Métodos Formais de Programação II

4.º Ano da LMCC (7008N2)
Ano Lectivo de 1999/2000

Exame (época especial) — 7 de Dezembro 2000
17h00
Sala 1319

PROVA SEM CONSULTA (3 $horas$)

GRUPO I

Questão 1 Identifique qual das seguintes leis é válida e explique por que razão as outras duas o não são:

$$(B \times C)^\star \cong B^\star \times C^\star \tag{1}$$
$$(B \times C)^\star \leq B^\star \times C^\star \tag{2}$$
$$C^\star \times B^\star \leq (B \times C)^\star \tag{3}$$

Sugestão: apoie a sua resposta no ficheiro `setcat.cam` listado no anexo desta prova.

Questão 2 Faça $g = id$ na lei (24). Que funções s, r e f obtém? Justifique formalmente.

Questão 3 Considere o seguinte modelo formal (muito!!) simplificado para uma base de dados de apuramento eleitoral:

$$Bd = Freg \rightharpoonup Inscritos \times Apuramento \tag{4}$$
$$Inscritos = I\!N \tag{5}$$
$$Apuramento = Partido \rightharpoonup I\!N \tag{6}$$

em que $Inscritos$ e $Partido$ são representados por 'strings'. Mostra-se a seguir um fragmento (idealizado) dessa base de dados, numa noite de eleições, tal como apareceria animado em CAMILA:

```
[ "S.Victor-Braga"->  < 1150, [ "CDS/PP"-> 100, "PS"-> 400, "PSD"-> 350, "CDU"-> 150 ]>,
  "Ucha-Barcelos"->   < 1030, [ "CDS/PP"-> 190, "PS"-> 590, "PSD"-> 290, "CDU"-> 110 ]>,
  "Paranhos-Porto"->  < 1270, [ "CDS/PP"-> 120, "PS"-> 450, "PSD"-> 370, "CDU"-> 110 ]>,
  "Vila Verde"->      < 1230, []>,
  "V.N.Foz-Coa"->     < 1010, [ "CDS/PP"-> 190, "PS"-> 330, "PSD"-> 390, "CDU"-> 70 ]>,
  "Anais-P.Lima"->    < 1450, []>
]
```

(Note-se que os dados de duas freguesias ainda não chegaram à base de dados.)

Solicitado a especificar a operação que realiza o apuramento total nacional, um técnico escreveu a seguinte especificação funcional,

$$Bd \xrightarrow{f_1} (a) \xrightarrow{f_2} (b) \xrightarrow{f_3} (c) \xrightarrow{f_4} (Partidos \rightharpoonup I\!N)$$

tendo desafiado os colegas a identificarem as funções f_1, f_2, f_3 e f_4, mostrando-lhes apenas o resultado da sua animação, tal como se segue:

Formato (a), após f_1:

```
[ "S.Victor-Braga"->  [ "CDS/PP"-> 100, "PS"-> 400, "PSD"-> 350, "CDU"-> 150 ],
  "Anais-P.Lima"->    [],
  "Vila Verde"->      [],
  "Paranhos-Porto"->  [ "CDS/PP"-> 120, "PS"-> 450, "PSD"-> 370, "CDU"-> 110 ],
  "V.N.Foz-Coa"->     [ "CDS/PP"-> 190, "PS"-> 330, "PSD"-> 390, "CDU"-> 70 ],
  "Ucha-Barcelos"->[ "CDS/PP"-> 190, "PS"-> 590, "PSD"-> 290, "CDU"-> 110 ]
]
```

Formato (b), após f_2:

```
[ < "CDS/PP","S.Victor-Braga"> -> 100,
  < "PS","S.Victor-Braga"> -> 400,
  < "PSD","S.Victor-Braga"> -> 350,
  < "CDU","S.Victor-Braga"> -> 150,
  < "CDS/PP","Paranhos-Porto"> -> 120,
  < "PS","Paranhos-Porto"> -> 450,
  < "PSD","Paranhos-Porto"> -> 370,
  < "CDU","Paranhos-Porto"> -> 110,
  < "CDS/PP","V.N.Foz-Coa"> -> 190,
  < "PS","V.N.Foz-Coa"> -> 330,
  < "PSD","V.N.Foz-Coa"> -> 390,
  < "CDU","V.N.Foz-Coa"> -> 70,
  < "CDS/PP","Ucha-Barcelos"> -> 190,
  < "PS","Ucha-Barcelos"> -> 590,
  < "PSD","Ucha-Barcelos"> -> 290,
  < "CDU","Ucha-Barcelos"> -> 11
]
```

Formato (c), após f_3:

```
[ "CDS/PP"->[ "S.Victor-Braga"-> 100, "Paranhos-Porto"-> 120, "V.N.Foz-Coa"-> 190, "Ucha-Barcelos"-
> 190 ],
  "PS"->[ "S.Victor-Braga"-> 400, "Paranhos-Porto"-> 450, "V.N.Foz-Coa"-> 330, "Ucha-Barcelos"-> 590 ],
  "PSD"->[ "S.Victor-Braga"-> 350, "Paranhos-Porto"-> 370, "V.N.Foz-Coa"-> 390, "Ucha-Barcelos"-> 290 ],
  "CDU"->[ "S.Victor-Braga"-> 150, "Paranhos-Porto"-> 110, "V.N.Foz-Coa"-> 70, "Ucha-Barcelos"-> 110 ]
]
```

Formato final $(Partidos \rightharpoonup I\!N)$, após f_4:

```
[ "CDS/PP"-> 600, "CDU"-> 440, "PS"-> 1770, "PSD"-> 1400 ];
```

1. Defina as funções f_1, f_2, f_3 e f_4, justificando a sua resposta.
2. Especifique o invariante (relevante!) sobre Bd que garante que em nenhuma freguesia o total dos votos apurados excede o número de eleitores inscritos.
3. O modelo Bd carece de uma transformação para se poder implementar em ambiente relacional. Qual? Quantas tabelas obtém? Justifique.

Questão 4 Atente na seguinte construção presente na linguagem de especificação CAMILA:

```
(*->f)(x) = [ k -> f(x[k]) | k <- dom(x) ];
```

Caracterize o tipo da construção `*->` através de um diagrama e identifique quais das seguintes igualdades (também escritas em sintaxe CAMILA) se verificam e porquê:

$$\texttt{(*->(lambda(x).x))(x) == x} \tag{7}$$

$$\texttt{(*->(lambda(x).f(g(x))))(x) == (*->(lambda(x).f(x)))((*->(lambda(x).g(x)))(x))} \tag{8}$$

$$\texttt{(*->(const(c)))(x) == const(c)} \tag{9}$$

GRUPO II

Responda às questões deste grupo após ter analisado o seguinte modelo em CAMILA de um sistema hierárquico de ficheiros (tipo UNIX ou MS/DOS):

```
FS :: D: Id -> Node;      /* FS  means file system                 */
Node = File | FS;         /* a Node is either a file or a directory */
Id = STR;                 /* a Node identifier is a string          */
File :: F: STR-seq;       /* a file is a sequence of characters     */
```

Questão 5 Complete as reticências no seguinte diagrama descritivo dos catamorfismos do tipo FS:

$$\begin{tikzcd} FS \arrow[r, bend left, "\cdots"] \arrow[r, phantom, "\cong"] \arrow[d, "(\!|g|\!)"'] & Id \rightharpoonup (File + FS) \arrow[l, bend left, "\cdots"] \arrow[d, "\ldots(\!|g|\!)\ldots"] \\ B & \ldots B \ldots \arrow[l, "g"'] \end{tikzcd} \tag{10}$$

e caracterize g para a situação em que:

1. $(\!|g|\!)$ é a função que conta o número de ficheiros presentes num sistema de ficheiros;
2. $(\!|g|\!)$ é a função que indica qual o tamanho do maior ficheiro presente num dado sistema de ficheiros (**NB:** pode e deve recorrer à função $strlen$ que calcula o comprimento de um 'string');

Questão 6 Especifique a função $find : Id \times FS \longrightarrow Bool$ que $find(i, fs)$ testa se um dado $i \in Id$ é (ou não) usado como identificador em fs.

Questão 7 Caracterize o *método de indução estrutural* associado ao tipo FS, isto é, complete as reticências em:

> *Para provar o facto*
> $$\forall x \in FS : \phi\, x$$
> *proceder da forma seguinte:*
> 1. Caso de base*: para* ....
>   *provar* ....
> 2. Caso indutivo*: seja* ....
>   (a) Hipótese de indução*: assumir* ....
>   (b) Salto indutivo*: provar* ....
>
> $\square$

Anexo

INDUÇÃO

Universal-cata	$k = (\!|\alpha|\!) \quad \Leftrightarrow \quad k \cdot in = \alpha \cdot (\mathsf{F}\, k)$	(11)
Cancelamento-cata	$(\!|\alpha|\!) \cdot in = \alpha \cdot \mathsf{F}\, (\!|\alpha|\!)$	(12)
Reflexão-cata	$(\!|in|\!) = id_{\mathsf{T}}$	(13)
Fusão-cata	$f \cdot (\!|\alpha|\!) = (\!|\beta|\!) \quad \text{if} \quad f \cdot \alpha = \beta \cdot (\mathsf{F}\, f)$	(14)
Absorção-cata	$(\!|g|\!) \cdot \mathsf{T}\, f \quad = \quad (\!|g \cdot \mathsf{B}(f, id)|\!)$	(15)

FUNCTORES

Functor-F	$\mathsf{F}(g \cdot h) \quad = \quad (\mathsf{F}\, g) \cdot (\mathsf{F}\, h)$	(16)
Functor-id-F	$\mathsf{F}\, id_A \quad = \quad id_{(\mathsf{F}\, A)}$	(17)
"Teorema grátis" de g	$(\mathsf{G}\, f) \cdot g \quad = \quad g \cdot (\mathsf{F}\, f)$	(18)

ISOMORFISMOS BÁSICOS

$$A \rightharpoonup B \quad \cong \quad (B+1)^A \tag{19}$$
$$\mathcal{P}A \quad \cong \quad A \rightharpoonup 1 \tag{20}$$

LEIS DE REFINAMENTO FORMAL DE DADOS

$$A \rightharpoonup (B \times C) \quad \leq \quad (A \rightharpoonup B) \times (A \rightharpoonup C) \tag{21}$$
$$A \rightharpoonup D \times (B \rightharpoonup C) \quad \leq \quad (A \rightharpoonup D) \times ((A \times B) \rightharpoonup C) \tag{22}$$
$$A \rightharpoonup (B \rightharpoonup C) \quad \leq \quad \mathcal{P}A \times ((A \times B) \rightharpoonup C) \tag{23}$$

$$\mathsf{U} \cong \mathsf{G}\,\mathsf{U} \wedge \mathsf{T} \cong \mathsf{F}\,\mathsf{T} \wedge \begin{tikzcd} \mathsf{F}\, X \arrow[rr, bend left, "s"] \arrow[rr, phantom, "\leq"] & & \mathsf{G}\, X \arrow[ll, bend left, "g"] \end{tikzcd} \Rightarrow \begin{tikzcd} \mathsf{T} \arrow[rr, bend left, "r = (\!|in_{\mathsf{U}} \cdot s|\!)_{\mathsf{F}}"] \arrow[rr, phantom, "\leq"] & & \mathsf{U} \arrow[ll, bend left, "f = (\!|in_{\mathsf{T}} \cdot g|\!)_{\mathsf{G}}"] \end{tikzcd} \tag{24}$$

FICHEIRO SETCAT.CAM

```
;-------------------------------------------------------------------------------
; setcat.cam = a simplistic model of set theory (based on category Set)
;              in CAMILA
;-------------------------------------------------------------------------------
; Creation date : 1999.03.31
; Last update :   1999.04.29
; (c) UM - M.F.P. I/II - 1998/99
;-------------------------------------------------------------------------------

;--- (a) Basics ----------------------------------------------------------------

; una is a generic function which converts a binary function into a unary
; function on pairs.
  una(f)=lambda(p).f(p1(p),p2(p));

; bin does the opposite of 'una'
  bin(f)=lambda(a,b).f(<a,b>);

; flip permutes the order of arguments of a binary function
  flip(f)=lambda(b,a).f(a,b);

; perm outputs a permutation of finite set s
  perm(s)= < a | a <- s >;

; get : B --> A --> (A -> B) --> B
; get totalizes a partial function (f) with a default value (u)
  get(u)=lambda(a).lambda(f).if a in dom(f) then f[a] else u;

eqk(k)= lambda(x).x==k;

strgth(x,b)={<x,y>|y<-b};

; fdp : (A <-> B) --> 2
  fdp <- (lambda(s).s=={} \/ s=={1}) _o_ ran _o_ (*->card) _o_ collect;

forall(s)=and-orio(true,s);

pconst(x)=lambda(b).mkf(swap-set(strgth(x,b)));

inseq(n) = perm(inseg(n));

uplus <- una(plus);

PLUS(s) = plus-orio([],s);

MAX(s) = max-orio(0,s);

selUp(s,f)=lambda(x). x + ((*->f)(x/s));

pred(n)=n.-1;

succ(n)=n.+1;

;--- (b) CCC -------------------------------------------------------------------

; id : A --> A
  id(x)=x;

; comp : (C <-- B) x (B <-- A) --> (C <-- A)
  comp(f,g)= lambda(x).f(g(x));

; const : C --> (A --> C)
  const(c)= lambda(x).c;

; excl : A --> 1
  excl <- const(NIL);

; curry : (A x B --> C) --> A --> (B -->C)
  curry(f) = lambda(a).lambda(b).'f(a,b);

; i1 : (A + B) <-- A
  i1(a) = <1,a>;

; i2 : (A + B) <-- B
  i2(b) = <2,b>;

; grd : (A --> 2) --> A --> (A + A)
  grd(p) = lambda(x). if p(x) then i1(x) else i2(x);

; is1 : ( A + B ) --> 2
  is1(x) = p1(x)==1;

; is2 : ( A + B ) --> 2
  is2(x) = p1(x)==2;

; split : ( A --> B x C )  <--  ( A --> B ) x ( A --> C )
  split(f,g)= lambda(x).<'f(x),'g(x)>;

; either : (A --> C) x (B --> C) --> ((A + B) --> C)
  either(f,g)=lambda(x).let (t=p1(x),y=p2(x)) in if t==1 then 'f(y) else 'g(y);

; prod : (A --> B) x (C --> D) --> ((A x C) --> (B x D))
  prod(f,g) = split(comp(f,p1),comp(g,p2));

; diag : (A --> B)  --> ((A x A) --> (B x B))
; diag is the diagonal functor construct
  diag(f) = prod(f,f);

; sum : (A --> B) x (C --> D) --> ((A + C) --> (B + D))
  sum(f,g) = either(comp(i1,f),comp(i2,g));

; swap : (A x B) --> (B x A)
  swap <- split(p2,p1);

; assocr : (A x B) x C --> A x (B x C)
  assocr <- split(comp(p1,p1),prod(p2,id));

; assocl : A x (B x C) --> (A x B) x C
  assocl <- split(prod(id,p1),comp(p2,p2));

; pwnil : A --> (A x 1)                /* pwnil = pair with NIL */
  pwnil <- split(id,excl);

; coswap : (A + B) --> (B + A)
  coswap <- either(i2,i1);
```

```
; coassocr : ((A + B) + C) --> (A + (B + C))
  coassocr <- either(sum(id,i1),comp(i2,i2));

; undistr : A x ( B + C ) <-- ( ( A x B ) + ( A x C ) )
  undistr <- either(prod(id,i1),prod(id,i2));

; split0 : (A --> 1) <-- 1
  split0 <- lambda(x). const(NIL);

; uneither0 : 1 <-- (0 --> A)
  uneither0 <- lambda(x). NIL;

; untab : ( A -> B ) --> (A -> (B + 1))
  untab(sigma)=comp(sum(curry(ap)(sigma),excl),grd(lambda(a).a in dom(sigma)));

; cf : ( A --> 2 ) <-- A-set            /* cf = characteristic function */
  cf(s) = lambda(a).a in s;

; pair2exp : ( 2 --> A ) <-- A x A
; an isomorphism
  pair2exp(p) = lambda(b). if b then p2(p) else p1(p);

; nestr : A × B × C  -->  A × (B × C)
  nestr <- split(p1,split(p2,p3));

; nestl : A × B × C  -->  (A × B) × C
  nestl <- split(split(p1,p2),p3);

; flatr : A × (B × C)  -->  A × B × C
  flatr(t)=<p1(t),p1(p2(t)),p2(p2(t))>;

; flatl : (A × B) × C  -->  A × B × C
  flatl(t)=<p1(p1(t)),p2(p1(t)),p2(t)>;

;--- (c) Partial maps ----------------------------------------------------------

; set2fm : ( A -> 1 ) <-- A-set
; dom is the inverse of set2fm
  set2fm(s) = [ x -> NIL | x <- s ];

; peither : ( ( A + B ) -> C )  <--  ( A -> C ) x ( B -> C )
; This is the "partial version" of either
  peither(f1,f2) = (i1->*)(f1) + (i2->*)(f2);

; unpeither : ( ( A + B ) -> C )  -->  ( A -> C ) x ( B -> C )
; The inverse of (unary) unpeither
  unpeither(f) = let (a=[ p2(x) -> f[x] | x <- dom(f): p1(x) == 1 ],
                      b=[ p2(x) -> f[x] | x <- dom(f): p1(x) == 2 ])
                 in < a , b >;

; join : ( A -> B x C )  <--  ( A -> B ) x ( A -> C )
; This is the "partial version" of split
  join(f1,f2) = [ a -> < f1[a],f2[a]> | a <- dom(f1) * dom(f2) ];

join_ <- una(join);

; unjoin : ( A -> B x C )  -->  ( A -> B ) x ( A -> C )
; The right-inverse of join_, ie join_ . unjoin = id.
  unjoin <- split(*->p1,*->p2);

; pcurry : (A x B -> C) --> A -> (B->C)
; pcurry is the "partial version" of curry.
; It is not epic and therefore not iso
  pcurry(f) =
        let (x=dom(f), y=p1-set(x))
        in [ a -> [ p2(p) -> f[p] | p <- x : p1(p)==a ] | a <- y ];

; unpcurry : (A x B -> C) <-- A -> ( B -> C )
; unpcurry is the "partial version" of uncurry.
; It is not monic and therefore not iso
  unpcurry(f) =
       PLUS({ let (g=f[a]) in [ <a,b>  -> g[b] | b <- dom(g) ] | a <- dom(f) });

; collect : (A x B)-set --> A -> B-set
; collect can be calculated from pcurry and set2fm
  collect(r) = [ a -> { p2(q) | q <- r: a==p1(q) } | a <- { p1(p) | p <- r } ];

; discollect : (A x B)-set <-- A -> B-set
; discollect can be calculated from unpcurry and set2fm
  discollect(f) = UNION({ { <a,b> | b <- f[a] } | a <- dom(f) });

; njoin : ((A -> D) x (A x B -> C)) --> (A -> D x (B->C))
; njoin (read: 'nested join') is epic but not mono
  njoin <- comp(join_,split(p1,comp(uplus,prod((*->const([])),pcurry))));

; unnjoin : ((A -> D) x (A x B -> C)) <-- (A -> D x (B->C))
; njoin . unnjoin = id, ie. unnjoin is njoin's right-inverse.
  unnjoin <- comp(prod(id,unpcurry),unjoin);

; cojoin : ( A -> (B + C) )  <--  ( A -> B ) x ( A -> C )
  cojoin(f,g) = let (h=f\dom(g), i=g\dom(f)) in  (*->i1)(h) + (*->i2)(i);

; uncojoin : ( A -> (B + C) )  -->  ( A -> B ) x ( A -> C )
  uncojoin(f) = let (x={ a | a <- dom(f): is1(f[a]) },
                     y={ a | a <- dom(f): is2(f[a]) })
                in  <(*->p2)(f/x),(*->p2)(f/y)>;

; ffinv : (A -> B) --> (B -> A-set)
; ffinv(f) "reverses" f.
  ffinv(f) = [ b -> { a | a<- dom(f): f[a]==b } | b <- ran(f) ];

; ffswap : (A -> B-set) --> (B -> A-set)
; ffswap(f) "swaps" f.
  ffswap <- collect _o_ swap-set _o_ discollect;

; unffinv : (A -> B) <-- (B -> A-set)
; unffinv is ffinv's right inverse, that is ffinv . unffinv = id
  unffinv(f)= let (x={ [ a -> b | a <- f[b] ] | b <- dom(f) },
                   y={ f \ UNION(dom-set(x-{f})) | f <- x })
              in  PLUS([],y);

; mkf :  (A -> B)  <--  (A x B)-set
; This is partial function !
  mkf <- comp((*->the),collect); /* ie [ p1(t) -> p2(t) | t <- r ] */

; mkr :  (A -> B)  -->  (A x B)-set
```

```
  mkr(f) = { <a,f[a]> | a <- dom(f) };

;--- (d) Sequences ------------------------------------------------------------

; fm2seq : A-seq <-- (NAT -> A)
; Turns a numeric index of A-values into the implicit sequence
  fm2seq(f) =
         if f==[] then <> else let (m=MAX(dom(f))) in fm2seq(f\{m}) ^ < f[m] >;

; seq2fm : A-seq --> (NAT -> A)
; A sequence is represented by a numerical index keeping track of the original
; order. Right-inverse of fm2seq.
  seq2fm(l) = [ x -> l[x] | x <- inseg(length(l)) ];

; zip  : (B x C)-seq  <--  B-seq  x  C-seq
; This is the "seq version" of join
  zip(l,m) =
        let (n=min(length(l),length(m))) in < < l[i],m[i] > | i <- inseq(n) >;

; unzip  : (B x C)-seq  -->  B-seq  x  C-seq
; The right-inverse of una(zip), ie una(zip) . unzip = id.
  unzip  <- split(p1-seq,p2-seq);

;--- (e) X = 1 + A x X structure hylo ------------------------------------------

inF  <- either(const(<>),una(cons));

outF <- comp(sum(const(nil),split(hd,tl)),grd(eqk(<>)));

recF(f)=sum(id,prod(id,f));

cata(g) = lambda(x).(comp(g,comp(recF(cata(g)),outF)))(x);

ana(g) = lambda(x).(comp(inF,comp(recF(ana(g)),g)))(x);

hylo(h,g) = comp(cata(h),ana(g));

;--- (e.1) application to set-folding ------------------------------------------

; nsetget : A-set --> A x A-set
; nsetget is undefined for the empty set
  nsetget(s) = let (a=choice(s)) in <a,s-{a}>;

; gets : A-set --> 1 + A x A-set
; gets is the A-set standard coalgebra
  gets <- comp(sum(const(NIL),nsetget),grd(eqk({})));

; tot2 : (B x B --> B) x B --> (A x B)-set --> B
; tot2 totalizes a binary relation on the second projection according to
; monoid structure f: B x B -> B and u: B
  tot2(f,u)= hylo(either(const(u),comp(una(f),prod(p2,id))),gets);

;--- (e.2) application to Nat-folding ------------------------------------------

getN <- comp(sum(const(nil),split(id,pred)),grd(eqk(0)));

fac <- hylo(either(const(1),una(mul)),getN);
```