Tipuri de date

Mihai Budiu

mai 1995

Subiect: Tipurile de date în în Pascal

Cuvinte cheie: tip de date, operatori, funcţii, puternic tipizat, polimorfism, supraîncărcare

Cunoştinţe necesare: Pascal, teoria elementară a mulţimilor

Contents

1 Introducere

Avantajele folosirii unui limbaj *puternic tipizat* sunt apreciate de orice programator experimentat. Printre ele sunt:

- uşurează înţelegerea programului de către om;
- încurajează o gîndire şi formulare disciplinată a problemei;
- uşurează munca de optimizare a compilatorului;
- permit depistarea erorilor într-o fază timpurie.

Un limbaj se numeşte "puternic tipizat" dacă în faza de execuţie a unui program nu pot apărea erori cauzate de tipuri incorecte ale expresiilor. Acest articol îşi propune să explice într-un cadru cît se poate de riguros o colecţie foarte simplă de fapte care descrie semnificaţia noţiunii de "tip de date".

2 Ce este un tip de date

Există mai multe definiţii posibile pentru tipurile de date, din mai multe perspective. O privire globală defineşte simultan atît tipurile de date cît şi operaţiile care se pot face cu ele, pentru că acestea sunt de fapt cele două faţete ale unei aceleiaşi monezi: nu pot exista una fără cealaltă. Noi vom trata cele două subiecte pe rînd şi în mod inegal, acordînd o oarecare prioritate tipurilor de date.

Astfel, în cea mai simplă accepţiune posibilă, un tip de date nu este altceva decît *o mulţime de valori*.

Ce este o valoare? Nu are nici o importanţă! Este un nume special dat elementelor care aparţin unui tip. Termenul "aparţin" este justificat, pentru că, reţineţi, tipul este *o mulţime*, în cel mai precis sens matematic.

Exemple? în Pascal tipul `boolean` este o mulţime cu două elemente. Aceste două elemente se notează în Pascal cu `false` şi `true`. `False` şi `true` se numesc de aceea *valori booleene*. Tipul `integer` este tot o mulţime care include, printre altele, nişte elemente (valori) care în Pascal se notează cu `-2, -1, 0, 1, 2, 3` etc. Tipul `integer` este un tip interesant, pentru că îşi propune să mimeze mulţimea matematica **Z** a numerelor întregi, dar nu reuşeşte prea bine, pentru că mulţimea `integer` este finită, iar **Z** nu! Care sunt limitele mulţimii `integer` nu se specifică. În dialectul Turbo Pascal (cel mai răspîndit pe PC-uri) `integer` are 65536 de elemente, de la cel notat `-32768` la cel notat `32767`. Elementele tipului `integer` sunt numite valori întregi (sau mai precis *integer values*).

Dacă aţi citit textul de mai sus cu atenţie aţi observat că ne-am ferit să spunem ca `false` este un element (valoare) boolean. Am spus `false` este o notaţie (o reprezentare) a unui element boolean! De aici încolo nu vom mai fi atît de scrupuloşi; dealtfel cele două noţiuni (element şi reprezentare) se pot adesea interschimba fără ambiguitate.

Vom zice de aceea:

$$boolean = \{false, true\}$$

unde acoladele sunt notaţia uzuală pentru mulţime.

3 Tipuri enumerate

Pascal pune la dispoziţie programatorului mai multe mecanisme prin care el să-şi construiască noi tipuri. Cel mai simplu dintre acestea este *enumerarea*. Enumerarea construieşte un nou tip cu un număr finit de elemente descriind între paranteze reprezentarea canonică a fiecărui element.

Exemplu (tipic) :

```
(rosu, verde, albastru, galben, mov, negru, alb)
```

este descrierea unui nou tip care are 7 elemente cu reprezentările `rosu`, `verde`, etc. Putem da acestui tip un nume folosind directiva `type` din Pascal.

```
type culoare = (rosu, verde, albastru, galben, mov, negru, alb);
```

Din punct de vedere matematic am putea scrie

$$culoare = \{rosu, verde, albastru, galben, mov, negru, alb\}$$

Tipul boolean din Pascal ar putea fi definit el însuşi astfel:

```
type boolean = (false, true);
```

De ce ne este oferit însă de-a gata vom afla din următoarea secţiune şi din cele consacrate operatorilor.

4 Tipuri fundamentale

Limbajul Pascal (standard) conţine patru tipuri gata construite. Pe acestea le numim *tipuri fundamentale*. Ele sunt:

```
real, integer, char, boolean
```

Pentru `boolean` am văzut

$$boolean = \{false, true\}$$

Pentru `integer` am putea scrie ceva de genul

$$integer = \{n\epsilon\mathbf{Z} \mid limita_{jos} \leq n < limita_{sus}\}$$

$limita_{jos}$ şi $limita_{sus}$ depind de dialectul de Pascal folosit. Pentru Turbo ele sunt (după cum am zis) `-32768`, respectiv `32768`.

Pentru `char` lucrurile sunt grele pentru că există unele caractere care *nu* au o reprezentare evidentă! Tot ce putem spune este că această mulţime include mulţimea

{`'a', 'b', ...'A', 'B', ..., '0', ..., '!', ...`}

a caracterelor care pot fi tipărite.

Iar pentru `real` lucrurile sunt şi mai încîlcite, pentru că, deşi `real` se doreşte o imagine a mulţimii $\mathbf{R}$ a numerelor reale, în realitate `real` este finită şi are elementele distribuite într-un mod foarte ciudat pe axa reală (nici măcar două elemente ale ei nu sunt echidistante)! Aceasta mulţime este subiectul *analizei numerice*, care studiază cum putem face calcule cînd putem manipula numai aproximări ale valorilor.

5 Derivarea tipurilor

Pascal (şi nu numai el) pune la dispoziţie nişte puternice mecanisme prin care putem construi noi tipuri plecînd de la cele existente. (Prin enumerare putem construi tipuri plecînd de la nimic.) Derivarea dă naştere unei noi mulţimi de elemente combinînd elementele unor tipuri deja existente.

Există o interesantă clasă de tipuri numite *tipuri scalare*. Tipurile `boolean, char, integer` şi toate tipurile construite prin enumerare sunt scalare. Aceste tipuri au întotdeauna un număr finit de elemente şi după cum vom vedea mai tîrziu, există (datorită operatorilor de comparaţie) o relaţie de ordine totală pe ele (adică dintre două elemente unul e mai mare şi celălalt mai mic)[1].

5.1 Subdomeniu

Există o operaţie prin care dintr-un tip *scalar* putem extrage o submulţime formată din elemente consecutive. Ea devine un nou tip care se va numi un *subdomeniu* ("subrange" în engleză) al celui iniţial. Pentru că pe tipurile scalare există o ordine, submulţimea se indică prin capetele ei. Exemplu:

```
1 .. 100
```

este un tip subdomeniu al `integer`, avînd 100 de elemente.

Iată un tip subdomeniu al tipului culoare definit mai sus, căruia îi dăm şi un nume:

```
type CulPrimara = rosu .. albastru;
```

Tipul numit CulPrimara are 3 elemente:

$$CulPrimara = \{rosu, verde, albastru\}.$$

Tipurile subdomeniu sunt toate scalare.

5.2 Articol (record)

Dacă avem la dispoziţie tipurile $T_1, T_2, \ldots T_n$ putem forma un nou tip cu construcţia:

```
record
  c1 : T1;
  c2 : T2;
  ....
  cn : Tn;
end
```

Ce mulţime reprezintă noul tip? O valoare de tip record subsumează n valori, fiecare din tipul T_i respectiv. Cu alte cuvinte putem asimila un record cu un tuplu (o mulţime ordonată) cu n elemente $(v_1, v_2, \ldots, v_n)$, unde $v_i \epsilon T_i \,|\, i \epsilon \{1 \ldots n\}$.

Să dăm un nume tipului de mai sus:

[1] Raportul Pascal marchează şi tipul `real` ca fiind scalar, dar din motive de omogenitate noi nu îl vom considera astfel, pentru că el nu se comportă în nici o privinţa ca un tip scalar

```
type R = record
   c1 : T1;
   c2 : T2;
   ...
   cn : Tn
end;
```

atunci

$$R = \{(v_1, v_2, \ldots, v_n) \mid v_i \epsilon T_i,\ i = 1 \ldots n\}.$$

sau altfel scris

$$R = T_1 \times T_2 \times \ldots \times T_n = \prod_{i=1}^{n} T_i$$

unde $\times$ denotă operaţia de produs cartezian.

Un tip construit cu `record` nu este scalar.

Notă: pentru deplina rigoare ar trebui să includem în mulţimea R şi numele cîmpurilor $c_1, c_2, ..., c_n$. O definiţie completă ar fi: produsul cartezian al tipurilor componente, avînd fiecare componentă etichetată cu numele corespunzător. Etichetarea se poate defini şi ea perfect riguros, dar o lăsăm pe seama cititorului.

5.3 Vector (matrice)

Dacă avem un tip oarecare B şi un tip scalar S (cu excepţia lui `integer`) putem forma din ele un nou tip cu operaţia (dînd şi nume tipului):

```
type A = array [S] of B;
```

Nu este prea complicat de văzut că asta înseamnă

$$A = B \times B \times \ldots \times B = B^{|S|}$$

unde produsul este luat de cardinal din S (numărul de elemente al lui S) ori. Aceasta pentru că un `array[S] of B` este o colecţie de atîtea valori de tipul B cîte valori conţine tipul S.

Un tip construit cu `array` nu este scalar.

Notă: Ca mai sus, fiecare componentă a unui element din produsul B^n este etichetată cu o valoare corespunzătoare din S.

5.4 Mulţime

Dacă avem un tip scalar S putem forma tipul (dacă S nu are prea multe elemente):

```
set of S
```

Ce valori are acest nou tip? Un element al tipului `set of S` este o mulţime formată cu elemente din S. Deci dacă avem declaraţia:

```
type R = set of S;
```

atunci

$$R = \{M \mid M \subset S\} = \mathcal{P}(S)$$

adică mulţimea părţilor lui S.

Un tip construit cu `set` nu este scalar.

5.5 Fişier

Fişierul este un tip interesant. Este prima metodă de a construi un tip care — teoretic vorbind — este infinit! (Toate tipurile de pînă acum sunt finite pentru că sunt construite prin combinaţii finite ale unor mulţimi finite). Bine-nţeles că practic şi acest tip este finit, dar cu o dimensiune maximă neprecizată.

Fiind dat un tip T (care nu e fişier la rîndul său) ce face operaţia:

```
file of T
```

?

Un fişier este asemănator unui `array`, dar are o lungime neprecizată. El conţine la un moment dat 0 sau 1 sau 2, ... valori de tipul T. Deci dacă F e definit

```
type F = file of T;
```

atunci ca mulţime

$$F = \bigcup_n \{T^n \mid n \epsilon \mathbf{N}\} = T^*$$

unde $\mathbf{N}$ este mulţimea numerelor naturale.

(O astfel de operaţie aplicată lui T (reuniunea tuturor puterilor) se numeşte operaţia lui Kleene, şi se notează cu T^*).

5.6 Record cu case

Ajungem la o operaţie mai puţin omogenă de combinare a tipurilor. Să considerăm directiva Pascal:

```
type RC = record
   c1 : T1;
   case c2 : T2 of
     v1 : ( c3 : T3 );
     v2 : ( c4 : T4 );
end
```

Un model aproximativ este dat de

$$RC \subset T1 \times T2 \times (T3 \cup T4)$$

pentru că o valoare de tip RC are primele două cîmpuri de tipurile T1, respectiv T2, şi un alt cîmp dintr-unul din tipurile T3 sau T4.

In realitate RC nu este atît de vastă, pentru că nu orice combinaţie din mulţimea indicată este în RC. Cînd cîmpul al doilea are valoarea v1, cîmpul al treilea nu poate fi de tipul T4! Deci corect este:

$$RC = T1 \times (\{v1\} \times T3 \cup \{v2\} \times T4)$$

{v1} este mulţimea formată din elementul v1 aparţinînd lui T2.

Un astfel de tip nu este scalar.

5.7 Pointer

Ultimul mod de derivare a tipurilor în Pascal este prin luarea referinţei la ele. Un tip

```
type PT = ^T;
```

are printre valorile sale o valoare specială `nil`, şi apoi valori care *arată* spre valori din tipul T. Tipul PT este deci izomorf (corespondent unu la unu) cu tipul T la care se adaugă un nou element:

$$PT = T \cup \{\mathbf{nil}\}.$$

Tipul pointer din Pascal este singurul tip care poate să includă în definiţia sa pe sine însuşi, ca în:

```
type
  PA = ^A;
  A = record
    i : I;
    n : PA;
  end;
```

Asta dă o interesantă "ecuaţie" pentru mulţimea PA:

$$PA = \{nil\} \cup A = \{nil\} \cup (I \times PA).$$

Nu este nimic contradictoriu în această formulă! Cele două mulţimi sunt într-adevăr egale, în sensul că fiecare element care se află în PA este fie **nil** fie se poate scrie sub forma (i, pa) unde $i \epsilon I$ şi $pa \epsilon PA$.

De fapt aceasta este definiţia (recursivă) a unei liste (aţi recunoscut probabil tipul PA ca fiind tipul celulei de bază care compune o listă simplu înlănţuită):

O listă (de tipul I) este sau

- lista vidă, sau
- o valoare de tipul I împreună cu o listă "mai mică" de acelaşi tip.

6 Numele tipurilor

Am folosit adesea în exemplele anterioare nume asociate tipurilor cu ajutorul directivei `type`. în construcţia:

```
type
 t = record
   a : array[1..10] of integer;
   c : char;
 end;
```

avem două parţi:

1. una care construieşte un nou tip

   ```
   record
     a: array[1..10] of integer;
     c: char;
   end
   ```

2. una care îi dă un nume acestei construcţii:

   ```
   type t = constructie;
   ```

Acestea sunt două operaţii distincte. Mai mult, în Pascal tipurile `t` şi `record a:array[1..10] of integer; c:char; end` sunt în realitate distincte deşi au aceeaşi structură (sunt formate prin produsele aceloraşi mulţimi: $integer^{10} \times char$.

Asta putem vedea dacă scriem:

```
var x, z: t;
    y: record array[1..10] of integer; c: char; end;
```

Acum `x` şi `y` au tipuri diferite. O operaţie de felul `x := y` nu este permisă! Putem face însă `x := z`.

Deci: două tipuri nu sunt *compatibile* chiar dacă au aceeaşi structură. Ele trebuie să aibă acelaşi *nume*, şi atunci de fapt sunt o aceeaşi mulţime.

7 Operatori

În realitate orice definiţie a unui tip ca mulţime trebuie să includă şi o mulţime de operaţii pe valorile sale, care iau naştere automat odată cu crearea sa. Ce este o operaţie? Aparent banală întrebarea nu are un răspuns evident. `div` este o operaţie între doi întregi. Ce înseamnă asta? Ea combină cele două valori întregi cărora li se aplică pentru a genera una nouă. Este deci o *funcţie* definită astfel:

$$div : integer \times integer \rightarrow integer$$

Ea asociază perechii (a, b) de numere întregi numărul “cîtul împărţirii lui a la b”.

`div` este o funcţie care nu se poate calcula pentru toate perechile de întregi. Dacă al doilea este 0 atunci operaţia nu are sens. Deci funcţia este *parţială*, adică nu este definită pentru toate valorile posibile ale argumentelor.

De fapt asta sunt toţi operatorii — funcţii din tipuri (sau produse carteziene de tipuri) în tipuri.

De ce sunt tipurile atît de legate de operatori? Pentru că crearea unui nou tip prin oricare din operaţiile de mai sus dă naştere în mod automat unor operatori care lucrează cu valori din acel tip. De exemplu definirea unui tip enumerare ca cel `culoare` de mai sus dă naştere următorilor operatori:

operatori	domeniu de definiţie	domeniu de valori
`< > <= >= <> =`	culoare × culoare	`boolean`
`succ pred`	culoare	culoare
`ord`	culoare	`integer`

Iată mai jos pentru fiecare tip ce operatori are (dacă este fundamental) şi căror operatori le dă naştere:

`boolean`

operatori	domeniu de definiţie	domeniu de valori
`= <> < > <= >=`	`boolean` × `boolean`	`boolean`
`and or`	`boolean` × `boolean`	`boolean`
`not`	`boolean`	`boolean`
`succ pred`	`boolean`	`boolean`
`ord`	`boolean`	`integer`

`integer`

operatori	domeniu de definiţie	domeniu de valori
`= <> < > <= >=`	`integer` × `integer`	`boolean`
`succ pred`	`integer`	`integer`
`ord`	`integer`	`integer`
`+ - * div mod`	`integer` × `integer`	`integer`
`odd`	`integer`	`boolean`

`char`

operatori	domeniu de definiţie	domeniu de valori
`= <> < > <= >=`	`char` × `char`	`boolean`
`succ pred`	`char`	`char`
`ord`	`char`	`integer`
`chr`	`integer`	`char`

`real`

operatori	domeniu de definiţie	domeniu de valori
`+ - / *`	`real` $\times$ `real`	`real`
`= <> < > <= >=`	`real` $\times$ `real`	`boolean`

pentru orice tip enumerare T

operatori	domeniu de definiţie	domeniu de valori
`pred succ`	T	T
`= <> < > <= >=`	T×T	`boolean`
`ord`	T	`integer`

pentru orice tip subrange T

moşteneşte toate operaţiile tipului din care este derivat

pentru orice tip `T = array[S] of B`

operatori	domeniu de definiţie	domeniu de valori
`[]`	T×S	B

(operaţia de indexare aplicată astfel `t[s]` unde $t \in T$ şi $s \in S$).

pentru orice tip mulţime `T = set of S`

operatori	domeniu de definiţie	domeniu de valori
`= <> <= >=`	T×T	`boolean`
`in`	S×T	`boolean`
`+ - *`	T×T	T
`[]` (discutabil)	S	T

(ultimul este constructorul de mulţime aplicat astfel: `[s, s1, s2..s3]` unde $s1, s2, s3, s4 \in S$).

pentru orice tip `record R = record e1:T1; ...  en:Tn; end`

operatori	domeniu de definiţie	domeniu de valori
.	R×mulţimea etichetelor	$\bigcup_i T_i$

(ultima este operaţia de selecţie, care se aplică astfel `r.e` unde $r \in R$ şi e este una din etichete.)

pentru orice tip pointer PA $= \uparrow$**A**

operatori	domeniu de definiţie	domeniu de valori
$\uparrow$	PA	A

(Operaţia de dereferenţiere, aplicată astfel: pa $\uparrow$ unde $pa \in PA$.)

pentru tipul fişier `F = file of T`

operatori	domeniu de definiţie	domeniu de valori
$\uparrow$	F	I

(Operaţia de scriere/citire din buffer aplicată astfel: f$\uparrow$ unde $f \in F$.)

În plus pe lîngă operaţiile descrise anterior o valoare din orice tip poate fi atribuită (`:=`) unei variabile de acelaşi tip (cu excepţia fişierelor).

8 Polimorfism şi supraîncărcare

După cum vedeţi anumiţi operatori sunt definiţi pentru mai multe tipuri. De exemplu `succ` merge pe orice tip scalar, dînd un rezultat de acelaşi tip. în realitate $succ : integer \rightarrow integer$ şi $succ : char \rightarrow char$ sunt doi operatori diferiţi dar care au acelaşi nume. Un astfel de operator, care funcţionează pe mai multe tipuri de date simultan şi a cărui comportare depinde de tipul la care se aplică se numeşte *supraîncărcat*. Adesea astfel de operatori sunt bineveniţi, adesea însă pot să dea naştere la probleme. De exemplu teoretic există patru operatori + astfel:

$$
\begin{aligned}
+ &: integer \times integer \rightarrow integer \\
+ &: real \times real \rightarrow real \\
+ &: integer \times real \rightarrow real \\
+ &: real \times integer \rightarrow real
\end{aligned}
$$

Toţi aceştia sunt supraîncărcări ale lui +. (Ca să nu vorbim de folosirea lui + ca reuniune de mulţimi). Putem să aplicăm unul şi să ne gîndim la altul, cum ar fi în următorul exemplu:

```
var
  i,j : integer;
  r : real;

  i := j + r;
```

Ori asta nu se poate, pentru că aici + este presupus a fi

$$+ : integer \times real \rightarrow integer$$

care nu există!

Există un gen special de operatori care seamănă cu cei supraîncărcaţi dar care nu sunt supraîncărcaţi ci *polimorfici*. Distincţia între ei nu este cîteodată foarte precisă. În Pascal la acest titlu aspiră de pildă operatorii "`.`" (pentru selecţie), "`[]`" (pentru indexare), etc.

De ce? Pentru că operatorul `[]` este definit pentru orice tipuri `R, S, B`:

```
R = array [S] of B
[] : R x S -> B
```

şi operează la fel independent de natura tipurilor `S` (scalar) şi `B`, extrăgînd din valoarea de tipul `R` (un array) pe cea etichetată cu valoarea dată de tipul `S`. Putem spune că `[]` este un *singur* operator, oricare ar fi tipul `R` format ca mai sus. `[]` este atunci un operator polimorfic — pentru că lucrează cu mai multe tipuri dar în exact acelaşi mod.

(Să observăm că mai există un operator notat tot cu `[]` pentru construit mulţimi, care este tot polimorfic! Cei doi operatori sunt distinşi după modul în care se scriu în expresie. Acesta este tot o supraîncărcare!)

9 Glosar cu traduceri în engleză

Tip de date (data type) o mulţime de valori căreia adesea i se adaugă un set de operatori şi un set de reguli de folosire a operatorilor.

Puternic tipizat (strongly typed) (despre un limbaj) care nu poate genera erori cauzate de tipuri eronate în timpul execuţiei.

Operator (operator) o operaţie între mai multe valori dînd ca rezultat o altă valoare; strict vorbind o funcţie (parţială) definită pe un produs cartezian de tipuri cu valori într-un tip.

Polimorfic (polymorphic) un operator care operează la fel asupra mai multor tipuri de date structurate asemănator.

Supraîncărcat (overloaded) (despre un operator) în realitate mai mulţi operatori cu acelaşi nume, dar care sunt distinşi după contextul în care se aplică (tipurile argumentelor şi al rezultatului).

Tip scalar (scalar type) un tip cu un număr finit de elemente asupra cărora este fixată o relaţie de ordine totală (cu ajutorul unor operatori).