism $m \in M$ with co-domain A is an isomorphism.

Assume that $\mathbf{K}$ has an initial object and all products (of sets of objects).
Definition: A class $K \in |\mathbf{K}|$ of objects is *closed under*:

(a) *products* if for any set $F \subseteq K$, the product of F belongs to K.

(b) *non-empty products* if for any non-empty set $F \subseteq K$, the product of F belongs to K.

(c) *sub-objects* if for any morphism $m \in M$, if the co-domain of m belongs to K then so does its domain.

(d) *quotients* if for any morphism $e \in E$, if the domain of e belongs to K then so does its co-domain.

Definition: A class $K \in |\mathbf{K}|$ is called a *variety* (resp. *quasi-variety*) if it is closed under quotients, sub-objects and products (resp. under sub-objects and non-empty products).

Prove the following theorems:

1. An object $A \in |\mathbf{K}|$ is reachable iff the unique morphism from the initial object to A is in E.

2. If $A \in |\mathbf{K}|$ is reachable then for every $B \in |\mathbf{K}|$ there is at most one morphism from A to B.

3. If $A, B \in |\mathbf{K}|$, B is reachable, and $f: A \to B$, then $f \in E$.

4. Every object $A \in |\mathbf{K}|$ has a unique (up to isomorphism) reachable sub-object.

5. For any object $A \in |\mathbf{K}|$, let $Ext(A)$ denote the class of objects in $\mathbf{K}$ such that $B \in Ext(A)$ iff there is a morphism from A to B. If $A \in |\mathbf{K}|$ is reachable then:

   (a) A is initial in $Ext(A)$

(b) $Ext(A)$ is a variety

6. Suppose $\mathbf{K}$ is E-co-well-powered, i.e. for every object $A \in |\mathbf{K}|$ there is a set of morphisms $E' \subseteq E$ with domain A such that for every $e \in E$ with domain A there is an $e' \in E'$ and isomorphism i such that $e = e'; i$. Then any non-empty quasi-variety in $\mathbf{K}$ has a reachable initial object.

Category Theory (1985)

This question consists of a number of loosely-connected sub-questions.

1. Let $\mathbf{K}$ be an arbitrary category. Prove that if $\mathbf{K}$ has coproducts (i.e. any set of objects in $\mathbf{K}$ has a coproduct) and coequalizers (i.e. any *pair* of morphisms in $\mathbf{K}$ with the same domain and codomain has a *coequalizer* then $\mathbf{K}$ is cocomplete (i.e. any diagram in $\mathbf{K}$ has a colimit). Is it enough to assume that any *pair* of objects has a coproduct? Give a proof or a counter-example.

2. Let $\mathbf{K}$ be an arbitrary category. Prove that if $\mathbf{K}$ has products (i.e. any set of objects $\mathbf{K}$ has a product) and equalizers (i.e. any pair of morphisms in $\mathbf{K}$ with the same domain and codomain has an equalizer) then $\mathbf{K}$ is complete (i.e. any diagram in $\mathbf{K}$ has a limit).

3. Let Σ be a many-sorted signature. As usual, $\mathbf{Alg}(\Sigma)$ is the category of Σ-algebras with Σ-homomorphisms as morphisms. Show that $\mathbf{Alg}(\Sigma)$ is complete.

4. Show that the category **Set** of sets with functions as morphisms is complete and cocomplete.

5. Consider the category **RSet** with sets as objects and binary relations as morphisms, i.e. for any sets A and B, $\mathbf{RSet}\ (A, B) = \{R \mid R \subseteq A \times B\}$. Composition is defined as usual: for $R \subseteq A \times B, S \subseteq B \times C, R; S = \{(a, c) \mid \exists\ b \in B.\ [(a, b) \in R\ \&\ (b, c) \in S]\}$. Is **RSet** cocomplete? Give a proof or a counter-example.

6. **Set** is a sub-category of **RSet**; let $\mathbf{I}$:**Set** $\rightarrow$ **RSet** be the inclusion functor, i.e. $\mathbf{I}$ is the identity map on objects (sets) and it maps any function $f: A \rightarrow B$ to its graph relation $\mathbf{I}(f) = \{(a, b) \mid f(a) = b\} \subseteq A \times B$. Show that $\mathbf{I}$ preserves colimits.

Algebras and categories (1984)

Preliminaries

By an ω-complete poset we mean an arbitrary partially ordered set $\langle X, \leq \rangle$ in which least upper bounds of all countable chains exist. A function between two ω-complete posets is ω-continuous if it preserves least upper bounds of all countable chains.

Let $\Sigma = \{\Sigma_n\}_{n \geq 0}$ be a (one-sorted) algebraic signature.

Define the category $\mathbf{Alg}_\omega(\Sigma)$ of ω-continuous Σ-algebras (with ω-complete posets as carriers and ω-continuous functions as operations) and ω-continuous Σ-homomorphisms (with the natural composition). In the following we identify any class of K of ω-continuous

Σ-algebras with the full sub-category of $\mathbf{Alg}_\omega(\Sigma)$ having objects K and assume that K is closed under isomorphism.

If X is a set (of variables), the algebra of Σ-terms with variables X is defined as usual. Moreover, for any term t with variables X, ω-continuous Σ-algebra A and valuation $v: X \to |A|$, the value of t in A under v, $t_A(v) \in |A|$, is defined as usual (state these definitions).

By a Σ-inequality we mean a triple $\langle X, t, t' \rangle$ written in the form $\forall X.t \leq t'$, where X is a set (of variables) and t, t' are Σ-terms with variables X.

An ω-continuous Σ-algebra A satisfies a Σ-inequality $\forall X.t \leq t'$ if and only if for any valuation $v: X \to |A|$,

$$t_A(v) \leq_A t'_A(v).$$

For any set of Σ-inequalities IE, let $Alg_\omega(\Sigma, IE)$ denote the class of all ω-continuous Σ-algebras that satisfy all Σ-inequalities in IE.

Let $\mathbf{U}_{\Sigma,IE}: Alg_\omega(\Sigma, IE) \to \mathbf{Poset}$ be the functor from $Alg_\omega(\Sigma, IE)$ to the category of all partially-ordered sets and monotone maps, mapping any ω-continuous Σ-algebra to its partially-ordered carrier and any ω-continuous Σ-homomorphism to its underlying map (show that this is in fact a functor).

Problem

Prove that for any (one-sorted) signature Σ and set of Σ-inequalities IE, the functor $\mathbf{U}_{\Sigma,IE}: Alg_\omega(\Sigma, IE) \to \mathbf{Poset}$ has a left adjoint.

Sub-problems, indicating one approach to a solution

1. Show that any set of ω-continuous Σ-algebras in $Alg_\omega(\Sigma, IE)$ has a product in $Alg_\omega(\Sigma, IE)$.

2. Define a notion of ω-continuous Σ-subalgebra in such a way that whenever $A \in Alg_\omega(\Sigma, IE)$, any ω-continuous Σ-subalgebra of A belongs to $Alg_\omega(\Sigma, IE)$ as well.

3. Prove that for any ω-continuous Σ-algebra A and set $X \subseteq |A|$ there is at least ω-continuous Σ-subalgebra $[X]_A$ of A which contains X. Show that whenever X generates A, i.e. $[X]_A = A$, any two ω-continuous Σ-homomorphisms $h1, h2: A \to B$ (B is an ω-continuous Σ-algebra) are equal if they are equal on X.

4. Prove that any class K of ω-continuous Σ-algebras closed under isomorphism, products and ω-continuous Σ-subalgebras has an initial algebra.
   Hint: If the product of K existed, its subalgebra generated by the empty set would be initial in K. To justify a similar construction which gives an initial algebra in K, prove that the cardinality of ω-continuous Σ-algebras generated by the empty set is bounded (and so there is a *set* G of ω-continuous Σ-algebras generated by the empty set such that any ω-continuous Σ-algebra generated by the empty set is isomorphic to an element of G).

Using (4), prove that for any set of Σ-inequalities IE and any partially ordered set $\langle X \leq \rangle$, there is an algebra in $Alg_\omega(\Sigma, IE)$ free over $\langle X \leq \rangle$.
Hint: Consider the signature $\Sigma(X)$, which is the extension of Σ by a constant for each element of X.

Algebras and categories (1982)

The paper "An algebraic description of programs with assertions, verification and simulation", *SIGPLAN Notices, 7, 1* by Burstall was written in 1971. Rewrite it more clearly using more standard categorical terminology and definitions. Perhaps you can extend the results e.g. to less restrictive notions of simulation between programs.

You need not rewrite the explanatory material nor necessarily redo the whole paper, but please make your answer intelligible.

Algebras and categories (1981)

Show that pushouts preserve epimorphisms, that is if $f: a \to b$ and $g: a \to c$ have pushout p with morphisms $g1: b \to p$ and $f1: c \to p$ then if f is epic so is $f1$. Show further that pushout preserves regular epimorphisms, where $f: a \to b$ is a regular epimorphism if it is the coequaliser of morphisms $h1, h2: z \to a$ for some z.

Category theory (1980)

1. Show that if a category has an initial object, binary coproducts and coequalisers then it has finite colimits. (Please don't look up the proof in a book).

2. Write briefly (say half a page) on the applications of colimit constructions in computation theory.

Category Theory (1979)

Write an account of some application of Category Theory in Computer Science. Give references.

9 Denotational semantics

Denotational Semantics (1988)

Read the first two pages of [MC], a substantial portion of which describes a model of networks of processes which appeals to "internal" and "external" traces.

Your task is to make this model precise by developing it into a denotational semantics of networks. To do this you will need to specify an abstract syntax of networks, and you will need to describe the domains of traces. Finally, you should comment on when two networks will have the same meaning, according to your semantics.

References

[MC] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Trans. on Software Engineering*, SE-7, 4 (1981).

Denotational Semantics (1987)

Read [THM] and answer the following questions.

(a) Specify an abstract syntax for a PROG language.

(b) Comment on the claims made on pages 485-487 about the inadequacies of standard store models. (Don't spend much time on this.)

(c) Give a denotational semantics of your PROG using store models and support (pp 487-488).

(d) Sketch proofs that the local storage discipline makes equivalent the examples given on page 489.

References

[THM] B. Trakhtenbrot, J. Halpern and A. Meyer. From denotational to operational and axiomatic semantics for Algol-like languages: an overview. *Proc. Logics of Programming Workshop*, Carnegie-Mellon, Springer LNCS Vol 164 (1984).

Denotational Semantics (1986)

(a) Give a brief description and comparison of the proof methods known as structural induction and computational induction.

(b) Read [Rus] carefully. Explain why the proof of the theorem in section 5 is (seriously) incorrect.

(c) Give a proof of the following lemma by computational induction on sem':

$$\text{for all } p',\ sem'(p') \sqsubseteq sem(\text{compiler}(p'))(\text{firstlabel}(\text{compiler}(p')))$$

(If several cases of a case analysis are essentially similar, you need only discuss one of them in detail.)

References

[Rus] B. Russell. Implementation correctness involving a language with goto statements. *SIAM J Computing* 6, 3 (1977).

Denotational Semantics (1985)

Consider a variant of the usual simple imperative language, allowing exceptions to be raised and handled. The primitive phrase classes are:

$$\begin{array}{ll} a \in \text{ACom} & \text{(Atomic Commands)} \\ b \in \text{BExp} & \text{(Boolean Expressions)} \\ l \in \text{Lab} & \text{(Labels)} \end{array}$$

and the syntax of commands $c \in \text{Com}$ is given by

$$\begin{array}{l} c ::= a \mid \textbf{skip} \mid c;c \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid \textbf{loop } c \\ \quad \mid \textbf{exception } l \textbf{ is } c \textbf{ in } c \mid \textbf{raise } l \end{array}$$

The command "**loop** c" repeats c until an exception is raised; thus

$$\textbf{while } b \textbf{ do } c$$

could be defined as

$$\textbf{exception } l \textbf{ is skip in loop (if } b \textbf{ then } c \textbf{ else raise } l\textbf{)}$$

where l is not free in c. We assume truth values $T = \{\text{true, false}\}$, a set S of states, and primitive semantic functions

$$\begin{array}{l} \mathcal{A}\text{: ACom} \to S \to S \\ \mathcal{B}\text{: BExp} \to S \to T \end{array}$$

We now define cpos as follows

$$\begin{array}{ll} r \in R = S_\perp + (\text{Lab} \times S)_\perp & \text{(results)} \\ \gamma \in C = S_\perp \to_\perp S_\perp & \text{(continuations)} \\ \rho \in \text{LEnv} = \text{Lab}_\perp \to_\perp C & \text{(Label environments)} \end{array}$$

[**Note:** You may find it convenient to use the injections $first\text{: } S \to R$ and $second\text{: Lab} \times S \to R$, and to use the syntax

$$\textbf{cases } e\text{: } first(s) \Rightarrow \ldots, second(l,s) \Rightarrow \ldots$$

for case analysis of the value of an arbitrary result-valued expression e (where the variables s, or l and s, become bound to the value components).]

(a) Give a direct semantics and a continuation semantics for Com:

$$\begin{array}{l} \mathcal{D}\text{: Com} \to S_\perp \to_\perp R \\ \mathcal{C}\text{: Com} \to \text{LEnv} \to C \to C \end{array}$$

(b) Write down an equation

$$\mathcal{C}[\![c]\!] = \ldots$$

which expresses the continuation semantics of c in terms of its direct semantics $\mathcal{D}[\![c]\!]$, and prove your equation from your definitions.

(c) With the help of (b) show that, whenever c contains no free labels,

$$\mathcal{D}[\![c]\!] = \mathcal{C}[\![c]\!] \perp (\lambda_{\perp} s.s)$$

(d) A context C is defined to be a "command with a hole in it", so that if c is a command then so is $C[c]$. You may assume that $\mathcal{D}[\![c_1]\!] = \mathcal{D}[\![c_2]\!]$ implies $\mathcal{D}[\![C[c_1]]\!] = \mathcal{D}[\![C[c_2]]\!]$, and likewise for $\mathcal{C}$. Prove that the following defines an equivalence relation $\equiv$ over Com:

$c_1 \equiv c_2$ iff, for every context $C[\ ]$ such that $C[c_1]$ and $C[c_2]$ have no free labels, $\mathcal{D}[\![C[c_1]]\!] = \mathcal{D}[\![C[c_2]]\!]$

(e) Assuming that for any state $s \in S'$ there is some $a \in$ ACom such that $\mathcal{A}[\![a]\!]s \neq s$, prove that

$$c_1 \equiv c_2 \text{ iff } \mathcal{D}[\![c_1]\!] = \mathcal{D}[\![c_2]\!]$$

(f) Suppose now that Com is extended by adding non-recursive parameterless procedures:

$$c ::= \ldots \mid \textbf{procedure } p = c \textbf{ in } c \mid \textbf{call } p$$

where p ranges over identifiers. Without giving full formal details, discuss (in not more than 400 words) how procedures might interact with exceptions, and how the semantic functions $\mathcal{D}$ and $\mathcal{C}$ might be modified to deal with this interaction correctly.

Denotational semantics (1984)

Three principles of programming language design are defined in [Ten]. See Chapter 8 of [Sch]. The principles are:

abstraction: every syntactic class of a language is allowed an abstract (i.e. procedure) definition mechanism.

parameterisation: an abstract may be parameterised on any syntactic class which may occur as a subpart of the abstract's body.

correspondence: every parameter transmission method corresponds to a binding method used in abstract definition.

qualification: every syntactic class is allowed a block structure construct for holding definitions.

Given the following syntax of an ML-like applicative language:

E: Expression
V: Varient
T: Typestruct
I: Identifier

$E::=$	$\mathbf{0}$	The number zero.
	$\mid \mathbf{succ}\ E$	The successor operation on numbers.
	$\mid I$	See the function construct below.
	$\mid E_1, E_2$	A pair of values.
	$\mid \lambda V{:}T.E$	A function — an argument with data type T is to be bound to V and used to evaluate E.
	$\mid E_1 E_2$	Function E_1 is applied to argument E_2.
	$\mid (E)$	Use parentheses as you please.
$V::=$	I	A binding variable can be a single identifier
	$\mid V_1, V_2$	or a pair of things — see Chapter 3 of [GMW] for details.
$T::=$	$\mathbf{nat}$	Numbers have type **nat**.
	$\mid T_1 \times T_2$	Pairs have this type.
	$\mid T_1 \rightarrow T_2$	Functions have this type.

answer the following questions:

(a) Provide a denotational semantics for the language which is faithful to the informal descriptions given next to the syntax definitions. Provide some informal documentation of your definition and note any design decisions you have made.

(b) Apply the qualification and abstraction principles to all syntax classes to cause the addition of **let** ... **in** ... blocks. You will need to

- augment the syntax definition
- add new semantic algebras and alter existing ones
- add new semantic clauses to the valuation functions.

When updating the definition, note any interference that the new constructs bring to the existing clauses of the semantic definition. Comment on trouble spots in the augmented language.

(c) Apply the parameterisation and correspondence principles to allow Expression abstracts to accept parameters. For simplicity, you may deal with Expression abstracts which take exactly one parameter. As in part (b), augment the syntax and semantics and note any trouble spots which arise.

(d) Based on your experiences in (a) – (c), prune the language down to a subset which you feel has the best pragmatics. Give the abstract syntax of the language and justify your selections and omissions.

References

[GMW] M.J. Gordon, A.J.R. Milner and C.P. Wadsworth. *Edinburgh LCF.* Springer LNCS, Vol. 78 (1979).

[Sch] D. Schmidt. *Denotational Semantics.* Allyn and Bacon (1986).

[Ten] R. Tennent. *Principles of Programming Languages.* Prentice Hall (1981).

(a) Below is given the abstract syntax of an ML subset. The subset is typed (but non-polymorphic) and contains the **abstype** mechanism. Based on your experiences with ML, define a denotational semantics in the spirit of ML for the language.

(b) A denotational definition should provide a guide to the implementation of the programming language. In particular, the semantic domains used in the definition should be mapped to built-in data types (e.g. integers) or to system level data structures (e.g. symbol tables). Based on the results of part (a), *briefly* and *informally* describe the built-in data types that an abstract machine for executing programs should have and the data structures that a compiler for this subset of ML to the abstract machine should use.

Syntax domains

P: Program
E: Expression
D: Declaration
B: Binding
T: Datatypename
I: Identifier
Z: Integernumeral

Syntax (annotated with informal semantics)

$P::=$	E	A program is just a single expression.
$D::=$	**let** B	Declare an ordinary binding.
	| **abstype** $I = T$ **with** B **end**	Declare a user-defined abstract type. An abstract type name I is introduced, and T, a type expression, is its representation type. The equation $I = T$ may be recursive! The operations upon I-typed objects are listed as bindings B.
$B::=$	$I = E$	Identifier I is bound to expression value E and has the same type as E. This is a non-recursive binding.
	| **rec** $I = E$	As before, but any references to I in E are recursive.
	| B_1 **and** B_2	A list of bindings.

$E::=$	D **in** E	Expressions may have local declarations; use static scoping.
	$\mid I$	I must have been declared earlier in an enclosing expression.
	$\mid E_1\ E_2$	Function E_1 is applied to argument E_2. Types must match in the obvious way.
	$\mid \lambda I{:}T.E$	A function abstraction which expects an argument of type T. The argument is bound to I for evaluation of E. Use call-by-value parameter evaluation and static scoping as in ML.
	$\mid E_1 + E_2$	Integer expressions E_1 and E_2 are added. Types must be checked as usual.
	$\mid E_1 = E_2$	Comparison for equality of arithmetic expressions. A truth value results.
	$\mid$ not E	Negation on truth-valued expression E.
	$\mid Z$	An integer constant.
	$\mid \langle E_1, E_2 \rangle$	A pair of expressions.
	$\mid$ fst E	Selects the first object from a pair. E must evaluate to an expression pair.
	$\mid$ snd E	Selects the second object from a pair.
	$\mid$ **if** E_1 **then** E_2 **else** E_3	The ML conditional: E_1 must be a truth value, and E_2 and E_3 must have the same type. The conditional evaluates in the usual way.
	$\mid$ abs E	Used only within the body B of '**abstype** $I = T$ **with** B **end**', abs coerces T-typed expression E to type I.
	$\mid$ rep E	Used only within the body B of '**abstype** $I = T$ **with** B **end**', rep coerces I-typed expression E to type T.
$T::=$	int	The data types include the integers,
	$\mid$ bool	the truth values,
	$\mid I$	user defined type names,
	$\mid (T_1 \rightarrow T_2)$	function spaces, and
	$\mid (T_1 \times T_2)$	product (pair) spaces.

Definitions for I and Z are omitted.

Denotational Semantics (1982)

Consider the following variation on the usual simple imperative language with facilities for exception raising and handling. First ACom, BExp and Lab are categories of Atomic Commands, Boolean Expressions and Labels ranged over by a, b and l, respectively. Now the category of commands is Com ranged over by c and given by:

$c::= a \mid$ **skip** $\mid c; c \mid$ **if** b **then** c **else** c **fi** $\mid$ **loop** c **cycle** $\mid$ **exception** l **is** c **in** $c \mid$ **raise** l

Here **loop** c **cycle** continually executes c until an exception is raised. For example one can get the effect of **while** b **do** c by

exception l **is skip in**
loop if b **then** c **else raise** l **fi**
cycle

where l is not free in c.

For the semantics we assume given a set S of states and functions $\mathcal{A}$: ACom $\to S \to S$ and $\mathcal{B}$: BExp $\to S \to T$.

(a) Give both a direct semantics

$$\mathcal{D}\text{: Com} \to (S_\perp \to_\perp (S_\perp + \text{Lab}_\perp \otimes S_\perp))$$

and a continuation semantics

$$\mathcal{C}\text{: Com} \to \text{LEnv} \to C \to C$$

where the cpo of label environments is $\text{LEnv} = \text{Lab}_\perp \to_\perp C$ where $C = (S_\perp \to_\perp S_\perp)$.

(b) Prove that the continuation semantics can be obtained from the direct semantics by the formula

$$\begin{aligned} \mathcal{C}[\![c]\!]\rho\gamma\sigma = \quad & \textbf{cases } \mathcal{D}[\![c]\!]\sigma \\ & \textbf{first } \sigma'.\gamma\sigma' \\ & \textbf{second } l', \sigma'.\rho l'\sigma' \end{aligned}$$

Use this to show that if c contains no free labels then

$$\mathcal{D}[\![c]\!]\sigma = \mathcal{C}[\![c]\!] \perp (\lambda_\perp \sigma'.\sigma')\sigma$$

(c) Define a relation $\equiv$ between commands by $c_1 \equiv c_2$ iff for all contexts $C[\ ]$ such that $C[c_1]$ and $C[c_2]$ have no free labels, $\mathcal{D}[\![C[c_1]]\!] = \mathcal{D}[\![C[c_2]]\!]$. Prove this is an equivalence relation.

Here a context $C[\ ]$ is a "command with one hole in it" so that if c is a command so is $C[c]$. For example if $C[\ ]$ is **exception** l **is** c_1 **in** $[\ ]$ then $C[c_2]$ is **exception** l **is** c_1 **in** c_2. Note that for any c_1, c_2 if $\mathcal{D}[\![c_1]\!] = \mathcal{D}[\![c_2]\!]$ then $\mathcal{D}[\![C[c_1]]\!] = \mathcal{D}[\![C[c_2]]\!]$ and similarly for $\mathcal{C}$.

Assume from here on that for any state σ there is an a with $\mathcal{A}[\![a]\!]\sigma \neq \sigma$. Show that for all commands c_1 and c_2

$$c_1 \equiv c_2 \text{ iff } \mathcal{D}[\![c_1]\!] = \mathcal{D}[\![c_2]\!]$$

Now use this and part (b) to show that for all commands c_1 and c_2

$$\mathcal{D}[\![c_1]\!] = \mathcal{D}[\![c_2]\!] \text{ iff } \mathcal{C}[\![c_1]\!] = \mathcal{C}[\![c_2]\!]$$

(d) Define an operational semantics using a *transition relation* $\rightarrow \subseteq \Gamma \times \Gamma$ between a set Γ of *configurations*. As configurations take the set

$$\Gamma = (\mathrm{Com} \times S) \cup S \cup (\mathrm{Lab} \times S)$$

and as *terminal* = *final* configurations take

$$T = S \cup (Lab \times S)$$

Define $\mathcal{O}\colon \mathrm{Com} \rightarrow (S_\perp \rightarrow_\perp (S_\perp + \mathrm{Lab}_\perp \otimes S_\perp))$ by

$$\mathcal{O}[\![c]\!](\sigma) = \begin{cases} \perp & \text{if } \sigma = \perp \text{ or if } \langle c, \sigma\rangle \text{ diverges} \\ \sigma' & \text{if } \langle c, \sigma\rangle \rightarrow^* \sigma' \\ l' \otimes \sigma' & \text{if } \langle c, \sigma\rangle \rightarrow^* \langle l', \sigma'\rangle \end{cases}$$

Prove that $\mathcal{D} = \mathcal{O}$. You need not give every detail but you should demonstrate understanding of the essential ideas of the proof.

Denotational Semantics (1981, 1979)

Describe how to add coroutines to *SMALL* (see [Gor]). Assume a coroutine named I is declared by **coroutine** $I(I_1); C$. To run I with actual parameter E one evaluates the expression **run** $I(E)$. To temporarily leave a coroutine one executes the command **leavewith** E; this passes E's value to the context which ran (or resumed) the coroutine. A temporarily-left coroutine can be resumed by evaluating the expression **resume** I; this "starts up" I at the beginning of the command following the **leavewith** E in I. To permanently leave a coroutine one evaluates the expression **quitwith** E; subsequent attempts to resume it cause errors. For example:

```
begin coroutine count(n);
                while n < 3 do
                  (leavewith n;
                    n:= n + 1);
                quitwith 3;
      output run count(0);      -  outputs 0 and leaves
      output resume count;      -  outputs 1 and leaves
      output resume count;      -  outputs 2 and leaves
      output resume count;      -  outputs 3 and quits
      output resume count;      -  causes an error
 end
```

In your description of the semantics of coroutines you should include semantic clauses for the constructs **coroutine** $I(I_1); C$, **run** $I(E)$, **resume** I, **leavewith** E and **quitwith** E.

References

[Gor] M.J.C. Gordon. *The Denotational Description of Programming Languages.* Springer (1979).

Denotational Semantics (1980)

This question concerns a simple compiler correctness proof.

(a) Consider the simple imperative language with the following syntax:
ACom - the *atomic commands*, a given set ranged over by a
BExp - the *Boolean expressions*, a given set ranged over by b
Com - the *commands*, ranged over by c, and given by:

$$c ::= a \mid \textbf{skip} \mid (c; c) \mid (\textbf{if } b \textbf{ then } c \textbf{ else } c) \mid (\textbf{until } b \textbf{ do } c)$$

For the denotational semantics we postulate a flat cpo $S_\perp$, of states, ranged over by σ and denotational functions,

$$\mathcal{A}\colon \text{ACom} \to (S_\perp \to_\perp S_\perp)$$
$$\mathcal{B}\colon \text{BExp} \to (S_\perp \to_\perp T_\perp)$$

where T is the set of truth values. Write a standard denotational semantics, defining

$$\mathcal{C}\colon \text{Com} \to (S_\perp \to_\perp S_\perp)$$

(b) Next, consider the "machine code" language with the following syntax:
Lab — an infinite set of *labels* ranged over by L
MProg — a set of *machine code programs* ranged over by m and given by

$$m ::= a \mid \textbf{skip} \mid \textbf{jump } L \mid \textbf{jump } L \textbf{ if } b$$
$$\mid (L_1\colon m_1 \ldots L_k\colon m_k) \qquad \text{(if the } L_i \text{ are all different)}$$

We consider a direct semantics $\mathcal{M}\colon \text{MProg} \to (S_\perp + \text{Lab}_\perp \otimes S_\perp)$ but you can define and use a continuation semantics instead if you prefer (alter the statement of compiler correctness if you do). The idea is that if $\mathcal{M}[\![m]\!](\sigma)$ is $in_0(\sigma')$ that means m exits normally in state σ' when started in state σ; if instead it is $in_1(L \otimes \sigma')$ that means m exits with a jump to L in state σ', when started in σ. Here are some clauses in the definition of m — you supply the other two:

$$\mathcal{M}[\![\textbf{skip}]\!] = in_0$$
$$\mathcal{M}[\![\textbf{jump}]\!] = \lambda\sigma.in_1(L \otimes \sigma)$$
$$\mathcal{M}[\![L_1\colon m_1 \ldots L_k\colon m_k]\!] = \rho[\![L_1]\!] \textit{ where}$$
$$\rho = \mu\rho.\lambda L. \quad \textbf{if } L = L_1 \textbf{ then } [\rho[\![L_2]\!], \lambda_\perp L, \sigma.\rho[\![L]\!](\sigma)] \circ \mathcal{M}[\![M_1]\!]$$
$$\textbf{elseif } L = L_{k-1} \textbf{ then } [\rho[\![L_k]\!], \lambda_\perp L, \sigma.\rho[\![L]\!](\sigma)] \circ \mathcal{M}[\![M_{k-1}]\!]$$
$$\textbf{elseif } L = L_k \textbf{ then } [in_0, \lambda_\perp L, \sigma.\rho[\![L]\!](\sigma)] \circ \mathcal{M}[\![M_k]\!]$$
$$\textbf{else } \lambda\sigma.L \otimes \sigma$$

(c) Here are two clauses in the definition of a "compiler" $Comp\colon \text{Com} \to \text{MProg}$ — you supply the missing clauses:

$$Comp[\![a]\!] = a$$
$$Comp[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!] =$$
L1: **jump** L4 **if** b
L2: $Comp(c_2)$
L3: **jump** L5
L4: $Comp(c_1)$
L5: **skip**

Prove the compiler correct in the sense that for all commands c,

$$in_0 \circ \mathcal{C}[\![c]\!] = \mathcal{M}[\![Comp[\![c]\!]]\!]$$

[**Advice:** Proceed by Structural Induction. It should be a straightforward calculation up to the **until** case when you might find it more convenient to prove two inequalities, using Fixpoint Induction or Computation Induction as appropriate.]

Denotational Semantics (1980)

Consider the following applied untyped λ-calculus.

- I ranges over *identifiers*
- B ranges over *basic expressions*
- F_1 ranges over *unary function symbols*
- F_2 ranges over *binary function symbols*
- E ranges over *expressions* given by:

$$E::= I \mid B \mid F_1 \mid F_2 \mid \lambda I.E \mid E(E)$$

Note that $\lambda I.E$ is a procedure, $E_0(E_1)$ is an application and procedures can be passed to procedures as parameters. There are two possible kinds of operational semantics:

Call-by-Value: Here parameters are evaluated when their procedures are called so that, for example, E_1 should be evaluated before applying E_0 to it in the expression $E_0(E_1)$.

Call-by-Name: Here one uses the textual replacement rule of Algol-60 so that actual para[meters are only evaluated when required in the body of the procedure.

(a) For each case outline a denotational semantics which properly reflects the intended behaviour. Explain your choice of semantical domains and why you chose direct or continuation semantics. You should assume a domain, V, of basic values and denotation functions $\mathcal{B}$, $\mathcal{F}_1$, $\mathcal{F}_2$, so that $\mathcal{B}[\![B]\!]$ is in V and $\mathcal{F}_i[\![F_i]\!]$ is in $V^i \to V$ for $i = 1, 2$.

(b) Generally, recursion is handled using the paradoxical combinator

$$Y_\lambda = \lambda F.(\lambda I.F(I(I)))(\lambda I.F(I(I)))$$

Show that this is useless in the Call-by-Value case and explain what this means in terms of the denotation of Y_λ. Show that an adequate replacement for Y_λ is provided by the combinator $Z = \lambda F.M(M)$, where M is $\lambda X.F(\lambda Z.X(X)(Z))$.

(c) Assume Y is the least fixed-point functional under the call-by-name semantics, show that $Y_\lambda(\lambda I.\lambda F.F(I(F)))$ is too.

Denotational Semantics (1979)

Consider the following imperative language:

Syntactic Sets

1. $b \in$ Bexp — The Boolean expressions
2. $a \in$ Act — The primitive actions
3. $L \in$ Lab — The set of labels y
4. $s \in$ Stat — The statements given by:

$$S ::= a \mid \textbf{dummy} \mid (s; s) \mid (\textbf{if } b \textbf{ then } s \textbf{ else } s) \mid (\textbf{while } b \textbf{ do } s) \mid (\text{L:s}) \mid \textbf{goto L}$$

Here "**goto** L" is executed by going to the statement labelled by the smallest textually enclosing L.

Semantic Domains

1. $t \in T_\perp$ — The truth values
2. $\sigma \in S_\perp$ — The states
3. $o \in O$ — The output domain
4. $\gamma \in C = S_\perp \to_\perp O$ — The continuations
5. $r \in R = S_\perp + (\text{Lab}_\perp \otimes S_\perp)$ — The results
6. $\theta \in DT = S_\perp \to_\perp R$ — The direct state transformations
7. $t \in CT = (C \times (\text{Lab}_\perp \to_\perp C)) \to_\perp C$ — The continuation state transformations

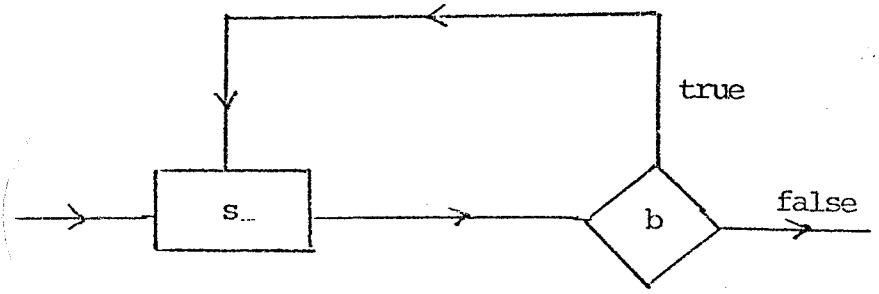
Denotations

1. $\mathcal{B}$: Bexp $\to (S_\perp \to_\perp T_\perp)$
2. $\mathcal{A}$: Act $\to (S_\perp \to_\perp S_\perp)$
3. $\mathcal{D}$: Stat $\to DT$ — see below
4. $\mathcal{C}$: Stat $\to CT$ — see below

Define $\mathcal{D}$ and $\mathcal{C}$. Define $\varphi: DT \to CT$ to show that $\mathcal{C} = \varphi \circ \mathcal{D}$. Show that if $O = R$ then $\mathcal{D}[\![S]\!] = \mathcal{C}[\![S]\!](id, \rho)$ for a suitable ρ.

Denotational Semantics (1978)

1. The language FLOW [Wad] is to be extended by an additional statement construct "**repeat** s **while** b", whose meaning is given informally by the following flowchart:



   (a) Give a semantic equation for this new construct analogous to that for **while**-statements (i.e. using the fixed point function fix), and indicate briefly (4 or 5 lines) how you arrived at your equation.

   (b) Prove by a fixed point method that "**repeat** s **while** b" is semantically equivalent to "s; **while** b **do** s", i.e. that

$$\mathcal{C}[\![\textbf{repeat } s \textbf{ while } b]\!] = \mathcal{C}[\![s; \textbf{while } b \textbf{ do } s]\!]$$

   (Note: No marks for defining the **repeat**-construct this way!)

2. The language L [Gor] is to be extended by a generalisation of the **valof/return** mechanism which allows these constructs to be tagged with identifiers indicating which **valof**-expression is to receive the value passed by a **return**-statement. Assume the additional syntax:

valof $I: S$	(an expression)
return E **to** I	(a statement)

   The meaning of "**valof** $I: S$" is: bind I to a new kind of value called a "tag value" associated with this **valof**-expression, then execute S until a **return**-statement is encountered. The meaning of "**return** E **to** I" is: evaluate E then return its value as that of the **valof**-expression whose tag value is bound to I.

   Additionally, "**valof** $I: S$" is to be a local binding construct, with S as the scope of I. Adopt also the design choice that the tag values bound to such identifiers may be used freely as expression values (in particular, they may be passed as parameters of procedures), except that they are not to be assignable, nor may they (or procedure values either) be returned by a **return**-statement.

   (a) Modify the semantic domains of [Gor], being sure to include in your list a suitable domain (call it Z) for tag values.

   (b) Give semantic equations for the two new constructs, and revise those for $\mathcal{E}[\![I]\!]$ and $\mathcal{S}[\![I := E]\!]$ as necessary.

   (c) Explain your domains and semantic equations (say, 3 or 4 lines on each), in particular noting any differences from the semantics of L and including comments on:

i. the choice of domain for tag values

ii. the composition of the domains D, E and V of denotable, expressible, and storable values

iii. error checks (run-time) in your semantic equations

iv. whether the constraints on tag values and procedure values are sufficient to enable this extended language to be implemented using a stack discipline for storage.

References

[Wad] C. Wadsworth, lecture notes.

[Gor] M. Gordon, lecture notes.

10 Domains

Domains (1988)

Notation. Dom_p is the O-category of Scott's domains and partial continuous functions, $PDom$ is the O-category of **pointed** domains (i.e. with a least element) and continuous functions, $PDom^E$ (Dom_p^E) is the corresponding ω-category of (p-)embeddings.

If A and B are domains, then $A \to B$ ($A \rightharpoonup B$) is the cpo of (partial) continuous functions. We write $in: A \lhd B$ ($in: A \lhd_p B$) to indicate that in is a (p-)embedding of A into B. If $F, F': A \to Dom_p^E$, then $In: F \lhd_p F'$ means that In is a natural transformation from F to F'. It is convenient to treat domains as ω-categories, in particular if $a \leq b$ then we write $a \to b$ for the only morphism from a to b.

1. Indexed Domain Theory ([CGW 87] may help):

1. If $A \in Dom_p^E$ and $F: A \to Dom_p^E$ is an ω-functor, let

   $\Sigma a: A.F(a)$ be $\{< a, b > | a \in A \wedge b \in F(a)\}$ with order
   $< a, b > \leq < a', b' > \stackrel{\Delta}{\Longleftrightarrow} a \leq a' \wedge F(a \to a')(b) \leq b'$

   prove that $\Sigma a: A.F(a)$ is a domain and that the projection $\pi: < a, b > \mapsto a$ is a continuous function from $\Sigma a: A.F(a)$ to A.

   If $in: A \lhd_p A'$ and $In: F \lhd_p F' \circ in$, prove that $\Sigma a: A.F(a) \lhd_p \Sigma a: A'.F'(a)$ via $(\Sigma in.In): < a, b > \mapsto < in(a), In_a(b) >$.

2. If both $F: A \to Dom_p^E$ and $G: (\Sigma a: A.F(a)) \to Dom_p^E$ are ω-functors, then extend $la: A.\Sigma b: F(a).G(a, b)$ to a functor from A to Dom_p^E (**not compulsary**: and prove that it is an ω-functor).

3. If $A \in PDom^E$ (i.e. A is pointed) and $F: A \to Dom_p^E$ is an ω-functor, let

   $\Pi a: A.F(a)$ be $\{f: A \to (\Sigma a: A.F(a)) | \pi \circ f = id_A\}$ with the pointwise order

   prove that $\Pi a: A.F(a)$ is a domain. An element $< t_a | a \in A >$ of $\Pi a: A.F(a)$ is called a **continuous section** of F.

   If $in: A \lhd A'$ and $In: F \lhd_p F' \circ in$, prove that $\Pi a: A.F(a) \lhd_p \Pi a: A'.F'(a)$ via $(\Pi in.In): < t_a | a \in A > \mapsto < F'(in \circ out(b) \to b) \circ In_{out(b)}(t_{out(b)}) | b \in A' >$.

   (**Not compulsary**: Is it possible to drop or relax the condition "A pointed", so that $\Pi a: A.F(a)$ is still a domain? No restrictions on F should be imposed)

4. If $F: A \to PDom^E$ and $G: (\Sigma a: A.F(a)) \to Dom_p^E$ are ω-functors, then extend $la: A.\Pi b: F(a).G(a, b)$ to a functor from A to Dom_p^E (**not compulsory**: and prove that it is an ω-functor).

2. l_2-interpretation We consider a subset of the l_2-calculus and a notion of l_2-structure, then you have to define how terms are interpreted in it.

$$\sigma \in Types ::= \quad t \quad | \quad \forall t.\sigma$$

Δ ranges over sequences of type variables $t_1, \ldots, t_n$ without repetitions, Γ ranges over sequences of type assignments $x_1{:}\tau_1, \ldots, x_m{:}\tau_m$ without repetitions of x_is. $FV(\Gamma) \subseteq \Delta$ means that all the free type variables of Γ are in Δ.

$$\text{var}\ \frac{}{\Delta.\Gamma \vdash x{:}\tau}\ x{:}\tau \in \Gamma \text{ and } FV(\Gamma) \subseteq \Delta$$

$$\forall I\ \frac{\Delta, t.\Gamma \vdash e{:}\sigma}{\Delta.\Gamma \vdash \Lambda t.e{:}\forall t.\sigma}\ t \notin FV(\Gamma)$$

$$\forall E\ \frac{\Delta.\Gamma \vdash e{:}\forall t.\sigma}{\Delta.\Gamma \vdash e(\tau){:}[\tau/t]\sigma}\ FV(\tau) \subseteq \Delta$$

M $\equiv < T^{\mathrm{M}}, \forall^{\mathrm{M}}, Set^{\mathrm{M}}, \Lambda^{\mathrm{M}}, App^{\mathrm{M}} >$ is a l_2-**structure** iff (we drop the superscript in $\_^{\mathrm{M}}$):

- T is a pointed domain
- $\forall \in (T \to T) \to T$
- $Set{:}T \to Dom_p^E$ is an ω-functor, we write t instead of $Set(t)$
- $\Lambda \in \Pi F{:}T \to T.(\Pi t{:}T.F(t)) \rightharpoonup \forall(F)$
- $App \in \Pi F{:}T \to T, t{:}T.\forall(F) \rightharpoonup F(t)$

Given a l_2-structure M define the l_2-**interpretation** $[\_]^{\mathrm{M}}$ so that:

1. $[\Delta.\sigma]{:}T^n \to T$, for all Δ s.t. $FV(\sigma) \subseteq \Delta \equiv t_1, \ldots, t_n$.
   $[\Delta.\Gamma]{:}T^n \to Dom_p^E$ is an ω-functor, for all Δ s.t. $FV(\Gamma) \subseteq \Delta$.
2. $[\Delta.\Gamma \vdash e{:}\sigma] \in \Pi a{:}T^n.[\Delta.\Gamma](a) \rightharpoonup [\Delta.\sigma](a)$, for all derivable $\Delta.\Gamma \vdash e{:}\sigma$.

A l_2-structure M is **standard** iff $\Lambda(F)$ is an isomorphism and the "β-axiom" holds, i.e. $App(F,t)(\Lambda(F)(f)) = f(t)$.

3. Universal domains and l_2-structures ([ABL 86] may help):

1. If $U \in Dom_p$, let $\mathrm{Fp}_p(U)$ be $\{in(A) | in{:}A \lhd_p U\}$ with the inclusion order. Prove that $\mathrm{Fp}_p(U)$ is a pointed domain and that $\mathrm{Fp}_p(U) \lhd U \rightharpoonup U$.
2. If $U \rightharpoonup U \lhd U$, then define a standard l_2-structure s.t. $T = \mathrm{Fp}_p(U)$ and $Set(t)$ is the subset t of U with the induced order.

References

[CGW 87] Coquand, T., Gunter, C. and Winskel, G. Domain theoretic models of polymorphism. Report TR-116, Univ. of Cambridge, Computer Laboratory (1987).

[ABL 86] Amadio, R. and Bruce, K. and Longo, G. The finitary projection model for second order lambda calculus and solutions to higher order domain equations. *Proc. IEEE Symp. on Logic in Computer Science*, Cambridge (1986).

Domains (1987)

1. Consider the nondeterministic imperative language

$$c ::= a \mid \textbf{skip} \mid \textbf{if } b \textbf{ then } c \textbf{ else } c' \mid \textbf{while } b \textbf{ do } c \mid c \textbf{ or } c'$$

where there are two given semantic functions, one $\mathcal{D}$: Acom $\to (S \to S)$ for atomic commands, a and another, $\mathcal{B}$: $Bexp \to (S \to T)$ for Boolean expressions, b. This language has an evident semantics $\mathcal{C}$: Com $\to (S \to \mathcal{P}(S))$ where $\mathcal{P}(S)$ is the cpo of all subsets of S under the subset ordering. Give this language a "weakest-liberal-precondition" semantics $\mathcal{H}$: Com $\to (\mathcal{P}(S) \to \mathcal{P}(S))$ so that

$$(*) \qquad \mathcal{H}[\![c]\!](D) = wlp(\mathcal{C}[\![c]\!], D) =_{\text{def}} \{\sigma \mid \mathcal{C}[\![c]\!](\sigma) \subset D\}$$

For example the semantic clause for conditionals might be:

$$\begin{aligned}\mathcal{H}[\![\textbf{if } b \textbf{ then } c \textbf{ else } c']\!] \quad &= \lambda D.(\mathcal{B}[\![b]\!]^{-1}(\text{tt}) \cap \mathcal{H}[\![c]\!](D)) \\ &\quad \cup (\mathcal{B}[\![b]\!]^{-1}(\text{ff}) \cap \mathcal{H}[\![c']\!](D)).\end{aligned}$$

Show that (*) holds. [Hint: For **while**-loops, use greatest fixed-points so that $\mathcal{H}[\![\textbf{while } b \textbf{ do } c]\!](D) = \bigcap_{n \le 0} \Phi^n(S)$ for a certain $\Phi: \mathcal{P}(S) \to \mathcal{P}(s)$.]

2. For P an ω-algebraic cpo (not necessarily with a bottom element), take the relational powerdomain of P to be $\mathcal{P}_R(P) =_{\text{def}} \mathcal{I}(P^0)$, where P^0 is the set of finite elements of P and $\mathcal{I}(P^0)$ is the complete lattice of the ideals of P^0 ordered by subset (ideals being downward-closed subsets, possibly empty).

   Now there is a "weakest-liberal-precondition" isomorphism,

$$\mathcal{W}: P \to \mathcal{P}_R(Q) \cong \mathcal{P}_R(Q) \to_{cm} \mathcal{P}_R(P)$$

   where P and Q are given ω-algebraic cpos and where on the left we take the continuous total maps and on the right we take the completely multiplicative maps, that is those g such that $g(\cap \mathcal{D}) = \bigcap_{D \in \mathcal{D}} g(D)$ for any $\mathcal{D} \subset \mathcal{P}_{\text{R}}(P)$.

   The isomorphism is well-defined by:

$$\mathcal{W}(f)(D) = wlp(f, D) =_{\text{def}} \{a \in P^0 \mid f(a) \subset D\}.$$

   It has inverse well-defined by:

$$\mathcal{W}^{-1}(g)(x) = \bigcap \{D \in \mathcal{P}_{\text{R}}(P) \mid \forall a \in P^0.\, a \sqsubseteq x \supset a \in g(D)\}.$$

   Establish these assertions about $\mathcal{W}$ and $\mathcal{W}^{-1}$.

Domains (1986)

This question presents a nondeterministic functional programming language. You are to provide both an operational and denotational semantics for it and show them equivalent.

The Language

Types: σ ranges over the set TExp of *type expressions* and φ over the set FTExp of *function type expressions.* These two sets are given by:

$$\sigma ::= 1 \mid \mathrm{N} \mid \sigma \times \sigma' \mid \sigma + \sigma'$$
$$\varphi ::= \sigma \to \mathcal{P}(\sigma')$$

It is intended that all type expressions represent sets: 1 is the one-point set, N is the set of natural numbers, $\sigma \times \sigma'$, $\sigma + \sigma'$ are the product and sum constructions. Further $\sigma \to \mathcal{P}(\sigma')$ represents the set of all functions from σ to subsets (in the ordinary sense — possibly empty) of σ'.

Expressions: x ranges over a set Var of variables, f over a set FVar of *function variables*, e over a set Exp of *expressions* and fe over a set of *function expressions.* The latter two sets are partially given by:

$$e ::= x \mid fe(e) \mid e \text{ or } e' \mid \ldots$$
$$fe ::= \lambda x \in \sigma.e \mid f \mid \mu f{:}\varphi.fe$$

It is intended that expressions e of type σ stand for subsets of σ and that function expressions fe of type φ stand for elements of φ. In particular the *function application* $fe(e)$ stands for all possible results of applying fe to an element of e and the *nondeterministic choice* e **or** e' for the union of e and e' (operationally this is nondeterministic choice) and the *abstraction* $\lambda x \in \sigma.e$ stands for the function which, given an element of x of σ, returns the set of results, e, and the *recursive definition* $\mu f{:}\varphi.fe$ stands for the f in φ recursively defined to be fe.

Now do the following:

1. **Syntax:** Fill in the syntax for expressions to cover 1, N, products and sums.

2. **Typing:** Give typing rules to show $e \in \sigma$ for expressions and $fe{:}\varphi$ for function expressions. The rules can be in natural deduction style with assumptions of either the form $x \in \sigma$ or $f{:}\varphi$.

3. **Operational Semantics:** Specify canonical forms, c, and give rules defining relations $e \Rightarrow c$ (read as "e may evaluate to c") and $\langle fe, c\rangle \Rightarrow c'$ (read as "fe may evaluate to c' when applied to c"). As an example here is the rule for recursion:

$$\frac{\langle fe(\mu f{:}\varphi.fe), c\rangle \Rightarrow c'}{\langle \mu f{:}\varphi fe.c\rangle \Rightarrow c'}$$

4. **Denotational Semantics:** Give this according to the intentions written above. Give the set $[\![\sigma]\!]$ that σ represents and the set $[\![\varphi]\!]$ that φ does. A *type-assignment* is a map α: Var $\to$ TExp and a *functional type-assignment* is a map β: FVar $\to$ FTExp. Say what α-environments, ρ and what β-environments, π, are (total functions will work). Give semantic clauses for the denotations

$$\mathcal{E}[\![e; \alpha, \beta]\!](\rho, \pi) \subset [\![\sigma]\!]$$
$$\mathcal{F}[\![fe; \alpha, \beta]\!](\rho, \pi) \in [\![\varphi]\!]$$

where $\alpha, \beta \vdash e \in \sigma$ and $\alpha, \beta \vdash fe{:}\varphi$ in appropriate senses.

5. Show that, for canonical forms, $[\![c]\!]$ is a singleton and $[\![c]\!] = [\![c']\!]$ if and only if $c = c'$. (We drop $\alpha, \beta, \mathcal{E}, \rho, \pi$ when understood or not needed.)

6. **Consistency Theorem:** Show that if $e \Rightarrow c$ then $[\![c]\!] \subset [\![e]\!]$. To do this state the corresponding result for function expressions. You need only present four cases: function application, nondeterministic choice, abstraction, recursive definition. State any substitution lemma, independence lemma or recursion lemma that you require.

7. **Completeness Theorem:** Define the relation $X \leq_\sigma e$ between subsets of $[\![\sigma]\!]$ and closed expressions, e, by:

$$X \leq_\sigma e \text{ iff } \forall a \in X.\exists c.\{a\} = [\![c]\!] \wedge e \Rightarrow c$$

And define the relation $h \leq_\sigma fe$ between elements of $[\![\varphi]\!]$ and closed function expressions, fe, by:

$$h \leq_\varphi fe \text{ iff } \forall a \in [\![\sigma]\!].\forall c.\{a\} = [\![c]\!] \supset \forall b \in h(a).\exists c'.\{b\} = [\![c']\!] \wedge \langle fe, c\rangle \Rightarrow c'$$

where $\varphi = \sigma \rightarrow \mathcal{P}(\sigma')$.

Show that for $e \in \sigma$ (e closed) and $fe{:}\varphi$ (fe closed) the following holds:

(a) $[\![\dot{e}]\!] \leq_\sigma e$

(b) $[\![fe]\!] \leq_\varphi fe$.

To do this you must state the generalisation for open e and fe. You need only present the four cases required in the consistency theorem.

8. Conclude that for closed expressions $e{:}\, [\![e]\!] = \bigcup\{[\![c]\!] \mid e \Rightarrow c\}$.

Domain theory (1985)

This question concerns equational logic for partial algebras where it is (unfortunately) necessary to consider assumptions of existence. An axiom system is given below. You are required to do an example within the system and prove its completeness (*not* its consistency).

Let Σ be a fixed signature throughout. A *partial* Σ-algebra A is a set $|A|$ together with a partial function $f_A{:}\, |A|^n \rightharpoonup |A|$ for each operation symbol (= function symbol) of arity $n \geq 0$. Terms t, u, v are formed from variables and operation symbols in the usual way. For each environment ρ mapping variables to elements of $|A|$, the value $A[\![t]\!](\rho)$ of t is defined in the evident way, but may not exist. (**Note:** ρ is total.)

The logic has two kinds of atomic assertions At, viz $E(t)$ (to be read as "t exists") and $t \equiv u$ (to be read as "t exists iff u does and they are equal when they both exist"). The logic assumes given a set Ax, of such atomic assertions as axioms. It gives rules to derive *sequents* of the form $\Gamma \vdash At$ where Γ ranges over finite sets of the form $\{E(t_1), \ldots, E(t_m)\}$;

so Γ is the set of assumptions of existence.

Here are the rules of derivation of sequents.

Axioms

$\emptyset \vdash At$ (if At is in Ax)

Sequents

1. $\Gamma \cup \{E(t)\} \vdash E(t)$

2. $$\frac{\Gamma \vdash At}{\Gamma \cup \Gamma' \vdash At}$$

Equivalence

1. $\Gamma \vdash t \equiv t$

2. $$\frac{\Gamma \vdash t \equiv u}{\Gamma \vdash u \equiv t}$$

3. $$\frac{\Gamma \vdash t \equiv u, \Gamma \vdash u \equiv v}{\Gamma \vdash t \equiv v}$$

4. $$\frac{\Gamma \cup \{E(t)\} \vdash t \equiv u, \Gamma \cup \{E(u)\} \vdash u \equiv t}{\Gamma \vdash t \equiv u}$$

Existence

1. $\emptyset \vdash E(x)$ (where x is any variable)

2. $$\frac{\Gamma \vdash E(t), \Gamma \vdash t \equiv u}{\Gamma \vdash E(u)}$$

3. $$\frac{\Gamma \vdash E(f(t_1, \ldots, t_n))}{\Gamma \vdash E(t_i)}$$ (where $n \geq 0$ is the arity of f and $1 \leq i \leq n$)

Congruence

$$\frac{\Gamma \vdash t_1 \equiv u_1, \ldots, \Gamma \vdash t_n \equiv u_n}{\Gamma \vdash f(t_1, \ldots, t_n) \equiv f(u_1, \ldots, u_n)}$$ (where $n \geq 0$ is the arity of f)

Substitution

1. $$\frac{\Gamma \vdash t \equiv u, \Gamma \vdash E(v)}{\Gamma \vdash t[v/x] \equiv u[v/x]}$$ (if x does not occur in Γ)

2. $$\frac{\Gamma \vdash E(t), \Gamma \vdash E(v)}{\Gamma \vdash E(t[v/x])}$$ (if x does not occur in Γ)

Here $t[v/x]$ is the term resulting from substituting v for x in t.

So, for example, the axiom rule says that $\emptyset \vdash At$ is derivable when At is in Ax, and the second rule of existence says that if $\Gamma \vdash E(t)$ and $\Gamma \vdash t \equiv u$ are derivable, so is $\Gamma \vdash E(u)$.

Example: The axioms for (partial) group theory are (where we use an infix $\cdot$ for multiplication and write e instead of $e(\ )$, etc.):

Associativity	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
Identity	$x \cdot e \equiv x, \quad e \cdot x \equiv x$
Inverse	$x \cdot x^{-1} \equiv e, \quad x^{-1} \cdot x \equiv e$
Existence	$E(e)$

Show that $\emptyset \vdash E(x^{-1})$ and $\emptyset \vdash E(x \cdot y)$ are derivable.

Returning to semantic issues, write $\models_{A,\rho} E(t)$ to mean $A[\![t]\!](\rho)$ is defined and $\models_{a,\rho} t \equiv u$ to mean that $A[\![t]\!]$ is defined, being d in $|A|$, iff $A[\![u]\!](\rho)$ is defined, being d too. We say A is an *Ax-algebra* if $\models_{A,\rho} At$ for every ρ and every At in Ax. We write $\{E(t_1), \ldots, E(t_m)\} \models_{A,\rho} At$ to mean that if $\models_{A,\rho} E(t_i)$, for $i = 1, \ldots, m$ then $\models_{A,\rho} At$.

Consistency: For every Ax-algebra A, if $\emptyset \vdash At$ is derivable then $\emptyset \models_{A,\rho} At$, for every ρ.

Completeness: If for every Ax-algebra A and every ρ, it is true that $\emptyset \models At$ holds then $\emptyset \vdash At$ is derivable.

You are asked to prove completeness only.

Hint: For every Γ one can define an algebra A_Γ with $|A_\Gamma| = \{[t] \mid \Gamma \vdash E(t)\}$ where $[t]$ is $\{u \mid \Gamma \vdash t \equiv u\}$. $f_A([t_1], \ldots, [t_n])$ is $[f(t_1, \ldots, t_n)]$ if that is in $|A_\Gamma|$, and is otherwise undefined. Look at $A_\Gamma[\![t]\!](\rho)$ for arbitrary t, ρ and use the algebra for suitable choices for Γ and ρ.

Domains (1984)

This question concerns the denotational and operational semantics of the following functional programming language (which can be considered as a fragment of a functional metalanguage for denotational semantics).

Syntax: There is a denumerably infinite supply of *function variables*, f. There are *type expressions* σ, *function type expressions* φ and *function expressions* fe, given by the following grammars (where $n \geq 0$):

Type Expressions: $\sigma ::= \times(\sigma_1, \ldots, \sigma_n) \mid +(\sigma_1, \ldots \sigma_n)$
Function Type Expressions: $\varphi ::= \sigma \rightharpoonup \sigma'$
Function Expressions:

$$fe ::= id_\sigma \mid fe' \circ fe \mid f \mid \mu f{:}\varphi.fe$$
$$\langle fe_1, \ldots fe_n \rangle \mid \pi_{\sigma_1,\ldots\sigma_n, i} \quad (\text{where } 1 \leq i \leq n)$$
$$in_{\sigma_1,\ldots,\sigma_n, i} \text{ (where } 1 \leq i \leq n) \mid [fe_1, \ldots, fe_n]$$

Informal Explanation

There are product and sum types (denoting sets) and function types for partial functions between them. There is the identity function and functions can be composed. Functions can be defined by recursion. Functions can be combined by target tupling and there are the projection functions. There are injection functions and functions can be defined by source tupling.

Now answer the following:

1. Not all expressions make sense. Give typing rules to show when $fe:\varphi$ under the assumptions that $f_1:\varphi_1, \ldots, f_m:\varphi_m$. Here is an example rule written in natural deduction style, leaving irrelevant assumptions implicit and showing cancelled assumptions by the square bracket convention.

$$\dfrac{\begin{array}{c}[f:\varphi] \\ \vdots \\ fe:\varphi\end{array}}{(\mu f{:}\varphi.fe):\varphi}$$

   Show every closed function expression has at most one type.

2. Give the denotations of types σ, as sets $[\![\sigma]\!]$ and of function types φ, as complete partial orders $[\![\varphi]\!]$ of partial functions.

3. Give the denotations $[\![fe]\!]$ in $[\![\varphi]\!]$ of function expressions, fe (where $fe:\varphi$ if $f_1:\varphi_1, \ldots, f_m:\varphi_m$) given denotations α_i in $[\![\varphi_i]\!]$ for f_i $(i = 1, \ldots, n)$. Pay attention to the issue of the continuity of expressions in variables, stating definitions and giving the necessary facts without proof. [**Hint:** You might find it best to introduce constants for elements of $[\![\varphi]\!]$ and begin with the denotations of *closed* expressions.]

4. The set of *canonical* expressions c is given by the following grammar:

$$C ::= \langle c_1, \ldots, c_n\rangle \mid in_{\sigma_1,\ldots,\sigma_n,i}(c) \qquad (n \geq 0)$$

   Give typing rules to show when $c \in \sigma$. Give denotations of canonical expressions so that $[\![c]\!] \in [\![\sigma]\!]$. (Note that $[\![c]\!] = [\![c']\!]$ iff $c = c'$).

5. Give the operational semantics of closed function expressions, fe, by defining an evaluation relation:

$$\langle fe, c\rangle \Rightarrow c'$$

   Show that if $\langle fe, c\rangle \Rightarrow c'$ where $fe:\sigma \rightharpoonup \sigma'$ and $c \in \sigma$ then $c' \in \sigma'$. (State without proof any lemma needed about substitution.)

6. Show that if $\langle fe, c\rangle \Rightarrow c'$ where $fe:\sigma \rightharpoonup \sigma'$ and $c:\sigma$ then $[\![fe]\!]([\![c]\!])$ is defined and equals $[\![c']\!]$. (State without proof any lemma needed about substitution.)

7. Show for the sublanguage *without* recursion that for any closed $fe\colon \sigma \rightharpoonup \sigma'$ and $c \in \sigma$ if $[\![fe]\!]([\![c]\!])$ is defined there is a c' such that $\langle fe, c\rangle \Rightarrow c'$. (**Warning:** This is deliberately stated in a weak form.)

8. Show for the full language that for closed $fe\colon \sigma \rightharpoonup \sigma'$ and $c \in \sigma$ if $[\![fe]\!]$ ($[\![c]\!]$) is defined then $\langle fe, c\rangle \Rightarrow c'$ for some c'.

   Hint: Add a constant $\emptyset_{\sigma,\sigma'}\colon \sigma \rightharpoonup \sigma'$ to denote the empty function from $[\![\sigma]\!]$ to $[\![\sigma']\!]$ and extend the result of (7). Now define abbreviations $\mu^{(n)}f{:}\varphi.fe$ (for $n \geq 0$) by putting $\mu^{(0)}f{:}\varphi.fe =_{def} \emptyset_{\sigma,\sigma'}$ (where $\varphi = \sigma \rightharpoonup \sigma'$) and $\mu^{(n+1)}f{:}\varphi.fe =_{def} fe[\mu^{(n)}f{:}\varphi.fe/f]$. Then say when an expression fe' in the language without recursion *approximates* one fe in the full language and show that then if $\langle fe', c\rangle \Rightarrow c'$ then $\langle fe, c\rangle \Rightarrow c'$. Then show that if $[\![fe]\!]([\![c]\!])$ is defined, so is $[\![fe']\!]([\![c]\!])$ for some approximation fe' of fe. (State without proof any lemma needed about substitution).

9. So far the language is trivial as every type has only finitely many members. Show briefly how you would extend the above to include natural numbers.

Postscript

In the above it is assumed that free and bound variables are defined in the normal way and that so is $fe[fe'/f]$, the substitution of fe' for f in fe.

Domains (1983)

The question is about the possibility of solving "domain equations" in the ordinary category **Set**, and the relation between solutions obtained there and those obtained in categories of cpo's. The solution of domain equations in **Set** was, in effect, proposed in [McC] §6, and has been revived in another form in [AM]. (It is not necessary to read these papers to answer the question, though it might be helpful to look at [AM] for 2.(c) & (d).)

1. Show that **Set** may be identified with (strictly: is isomorphic with) the subcategory of $\mathbf{CPO}_\perp$ consisting of the flat domains and the very strict morphisms. What standard functors on $\mathbf{CPO}_\perp$ cut down to $\times, +$ (product, coproduct) on **Set**, modulo this identification? Let $\mathbf{P}_{\text{fin}}$ be the "finite power set" functor on **Set**, so that $\mathbf{P}_{\text{fin}}(S)$ is the set of finite subsets of S, and $\mathbf{P}_{\text{fin}}$ is defined on morphisms in the obvious way. Show that $\mathbf{P}_{\text{fin}}$ can be considered as (the restriction of) one of the usual power domains. Can anything similar be done with function space (exponentiation)?

2. (a) State a version of the initial fixed-point theorem ([Plo] Ch.5) for **Set** rather than $\mathbf{CPO}_E$; also state a *greatest* fixed-point theorem for **Set** (or, what is the same thing, an initial fixed-point theorem for $\mathbf{Set}^{\text{op}}$), formulating carefully the notions of continuity of functors needed for the initial and greatest fixed points, respectively (call them ω-continuity and ω^{op}-continuity). [The use of greatest fixed-points is advocated in [AM]. It would perhaps be better to call these fixed-points "terminal" or "final" rather than "greatest".] Show that $\mathbf{P}_{\text{fin}}$ is ω-continuous but not ω^{op}-continuous in **Set**.

   (b) Show informally (full details not required) that, at least for the "usual" functors which preserve injections, the identification of **Set** with a subcategory of $\mathbf{CPO}_\perp$ extends to initial fixed-points, so that the initial fixed-point theorem for **Set** can be considered as a restricted case of the initial algebra theorem ([Plo] Ch.5, Th.2).

(c) A domain of finite and infinite binary trees may be introduced as the initial solution of

$$T \cong \mathcal{O} \oplus A \otimes T_\perp \otimes T_\perp \tag{1}$$

(A is a flat domain of labels, $\mathcal{O}$ is the two-point domain.) As pointed out by [AM], one could also use the greatest solution in **Set** of

$$T \cong 1 + A \times T \times T \tag{2}$$

However, the "domains" obtained as solutions of (1) and (2) are not quite the same, even as sets; point out the difference, and briefly explain how it arises. How, by a slight modification of (2), might we define a "domain" of infinite trees only? What happens if the corresponding modification is made in (1)?

(d) (Optional) Any comments?

References

[McC] J. McCarthy. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, North-Holland 1963.

[AM] M. Arbib and E. Manes. Parameterized data types do not need highly constrained parameters. *Information and Control* 52, 139-158 (1982).

[Plo] G. Plotkin, lecture notes.

Domains (1982)

The category **Equiv** has as objects structures $(X, \equiv)$ where $\equiv$ is an equivalence relation over X and as morphisms $f\colon (X,\equiv) \to (X',\equiv')$ total functions $f\colon X \to X'$ such that for any a, b in X if $a \equiv b$ then $f(a) \equiv' f(b)$.

(a) Show that **Equiv** has finite products and coproducts. Define a "finite-set" functor $\mathcal{F}$:**Equiv**→**Equiv** so that on objects

$\mathcal{F}(X, \equiv) = \langle \{u \mid u \text{ is a finite subset of } X\}, \text{(to be filled in)} \rangle$

Define a "finite-function" functor $\mathbf{Map}_A$: **Equiv** → **Equiv** (A any *set*) so that on objects

$$\mathbf{Map}_A(X,\equiv) = \langle \{f\colon u \to X \mid u \text{ is a finite subset of } X\}, \text{(to be filled in)} \rangle$$

(b) A morphism $f\colon (X,\equiv) \to (X',\equiv')$ is an *inclusion* if $X \subseteq X'$ and for all a, b in X if $a \equiv b$ then $a \equiv' b$ (i.e. $\equiv \subseteq \equiv'$). Let $\mathbf{Equiv}^{\mathrm{I}}$ be the subcategory of inclusions. Show it has ω-colimits of chains $\langle (X_n, \equiv_n), f_n \rangle$. What should the definition of a continuous functor,

$$\mathbf{G}\colon (\mathbf{Equiv}^I)^n \to \mathbf{Equiv}^I$$

of n arguments be? Show that all the functors considered in part (a) when specialised to inclusions yield functors over $\mathbf{Equiv}^I$ which are continuous *(The obvious sense of specialisation is intended).*

(c) Show, with suitable definitions, that every continuous $\mathbf{G}: \mathbf{Equiv}^{\mathrm{I}} \to \mathbf{Equiv}^{I}$ has an initial fixed-point (=algebra), $\eta_{\mathbf{G}}: \mathbf{G}(Fix_{\mathbf{G}}) = Fix_{\mathbf{G}}$ (*note the equality!*). Show that if $\mathbf{F}: \mathbf{Equiv} \to \mathbf{Equiv}$ specialises to such a $\mathbf{G}$ then the initial $\mathbf{G}$-algebra is also the initial $\mathbf{F}$-algebra. Finally demonstrate a *Principle of Structural Induction* for $Fix_{\mathbf{G}}$ and illustrate it in the cases

(i) *Integers*: $N = 1 + N$
(ii) *Hereditarily Finite Sets over X*: $H = \mathcal{F}(X + H)$
(iii) *A-trees over X*: $T = (X \times \mathbf{Map}_A T)$

(d) Let Σ be a one-sorted signature. What should the category of Σ-algebras over **Equiv** be? Let E be a collection of equations in the signature Σ. When should a Σ-algebra satisfy E? Show that there is a free Σ-algebra $Alg_{\Sigma,E}(X, \equiv)$ satisfying E over any $(X, \equiv)$. Show too that $Alg_{\Sigma,E}$ yields a functor $\mathbf{F}_{\Sigma,E}: \mathbf{Equiv} \to \mathbf{Equiv}$ where on objects

$$\mathbf{F}_{\Sigma,E}(X, \equiv) = \text{equivalence relation of } Alg_{\Sigma,E}(X, \equiv)$$

Demonstrate that $\mathbf{F}_{\Sigma,E}$ specialises to a continuous functor on $\mathbf{Equiv}^{I}$. What happens if you try to obtain $\mathcal{F}$ as such a $\mathbf{F}_{\Sigma,E}$?

(e) Indicate briefly what would happen in the above if inclusions were replaced by strict inclusions where an inclusion $f: (X, \equiv) \to (X', \equiv')$ is *strict* iff for all a, b in X we have $a \equiv b$ iff $a \equiv b'$ (i.e. $\equiv \; = \; \equiv' \restriction x$).

Domains (1981)

A *nondeterministic cpo (ndcpo)* is a structure $\langle D, \sqsubseteq, \cup \rangle$ where $\langle D, \sqsubseteq \rangle$ is a cpo and $\cup: D^2 \to D$ is a binary continuous function such that:

Associativity: $x \cup (y \cup z) = (x \cup y) \cup z$	$(x, y, z \in D)$
Commutativity: $x \cup y = y \cup x$	$(x, y \in D)$
Absorption: $x \cup x = x$	$(x \in D)$

A function $f: D \to E$ over ndcpos is *linear* if for all x, y in D we have:

$$f(x \cup y) = f(x) \cup f(y)$$

The category **ND** has the ndcpos as objects and the linear continuous maps as morphisms.

1. Show that **ND** has (binary) products.
2. Define the subcategory of embeddings, $\mathbf{ND}^{\mathrm{E}}$.
3. Show $\mathbf{ND}^{\mathrm{E}}$ has direct limits of ω-chains and there is a simple local criterion for the universality of cones.
4. Show how to turn product into a continuous functor on $\mathbf{ND}^{\mathrm{E}}$.

Domains (1980)

The untyped λ-calculus has the following syntax:

Var: a set of *variables*, ranged over by $x, y, \ldots$.
Exp: a set of *expressions*, ranged over by e, and given by

$$e ::= x \mid e(e) \mid (\lambda x.e)$$

Let D be a cpo and suppose $(D \to D) \xrightarrow{\varphi} D \xrightarrow{\psi} (D \to D)$ are continuous functions of the indicated types between D and the space $D \to D$ of all continuous functions from D to D.

(a) Define the natural denotational semantics $\mathcal{E}: Exp \to Env \to D$ using φ and ψ, where $Env = Var \to D$ is the set of environments.

(b) Show that for all expressions e and e', variables x, and environments ρ, that, in an evident notation:

$$\mathcal{E}[\![[e'/x]e]\!](\rho) = \mathcal{E}[\![e]\!](\rho)[\mathcal{E}[\![e']\!](\rho)/x])$$

You may assume, for the sake of simplicity, that x is not a bound variable of e.

(c) Show that, assuming $\varphi \circ \psi = id$, the following restricted version of the β-rule is true:

$$(\beta) \quad (\lambda x.e)(e') = [e'/x]e \quad (x \text{ does not occur bound in } e)$$

(d) Show that the η-rule is true iff $\varphi \circ \psi = id$; the η-rule is

$$(\eta) \quad (\lambda x.e(x)) = e \quad (x \text{ not free in } e)$$

You can assume that for all expressions e and environments ρ and ρ', $\mathcal{E}[\![e]\!](\rho) = \mathcal{E}[\![e]\!](\rho')$ whenever $\rho[\![x]\!] = \rho'[\![x]\!]$ for any variable x free in e.

(e) Let A be an arbitrary set (of "atoms"); let S be the least set such that:

$$S = A \cup (\mathcal{P}_\omega(S) \times S)$$

where for any set X, $\mathcal{P}_\omega(X) = \{U \subseteq X \mid U \text{ finite }\}$ is the set of finite subsets of X. In other words, $S = \cup_{n \geq 0} S_n$ where $X_0 = \emptyset$ and $S_{n+1} = A \cup (\mathcal{P}_\omega(\mathcal{S}) \times \mathcal{S})$. Now let D be the complete lattice $\mathcal{P}(\mathrm{S})$ of all subsets of S and define φ and ψ by:

$$\varphi(f) = \{\langle X, s\rangle | X \in \mathcal{P}_\omega(S), s \in S, s \in f(X)\}$$
$$\psi(d)(e) = \{s | \exists X \in \mathcal{P}_\omega(S).\langle X, s\rangle \in d \text{ and } X \subseteq e\}$$

Show that $\psi \circ \varphi = id$. What is the relation between $\varphi \circ \psi$ and id_D and what does this portend for the η-rule?

(f) Prove the following formulae (where X, Y range over $\mathcal{P}_\omega(S)$ and s ranges over S):

$$\mathcal{E}[\![(\lambda x.x)]\!](\rho) = \{\langle X, s\rangle \mid s \in X\}$$
$$\mathcal{E}[\![(\lambda x.(\lambda y.x))]\!](\rho) = \{\langle X, \langle Y, s\rangle\rangle \mid s \in X\}$$
$$\mathcal{E}[\![(\lambda x.x(x))]\!](\rho) = \{\langle \{\langle X, s\rangle\} \cup Y, s\rangle \mid X \subseteq Y\}$$
$$\mathcal{E}[\![x.x(x))(\lambda x.x(x))]\!](\rho) = \varphi$$

What do you guess $\mathcal{E}[\![\lambda x.\lambda y.\lambda z.x(z)(y(z))]\!](\rho)$ is?

Semantics (1979)

Let *Stat* be *any* solution (i.e. not necessarily the standard one obtained by inverse limits or by retracts) of the cpo equation

$$Stat = Act_\perp + [Stat \times Stat]$$

where *Act* is a set of basic actions (primitive statements).

(a) Write down, in terms of the standard functions over cartesian products and coalesced sums of cpos, definitions of the natural abstract syntax operations on *Stat*:

$$\begin{array}{ll} isprim, isseq: & Stat \to T \\ mkprim & : Act_\perp \to Stat \\ mkseq & : Stat \times Stat \to Stat \\ act & : Stat \to Act_\perp \\ stat_1, stat_2 & : Stat \to Stat \end{array}$$

(b) Let $copy: Stat \to Stat$ be the least fixed point solution of the recursive definition

$$\begin{aligned} copy(s) = \; & isprim(s) \to mkprim(act(s)) \mid \\ & isseq(s) \to mkseq(copy(stat_1(S)), copy(stat_2(s))) \mid \perp_{Stat} \end{aligned}$$

Let $P[s]$ be an ω-inductive property of $s \in Stat$, and assume the following hold:

i. $P[\perp]$

ii. $\forall a \in Act.\ P[mkprim(a)]$

iii. $\forall s_1, s_2 \in Stat.\ P[s_1] \wedge P[s_2] \Rightarrow P[mkseq(s_1, s_2)]$

Prove that

$$\forall s \in Stat.\ P[copy(s)].$$

(c) Let C be a cpo with (continuous) functions

$$\begin{array}{ll} \mathcal{A} & : Act_\perp \to C \\ compose & : C \times C \to C \end{array}$$

and define $\mathcal{C}: Stat \to C$ as the least fixed point solution of

$$\begin{aligned} \mathcal{C}(s) = \; & isprim(s) \to \mathcal{A}(act(s)) \mid \\ & isseq(s) \to compose(\mathcal{C}(stat_1(s), \mathcal{C}(stat_2(s))) \mid \perp_C \end{aligned}$$

Prove that

$$\forall s \in Stat.\ \mathcal{C}(copy(s)) = \mathcal{C}(s).$$

(d) Let $Q[c]$ be an ω-inductive property of $c \in C$. Combine the results of parts (b) and (c) to state and prove a principle for inferring

$$\forall s \in Stat.\ Q[\mathcal{C}(s)]$$

(e) Comment briefly (about 75 words) on why an extra axiom

$$\forall s \in Stat.\ copy(s) = s$$

was not needed for part (d), indicating what this axiom would tell you about Stat.

[*Note*: For proofs by induction mention clearly the induction rule you are using].

Semantics (1978)

Let Σ be a ranked alphabet of operators; that is, each $\sigma \in \Sigma$ has an arity of $n \geq 0$. Denote by Σ_n the subset of Σ of operators with rank n.

$W_\Sigma(\{\bot\})$ is the set of words generated for the symbol $\bot$, that is

i. $\bot$ is a word

ii. If $w_1, \ldots, w_n$ are words, and $\sigma \in \Sigma_n$, then $\sigma w_1 \ldots w_n$ is a word.

There are two ways of defining a partial order over $W_\Sigma(\{\bot\})$:

(a) $w \leq w'$ iff this can be inferred in a finite number of steps from

i. $\bot \leq w$ for every word w

ii. If $w_i \leq w_i'$ for $1 \leq i \leq n$ and $\sigma \in \Sigma_n$ then $\sigma w_1 \ldots w_n \leq w_1' \ldots w_n'$

(b) $w \sqsubseteq w'$ iff w' may be got from w by replacing all occurrences of $\bot$ in w by words; that is, if $w = w[\bot, \ldots, \bot]$ then $w' = w[v_1, \ldots, v_n]$ for some words v_i.

1. Prove that (a) and (b) are equivalent, i.e. show that

$$\forall w, w'. w \leq w' \Leftrightarrow w \sqsubseteq w'$$

(You will need structural induction, or induction on the number of steps of an inference under (a), or both).

2. Prove also that $\leq$ (or $\sqsubseteq$) is indeed a partial order.

3. Show that any pair of words w, w' has a greatest lower bound under $\leq$ (or $\sqsubseteq$).

4. Discuss (in about 200 words) the importance of $W_\Sigma(\{\bot\})$ in semantics.

Semantics (1978)

We can generalise from sums and products of cpos over sets, such as $\sum_{i \in S} D_i$ or $\prod_{i \in S} D_i$, to sums and products over cpos.

Definition. Let D be a cpo. It is therefore a category with at most one morphism from one object (= element of D) to another and there is a morphism from d to d' iff $d \sqsubseteq d'$ (in which case the pair d, d' determines it and we call the morphism $d \to d'$ here). A functor $\mathbf{F}$: $D \to \mathbf{O}_\bot^E$ is *ω-continuous* iff whenever $\langle d_n \rangle$ is an increasing sequence in D with lub d then the co-chain

$$\triangle = \langle \mathbf{F}(d_n), \mathbf{F}(d_n \to d_{n+1}) \rangle$$

has as co-limiting cone $\rho: \Delta \to \mathbf{F}(d)$ where $\rho_n = \mathbf{F}(d_n \to d)$.

Now if D is a cpo and $\mathbf{F}: D \to \mathbf{O}_{\perp}^{E}$ is an ω-continuous functor we define $\sum_D \mathbf{F}$ to be the set $\{\langle d, e\rangle \mid e \in \mathbf{F}(d)\}$ ordered by:

$$\langle d, e\rangle \sqsubseteq \langle d', e'\rangle \text{ iff } d \sqsubseteq d' \wedge \mathbf{F}(d \to d')(e) \sqsubseteq e'$$

(a) Show that $\sum_D \mathbf{F}$ is a cpo.

(b) (Examples)

i. Show that $D_{\perp} = \sum_{\mathcal{O}} \mathbf{F}$ where $\mathbf{F}(\perp) = U, \mathbf{F}(\top) = D$, and $\mathbf{F}(\perp \to \top) = \perp_{U,D}$.

ii. Show that $D_{\perp} + E_{\perp} = \sum_T \mathbf{F}$ where $\mathbf{F}(\perp) = U, \mathbf{F}(tt) = D, \mathbf{F}(ff) = E, \mathbf{F}(\perp \to tt) = \perp_{U,D}$, and $\mathbf{F}(\perp \to ff) = \perp_{U,E}$.

iii. Show that $D \times E = \sum_D \mathbf{F}$ where $\mathbf{F}(d) = E$ and $\mathbf{F}(d \to d') = id_E$.

iv. What about denumerable sums?

For products, we say that a function $f: D \to \bigcup_{d \in D} \mathbf{F}(d)$ is *well-typed* iff $\forall d \in D. f(d) \in \mathbf{F}(d)$. It is *monotonic* iff

$$\forall d, d' \in D. d \sqsubseteq d' \Rightarrow \mathbf{F}(d \to d')(f(d)) \sqsubseteq f(d').$$

It is *continuous* iff whenever $\langle d_n \rangle$ is an increasing sequence with lub d then

$$f(d) = \bigsqcup_n \mathbf{F}(d_n \to d)(f(d_n)).$$

We define $\prod_D \mathbf{F}$ to be the set

$$\prod_D \mathbf{F} = \{f: D \to \bigcup_{d \in D} \mathbf{F}(d) \mid f \text{ is well-typed and continuous}\}$$

with the pointwise ordering:

$$f \sqsubseteq g \text{ iff } \forall d \in D.\ f(d) \sqsubseteq g(d).$$

(c) Show that $\prod_D \mathbf{F}$ is a cpo.

(d) (Examples)

i. Show that $D \times E = \prod_T \mathbf{F}$ where $\mathbf{F}$ is as in part b(ii).

ii. Show that $D \to E = \prod_D \mathbf{F}$ where $\mathbf{F}$ is as in part b(iii).

iii. What about denumerable products?

11 Operational semantics

Operational Semantics (1988)

This question is divided into two parts. For 2/3 credit, answer the first part, and for full credit answer the second as well.

Part I

Consider the following fragment of ML:

$$e \ ::= \ x \mid c \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid \texttt{ref}\, e \mid \texttt{!}e \mid e := e'$$

The metavariable x ranges over the set $\mathcal{X}$ of variables, and the metavariable c ranges over some set $\mathcal{C}$ of constants. Informally, $\texttt{ref}\, e$ allocates a new cell in storage, initializes it to the value of e, and returns a reference to that cell. If e evaluates to a reference, then $\texttt{!}e$ returns the contents of the corresponding cell, and $e := e'$ replaces its contents with the value of e'. This can be made precise by defining an operational semantics which employs an explicit store.

Let $\mathcal{V} = \mathcal{C} \cup \mathcal{A}$ be the set of values, consisting of the constants and a set $\mathcal{A}$ of *addresses*. Let $\mathcal{S} = \mathcal{A} \rightarrow_{fin} \mathcal{V}$ be the set of *stores*, which are finite functions[3] from $\mathcal{A}$ to $\mathcal{V}$. Let $\mathcal{E} = \mathcal{X} \rightarrow_{fin} \mathcal{V}$ be the set of *environments*. Let v range over $\mathcal{V}$, a over $\mathcal{A}$, s over $\mathcal{S}$, and E over $\mathcal{E}$. The store $s[a \leftarrow v]$ agrees with s on addresses other than a, and sends a to v. Similarly, the environment $E[x \leftarrow v]$ agrees with E on variables other than x, and sends x to v. The function $next(s)$ yields $a \in \mathcal{A}$ such that $a \notin \mathrm{dom}(s)$. Assume that there is a special constant $done \in \mathcal{C}$.

The relation $E \vdash e, s \Rightarrow v, s'$ is defined to be the least relation closed under the following rules of inference:

$$E \vdash c, s \Rightarrow c, s$$

$$\frac{x \in \mathrm{dom}(E)}{E \vdash x, s \Rightarrow E(x), s}$$

$$\frac{E \vdash e, s \Rightarrow v, s' \quad E[x \leftarrow v] \vdash e', s' \Rightarrow v', s''}{E \vdash \texttt{let}\ x = e\ \texttt{in}\ e', s \Rightarrow v', s''}$$

$$\frac{E \vdash e, s \Rightarrow v, s'}{E \vdash \texttt{ref}\, e, s \Rightarrow next(s'), s'[next(s') \leftarrow v]}$$

$$\frac{E \vdash e, s \Rightarrow a, s' \quad a \in \mathrm{dom}(s')}{E \vdash \texttt{!}e, s \Rightarrow s'(v), s'}$$

$$\frac{E \vdash e, s \Rightarrow a, s' \quad E \vdash e', s' \Rightarrow v, s''}{E \vdash e := e', s \Rightarrow done, s''[a \leftarrow v]}$$

A natural type system for this language is based on taking the following language of types:

$$\tau \ ::= \ b \mid \tau\ \textbf{ref} \mid \textbf{cmd}$$

[3]That is, partial functions with finite domain.

The metavariable b ranges over some set of base types. The type τ **ref** is the type of references to cells containing a value of type τ, and the type **cmd** is a distinguished base type. Let CT be a function assigning a base type to each constant, with $CT(\mathit{done}) = \mathbf{cmd}$.

The problem is to define a type system for the above language and prove that it is sound. More precisely,

1. Based on the informal description given above, define the relation $s \models v\colon \tau$ specifying the values of type τ relative to a store s.

2. Define a *monomorphic* type inference system for deriving assertions of the form $T \vdash e\colon \tau$ (where T is a *type environment* assigning types to variables.)

3. Prove that the type system so defined is *sound* with respect to the operational semantics in the sense that if an expression is assigned a type τ and evaluates, in a store s, to a value v and a new store s', then v has type τ, relative to s'.

Part II

The type system discussed in Part I is based on the fact that, in a given store, every value has at most one type. In this part of the question we consider some of the problems that arise by admitting a degree of polymorphism. To do so, we extend the language to include the following operations on lists:

$$e \ ::= \ \texttt{nil} \mid e::e' \mid \texttt{hd}(e) \mid \texttt{tl}(e)$$

The operational semantics is extended to account for the new expression forms in the obvious way, taking `nil` and $v::v'$ as values. The language of type expressions is extended to include types of the form τ **list** whose values are `nil` and $v::v'$, where v is a value of type τ and v' is a value of type τ **list**. Note that `nil` is a value of every list type.

We shall now extend the type inference system defined in Part I to admit a polymorphic typing rule for `let`, so that the following program is well-typed:

```
let r = nil
 in let x = 1 :: r
     in true :: r
```

On the other hand, the following program should *not* be admitted as well-typed:

```
let r = ref nil
 in let x = (r := (1 :: nil))
     in let y = (r := (true :: nil))
         in (!r)+1
```

Recall that for pure ML, the language of type expressions is extended to include *type variables* t. In addition, a set of *type schemes* σ of the form $\forall t_1, \ldots, t_n.\tau$ is introduced in order to express polymorphism. An *instance* of a type scheme $\forall t_1, \ldots, t_n.\tau$ is a type τ' obtained by replacing each t_i in τ by a type expression; write $\sigma > \tau'$ when this is the case. Given a type environment T (assigning type schemes to variables) and a type τ, $\overline{T}(\tau)$ is defined to be the type scheme $\forall t_1, \ldots, t_n.\tau$ such that t_i occurs free in τ but not in the type scheme assigned to any variable by T. Soundness of the pure system is proved by taking the values of a type scheme σ to be the values of every closed instance of σ.

The above examples show that the usual polymorphic typing rule for `let` is not sound in the presence of references. A crude, but effective, way to ensure soundness is to distinguish two cases, based on the form of the expression bound to the identifier by a `let`:

$$\frac{e \in \mathcal{C} \quad T \vdash e\colon \tau \quad T[x \leftarrow \overline{T}(\tau)] \vdash e'\colon \tau'}{T \vdash \texttt{let } x = e \texttt{ in } e'\colon \tau'}$$

$$\frac{e \notin \mathcal{C} \quad T \vdash e\colon \tau \quad T[x\colon \tau] \vdash e'\colon \tau'}{T \vdash \texttt{let } x = e \texttt{ in } e'\colon \tau'}$$

The idea is that the constants do not create references to the store, and hence may be safely taken to be polymorphic. (More refined criteria are available, but this one will serve to illustrate the method.) The remaining rules for the type inference system follow the pattern of Part I, with the exception of the variable rule which must employ the type scheme instantiation relation.

The problem is to formulate and prove a soundness theorem for this type system with respect to the operational semantics. The principal complication arises from the fact that `nil` is polymorphic, making it difficult to define the values of a type relative to a store in a way that is useful for establishing soundness. One way to handle this problem is to introduce *store typings* assigning types to addresses, and then to define a 4-place relation $s\colon S \models v\colon \tau$ defining the values of type τ with respect to a *typed* store. The definition may be given uniformly, regardless of whether or not S and τ contain type variables. You will need to extend this relation to type schemes, following the informal motivation given above, and to environments. Once you have defined this relation, formulate and prove the soundness theorem for the type inference system. As a check, verify that the unrestricted `let` rule is unsound.

Operational Semantics (1987)

1. Design some additional rules for raising and handling exceptions in the SMC machine. For this purpose, use the **raise** and **exception** constructs introduced in section 4.6 of [MM]. Do not give any formal proofs, but justify your design by discussion.

2. Complete the proof of the second part of Theorem 4.4 in [MM], the equivalence of two semantic definitions:

$$p, M \longrightarrow^* \mathbf{null}, M' \quad \text{iff} \quad p, M \Longrightarrow M'$$

   [Deal carefully with *all* the cases not treated in [MM].]

3. Define an SECD machine derivative that supports both static and dynamic scoping of variables. Assume the existence of a new feature, $dynamic(x)$, which returns the value of x in the dynamic environment. Undecorated variables should be bound in the static environment. Show the steps necessary to evaluate the expression

$$(\lambda y.((\lambda f.(\lambda y.f\ 3)\ 4)(\lambda x.x + y - dynamic(y)))\ 5$$

   on your machine.

4. Show how the expression

$$\textbf{let fun } f\ x = \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * f(x-1) \textbf{ in } f\ 2$$

can be represented as a graph suitable for evaluation using graph reduction. Describe the steps necessary to reduce this expression to WHNF (Weak Head Normal Form), illustrating your answer with the intermediate states of the graph where appropriate.

References

[MM] R. Milner and K. Mitchell. Language semantics and implementation (lecture notes).

Operational Semantics (1986)

Consider the following type inference rules (from [DM 82]) for the **let**-lambda-calculus:

TAUT: $A \vdash x{:}\sigma \quad (x{:}\sigma \text{ in } A)$

INST: $$\frac{A \vdash e{:}\sigma}{A \vdash e{:}\sigma'} \quad (\sigma > \sigma')$$

GEN: $$\frac{A \vdash e{:}\sigma}{A \vdash e{:}\forall\alpha\sigma} \quad (\alpha \text{ not free in } A)$$

COMB: $$\frac{A \vdash e{:}\tau' \to \tau \quad A \vdash e'{:}\tau'}{A \vdash (ee'){:}\tau}$$

ABS: $$\frac{A_x \cup \{x{:}\tau'\} \vdash e{:}\tau}{A \vdash (\lambda x.e){:}\tau' \to \tau}$$

LET: $$\frac{A \vdash e{:}\sigma \quad A_x \cup \{x{:}\sigma\} \vdash e'{:}\tau}{A \vdash (\textbf{let } x = e \textbf{ in } e'){:}\tau}$$

In these rules, *types* τ are built by arrow ($\to$) from primitive types ι and type variables α, and *type schemes* σ are of the form $\forall\alpha_1 \ldots \forall\alpha_m\tau$. $\sigma > \sigma'$ (σ' is a *generic instance* of σ) means that $\sigma = \forall\alpha_1 \ldots \forall\alpha_m\tau$ and $\sigma' = \forall\beta_1 \ldots \forall\beta_n\tau'$, where $\beta_1, \ldots, \beta_n$ are not free in σ and $\tau' = [\tau_1/\alpha_1, \ldots \tau_m/\alpha_m]\tau$. Assumptions A are of the form $\{x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k\}$ where $x_1, \ldots, x_k$ are distinct, and A_x means A with any element $x{:}\sigma$ removed.

1. An expression e of the **let**-lambda calculus may be converted into a **let**-free expression $\tilde{e}$, by repeatedly replacing any sub-expression of the form **let** $x = e_1$ **in** e_2 by $[e_1/x]e_2$, the result of substituting e_1 for free occurrences of x in e_2 (with appropriate bound variable changes). Define $\tilde{e}$ inductively on the structure of e, and discuss briefly what semantic property will ensure that e and $\tilde{e}$ are semantically equivalent. (This type is independent of type inference).

2. Give a detailed proof of the following theorem about type inference:

   $A \vdash e{:}\sigma$ implies $A \vdash \tilde{e}{:}\sigma$

   stating any properties of generic instantiation which you require.

3. Is the converse of this theorem in general? If not, demonstrate the falsity and postulate a condition on e under which it should hold. Then give a proof of the (modified) converse. You may use the following without proof:

   Lemma: If $B \vdash [e'/y]e : \sigma$, and y occurs free in e, then for some σ', $B \vdash e' : \sigma'$ and $B_y \cup \{y : \sigma'\} \vdash e : \sigma$.

References

[DM 82] L. Damas and R. Milner. Principal type schemes for functional programs. Proc. 9th ACM Symp. on Principles of Programming Languages, 1982.

Operational Semantics (1985)

In an extension of the untyped lambda-calculus, the syntax of expressions $e \in \mathrm{Exp}$ is as follows:

$$e ::= x \mid k \mid e\,e \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \mid \lambda x.e$$
$$\mid \textbf{let } x = e \textbf{ in } e \mid \textbf{letrec } y = \lambda x.e \textbf{ in } e$$

where $k \in \mathrm{Const}$ and $x, y \in \mathrm{Var}$ (constants and variables).

Without recursion, one can give operational semantics in terms of Val, Clos and Env (values, closures and environments) as the smallest sets satisfying the equations

$$\mathrm{Val} = \mathrm{Const} \times \mathrm{Val}^* + \mathrm{Clos}$$
$$\mathrm{Clos} = \mathrm{Var} \times \mathrm{Exp} \times \mathrm{Env}$$
$$\mathrm{Env} = \mathrm{Var} \xrightarrow{\mathrm{FIN}} \mathrm{Val} \qquad \text{(finite maps)}.$$

(The two different kinds of value may be written $kv_1 \ldots v_n, [x, e, \rho]$).

1. To handle recursion, assume larger sets $\mathrm{Val}_{\mathrm{INF}}$, $\mathrm{Clos}_{\mathrm{INF}}$ and $\mathrm{Env}_{\mathrm{INF}}$ satisfying the above equations, but allowing infinitary closures; in particular assume (for each x, e, ρ) a unique closure c — denoted $\mu y.[x, e, \rho]$ — such that $c = [x, e, \rho + (y, c)]$. (The environment $\rho' = \rho + (y, c)$ is equal to ρ except that $\rho'(y) = c$). Now define an operational semantics $\vdash_{\mathrm{FIN}} \subseteq \mathrm{Env}_{\mathrm{INF}} \times \mathrm{Exp} \times \mathrm{Val}_{\mathrm{INF}}$, where

   $$\rho \vdash_{\mathrm{INF}} e \to v$$

   means that e evaluates to v in environment ρ. Assume call-by-value, and also assume that if e_1 and e_2 have values $kv_1 \ldots v_n$ and v, then $e_1 e_2$ has values $kv_1 \ldots v_n v$.

2. As an alternative, to avoid infinitary values, consider sets $\mathrm{Val}_{\mathrm{FIN}}$, $\mathrm{Clos}_{\mathrm{FIN}}$ and $\mathrm{Env}_{\mathrm{FIN}}$ as the smallest solution of

   $$\mathrm{Val} = \mathrm{Const} \times \mathrm{Val}^* + \mathrm{Clos}$$
   $$\mathrm{Clos} = \mathrm{Var} \times \mathrm{Exp} \times \mathrm{Env} + \mathrm{Var} \times \mathrm{Exp} \times \mathrm{Env} \times \mathrm{Var}$$
   $$\mathrm{Env} = \mathrm{Var} \xrightarrow{\mathrm{FIN}} \mathrm{Val}$$

Here the second form of closure, $c = [x, e, \rho, y]$, is intuitively understood as $[x, e, \rho]$ except that, in evaluating e, y is bound to c itself. Define an operational semantics $\vdash_{\text{FIN}} \subseteq \text{Env}_{\text{FIN}} \times \text{Exp} \times \text{Val}_{\text{FIN}}$, under the same assumptions as in (1).

3. Formulate a theorem which relates $\vdash_{\text{INF}}$ and $\vdash_{\text{FIN}}$ in the most satisfactory way, and prove it. (This part of the question carries most weight).

4. Will your theorem still hold if you extend the language to allow non-determinism in some form, for example the expression form "e **or** e"? If so, will your proof also extend? Discuss in not more than 300 words.

Operational semantics (1984)

This question refers to [Mil 84].

(a) An alternative operational semantics for the ML subset is required, in which each sentence

$$\rho, \eta \vdash \gamma \rightarrow \gamma'$$

(where $\gamma, \gamma' \in \Gamma$ are configurations) represents a single evaluation step rather than a complete evaluation. Define carefully an appropriate class Γ of configurations, and give the required semantics.

(b) (This part does not depend on (a) above.) It may be loosely claimed of the given semantics that "whenever an expression evaluates to an address, then the address points to a well-defined value".

How would you formulate this claim precisely? Give essential details of the inductive proof, if necessary correcting the semantics to ensure the claim.

(c) (This depends partly on (b) above.) Suppose that it is required to change the semantics so that, informally, "whenever an exception is handled, the store reverts to what it was before". Find a sensible precise form of this imprecise requirement, and suggest how the given semantics may be adjusted (almost) to meet the requirement. Would you need to impose constraints on the language, to preserve the truth of (b) above?

References

[Mil 84] R. Milner. Operational semantics of an ML subset. Internal Research Memo, 1984.

Operational Semantics (1983)

The simple programming language L (in [Plo], p.14 and elsewhere) is to be augmented by two commands to allow "escape" values to be "trapped".

$$c ::= \ldots \mid \textbf{escape}\ e \mid c\ \textbf{trap}\ v$$

The first command escapes from the normal control flow with the value of expression e, and the second traps any escape generated by c, assigning the escape value to variable v.

1. Give two forms of operational semantics for this language: (a) an SMC-type machine, and (b) an axiomatized transition system. (You need not detail the evaluation of expressions in either case.)

2. How would you state the theorem which expresses the equivalence of these two semantic definitions? Discuss, without carrying out the proof of equivalence, the inductive methods which it may require, and whether the propositions which are proved by induction need to be stronger than the statement of the theorem.

References

[Plo] G.D. Plotkin. A structural approach to operational semantics. Technical report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

Operational Semantics (1980)

We modify the language of recursion equations and its reduction-rule semantics [GW1] to include non-deterministic choice symbolised by **or**.

Suppose the defining evironment for functions is

$$f_i(X_1, \ldots, X_{n_i}) = T_i \qquad (1 \leq i \leq m)$$

where f_i takes n_i arguments and the terms T_i have free variables included in $\{X_1, \ldots, X_{n_i}\}$ and take a form given by

$$t ::= \underline{n} \mid X_i \mid -(\underline{n}, \underline{m}) \mid +(\underline{n}, \underline{m}) \mid f_i(t, \ldots, t) \mid \textbf{if } t \textbf{ then } t \textbf{ else } t \mid t \textbf{ or } t$$

(Here $\underline{n}, \underline{m}$ are numerals and $X_1, X_2 \ldots$ are variables).

(a) Write down a set of reduction rules $\rightarrow$ for closed terms t of the form above. These should include t_1 **or** $t_2 \rightarrow t_1$ and t_1 **or** $t_2 \rightarrow t_2$. Represent true by $\underline{0}$ and false by all other numerals. The reduction rules should reflect this and that function evaluation is call-by-name.

Because of non-deterministic choice the relation $\rightarrow^*$ forms a tree from an initial closed term t. Write $t \uparrow$ if any branch fails to reduce to a numeral. For a closed term t

$$eval(t) =_{\text{def}} \{n \mid t \rightarrow^* n\} \cup \{\bot \mid t \uparrow\}$$

gives an operational semantics.

(b) To give a denotational semantics use the powerdomain of integers,

$$P[N_\bot] = \{A \subseteq N_\bot \mid A \neq \emptyset \text{ and } (\bot \notin A \Rightarrow A \text{ is finite})\}$$

ordered by $A \sqsubseteq_{EM} B$ iff $\forall a \in A. \exists b \in B. a \sqsubseteq b$ and $\forall b \in B. \exists a \in A. a \sqsubseteq b$. Show that $P[N_\bot]$ is a cpo.

Now take the environment as $x = (x_0, x_1, \ldots, x_i, \ldots)$ in $P[N_\bot]^\infty$ and the function environment as $\emptyset = (\emptyset_0, \ldots, \emptyset_m)$ in $\prod_{0 \leq i \leq m}[P[N_\bot]^n i \rightarrow P[N_\bot]]$. Define the denotational function M by:

$M[\![X_i]\!]\emptyset = x_i$
$M[\![\underline{n}]\!]\emptyset x = \{n\}$
$M[\![f_i(t_1,\ldots,t_{n_i})]\!]\emptyset x = \emptyset_i(M[\![t_1]\!]\emptyset x,\ldots,M[\![t_{n_i}]\!]\emptyset x)$
$M[\![t_1 \textbf{ or } t_2]\!]\emptyset x = M[\![t_1]\!]\emptyset x \cup M[\![t_2]\!]\emptyset x$
$M[\![op(t_1,t_2)]\!]\emptyset x = \{op(v,w) \mid v \in M[\![t_1]\!]\emptyset x \text{ and } w \in M[\![t_2]\!]\emptyset x\}$
$M[\![\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3]\!]\emptyset x = \bigcup\{cond(y, M[\![t_2]\!]\emptyset x, M[\![t_3]\!]\emptyset x) \mid y \in M[\![t_1]\!]\emptyset x\}$

where op is the strict extension of $+$ or $-$ and $cond\colon N_\perp \times P[N_\perp]^2 \to P[N_\perp]$ is defined by

$$\begin{aligned} cond(x,Y,Z) &= \{\perp\} \text{ if } x = \perp \\ &= Y \text{ if } x = 0 \\ &= Z \text{ if } x \neq \perp, 0. \end{aligned}$$

You may assume that $M[\![t]\!]$ is continuous.

Define $\Gamma_i(\emptyset) =_{def} \lambda x_1,\ldots,x_{n_i}.M[\![T_i]\!]\emptyset[x_1,\ldots,x_{n_i};x]$ and $\Gamma(\emptyset) = (\Gamma_1(\emptyset),\ldots\Gamma_m(\emptyset))$.

(c) Show for t a closed term that $M[\![t]\!]fix(\Gamma)x = eval(t)$.
Hint. Use labelled terms to characterise $M[\![t]\!]\Gamma^n(\perp)x$ in an operational way. The proof follows the lines of [GW2] very closely. Use any results of the notes without proof provided you state them clearly.

(d) How would you modify the denotational semantics for call-by-value evaluation of the functions f_i? (Do not spend very long on this part unless for bonus points; some informal justification of your guess will do.)

References

[GW1], [GW2] Glynn Winskel, course notes.

Operational Semantics (1979)

A very simple non-deterministic programming language E is given by the following syntax, where A is a set of basic commands and NIL stands for termination:

i. NIL is a program
ii. if e is a program, so is $a\colon e$
iii. if d and e are programs, so is $(d + e)$

Example: $(a_1\colon(a_2\colon \text{NIL} + a_2\colon a_3\colon \text{NIL}) + a_1\colon a_2\colon \text{NIL})$ is a program. (Notice that there are no tests and no iteration; '+' stands for choice).

The problem is to investigate the operational semantics of E, by considering what sequences of basic commands can be executed by a program. We can define a production relation $\xrightarrow{a}$ over E, for each $a \in A$, by the following axiom and rules:

$$a\colon e \xrightarrow{a} e \qquad \frac{d \xrightarrow{a} d'}{(d+e) \xrightarrow{a} d'} \qquad \frac{e \xrightarrow{a} e'}{(d+e) \xrightarrow{a} e'}$$

Here are two possible ways of defining operational equivalence of programs:

i. Extend $\xrightarrow{a}$ to $\xrightarrow{s}$ for $s \in A^*$ by $e \xrightarrow{a_1 \ldots a_n} e'$ iff $e \xrightarrow{a_1} \ldots \xrightarrow{a_n} e'$ then define:

$$d \; eq \; e \text{ iff } \forall s \in A^*. \begin{cases} d \xrightarrow{s} d' \Rightarrow \exists e'. e \xrightarrow{s} e' \\ and \quad e \xrightarrow{s} e' \Rightarrow \exists d'. d \xrightarrow{s} d' \end{cases}$$

ii. Define a sequence $EQ_0, EQ_1, \ldots$ of equivalence relations over E by taking $EQ_0 = E \times E$, and

$$d \; EQ_{n+1} \; e \text{ iff } \forall a \in A. \begin{cases} d \xrightarrow{a} d' \Rightarrow \exists e'. (e \xrightarrow{a} e' \wedge d' \; EQ_n \; e') \\ and \quad e \xrightarrow{a} e' \Rightarrow \exists d'. (d \xrightarrow{a} d' \wedge d' \; EQ_n \; e') \end{cases}$$

then define: $d \; EQ \; e$ iff $\forall n. d \; EQ_n \; e$.

(a) Prove that both eq and EQ are congruences, i.e. (for eq) $d \; eq \; e \Rightarrow a{:}d \; eq \; a{:}e$ and $d \; eq \; d', e \; eq \; e' \Rightarrow (d + e) \; eq \; (d' + e')$.

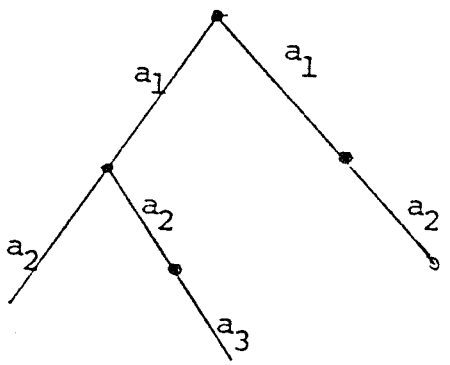
(b) Are eq and EQ the same relation? If so, prove it; if not, give and justify a counter-example.

(c) Prove that $(d + e) EQ (e + d)$, and find two or three other laws satisfied by EQ or eq or both.

(d) A preorder (reflexive transitive relation) $\leq$ over E can be defined analogously to EQ: take $\leq_0 = E \times E$, and $d \leq_{n+1} e$ iff $\forall a \in A. d \xrightarrow{a} d' \Rightarrow \exists e'. (e \xrightarrow{a} e' \wedge d' \leq_n e')$. Then define: $d \leq e$ iff $\forall n. d \leq_n e$. Is it then true that $d \; EQ \; e$ iff ($d \leq e$ and $e \leq d$)? Prove this, or give and justify a counter-example.

Hint

It helps to think of programs as finite trees, with arcs labelled by members of A — e.g. for the example,



Copyright © 1988, Laboratory for Foundations of Computer Science, University of Edinburgh. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.