Philosophie générale de MuPAD
Différences par rapport à Maple

Paul Zimmermann

Fonctions de base

Le premier argument est toujours l'opérande principal :

```
>> l:=[a,b,c]: op(l, 2);

                                  b

>> l:=[[a,1],[b,2],[c,3]]: map(l, op, 1);

                               [a, b, c]
```

Le dernier cas s'écrit `map2(op, 1, l)` en MAPLE, nécessitant une fonction spéciale `map2`.

Primitives de programmation

f := **proc**(. . .)
begin
. . .
end_proc

if . . . **then** . . . **elif** . . . **else** . . . **end_if**

for i **from** 0 **to** 100 **step 3 do** . . . **end_for**

while cond **do** . . . **end_while**

Nouvelles primitives

repeat … **until** cond **end_repeat**

```
>> case i
   of 1 do r:=1; break;
   of 2 do r:=2; next;
   of 3 do
   of 2 do r:=3; break;
   otherwise r:=4
   end_case;
```

$i = 1 \to r = 1,\quad i = 2 \to r = 3,\quad i = 3 \to r = 3,\quad i = 4 \to r = 4$

Instructions vs expressions

Le résultat de toute instruction peut être utilisé comme expression :

```
>> x:=13: a:=(if x mod 2 = 0 then x/2 else 3*x+1 end_if);
                                    40
```

Portée dynamique des variables

MuPAD 1.4 :

```
>> x:=3:   q:=proc() begin x end_proc:
>> p:=proc(x) begin q() end_proc:  p(1);

                              1

>> p:=proc(x) local q;
   begin q:=proc() begin x end_proc; q() end_proc:  p(1);

                              1
```

MuPAD 1.5 implante la portée lexicale comme Maple V.5, et renverra 3 et 1 respectivement.

Attributs de fonctions

```
>> d := funcattr(d, "expand", proc(x) begin
      if type(x)="_plus" then map(x,d) else d(x) end_if
   end_proc):
>> d(a+2*b+3*c);

                        d(a + 2 b + 3 c)

>> expand(%);

                  d(a) + d(2 b) + d(3 c)
```

Vérification automatique de types

```
>> fib := proc(n : DOM_INT) : DOM_INT begin
     if n>0 then fib(n-1)+fib(n-2) else 1/2 end_if end_proc:
>> fib(0):=0: fib(1):=1:
>> fib(x);

Error: Wrong type of argument n (type DOM_INT expected)
       in procedure fib. Actual argument is x.

>> fib(-1);

Error: Wrong type of returned value (type DOM_INT expected)
       in procedure fib. Actual value is 1/2.
```

Domaines

Deux types de domaines :

- domaines bruts
- domaines “mathématiques”

Domaines bruts

Surcharge automatique des fonctions du noyau, mais pas d'héritage.

```
>> d := domain("annuaire"):
>> claude := new(d, "A209", 3011);
                     new(annuaire, "A209", 3011)


>> extop(claude, 1);

                               "A209"


>> d::print:=proc(x) begin extop(x,1)." -> ".extop(x,2) end_proc:
>> claude;

                            A209 -> 3011
```

```
>> d::phone := proc(x) begin extop(x,2) end_proc:
>> phone(claude);
                                 phone(A209 -> 3011)
>> d::phone(claude);
                                        3011


>> phone := proc(x) begin
     if x::phone<>FAIL then x::phone(args())
     else procname(x) end_if
   end_proc:
>> phone(claude);
                                        3011


>> d::new := proc() begin new(d,args()) end_proc:
>> marie_christine := d("B134", 2052);

                                    B134 -> 2052
```

Domaines mathématiques

Permettent à la fois surcharge et héritage.

– prédéfinis (domaine **Dom**)

– définis par l'utilisateur

```
>> Z17 := Dom::IntegerMod(17):
>> a:=Z17(3): b:=Z17(5): a/b; # implante' dans domaine #

                                  4 mod 17
>> gcd(a, b); # he'rite' de Field #

                                  1 mod 17
```

La définition de IntegerMod

```
Dom::IntegerMod:= DomainConstructor(
    Dom::IntegerMod,
    [ Mod ],
    [ ],
    (if args(0) <> 1 then error("wrong no of args") end_if;
     if domtype(Mod) <> DOM_INT then error("...") end_if;
     if Mod<2 then error("modulus must be > 1") end_if),
    Dom::BaseDomain,
    [ (if isprime(Mod) then Cat::Field
        else Cat::CommutativeRing end_if) ],
    [ Ax::canonicalRep, Ax::effectiveOperation("_mult"), ...
    ],
    "characteristic" = Mod, ...
):
```

Le domaine `BaseDomain`

C'est le domaine de base (minimal) :

```
Dom::BaseDomain:= DomainConstructor(
    Dom::BaseDomain,
    [],
    [],
    NIL,
    NIL,
    [ Cat::SetCat ],
    [],
    ...
):
```

Implante les méthodes `equal`, `hasProp`, `allAxioms`, `allSuperDomains`, `allCategories`, `whichEntry`, `allEntries`.

```
>> Z16:=Dom::IntegerMod(16): Z16::hasProp(Cat::Field);

                                       FALSE
>> Z17::allAxioms();

{Ax::canonicalRep, Ax::normalRep, Ax::effectiveOperation("_mult"),

   Ax::canonicalUnitNormal, Ax::effectiveOperation("_invert"),

   Ax::noZeroDivisors, Ax::effectiveOperation("_divide"),

   Ax::closedUnitNormals}

>> Z17::allSuperDomains();

                                  [Dom::BaseDomain]
```

```
>> Z17::allCategories();

[Cat::Field, Cat::EuclideanDomain, Cat::FactorialDomain,
 Cat::PrincipalIdealDomain, Cat::GcdDomain, Cat::Ring,Cat::SetCat,
 Cat::IntegralDomain, Cat::Rng, Cat::Monoid, Cat::SkewField,
 Cat::LeftModule(Dom::IntegerMod(17)), Cat::EntireRing,
 Cat::CommutativeRing, Cat::Algebra(Dom::IntegerMod(17)),
 Cat::AbelianGroup, Cat::SemiGroup, Cat::AbelianSemiGroup,
 Cat::RightModule(Dom::IntegerMod(17)), Cat::AbelianMonoid,
 Cat::Module(Dom::IntegerMod(17)), Cat::CancellationAbelianMonoid]

>> Z17::whichEntry("_mult");

                              Dom::IntegerMod(17)
>> Z17::whichEntry("gcd");

                                 Cat::Field
```

```
>> Z17::allEntries();

{"_invert", "one", "printElem", "testtype", "sqrfree", "iszero",
   "_subtract", "equiv", "rem", "getAxioms", "unitNormalRep",
   "gcd", "whichEntry", "coerce", "print", "size", "unitNormal",
   "equal", "new", "divides", "euclideanDegree", "divide", "subs",
   "allEntries", "_power", "Factor", "allCategories",
   "characteristic", "convert", "convert_to", "expr", "_negate",
   "zero", "_divide", "order", "irreducible", "new_extelement",
   "isone", "associates", "allSuperDomains", "info", "isUnit",
   "getCategories", "_mult", "lcm", "gcdex", "isSquare", "factor",
   "allAxioms", "_plus", "quo", "idealGenerator", "hasProp",
   "undefinedEntries", "TeX", "random", "getSuperDomain",
   "subsex", "intmult"}
```

Philosophie du système

Au lieu d'avoir plein de fonctions (`solve`, `isolve`, `msolve`, `rsolve`, `dsolve`, `pdesolve`, `...`), on en a une seule (`solve`) avec des domaines différents :

```
>> eq:=ode( {f''(t)+4*f(t)=sin(2*t), f(0)=0, D(f)(0)=0}, f(t)):
>> solve(eq);

                      { sin(2 t)   t cos(2 t) }
                      { -------- - ---------- }
                      {    8           4      }


>> solve(rec(u(n+1) = 2*u(n)*(n+1)/n, u(n)));

                                    n - 1
                           {n u(1) 2     }
```

Les polynômes

Deux représentations :

- sommes de produits à la Maple
- poly(expression, liste_vars, type_coeffs)

```
>> p:=x^5 + y^5:
>> q:=poly(x^5 + y^5, [x,y]);
                                  5    5
                           poly(x  + y , [x, y])

>> q:=poly(x^5 + y^5, [x,y], Z17);


                      5    5
               poly(x  + y , [x, y], Dom::IntegerMod(17))
```

```
>> Z17::name:="Z17": q^4;

        20        15   5        10   10        5   15     20
  poly(x    + 4 x    y   + 6 x    y    + 4 x   y    + y   , [x, y], Z17)

>> factor(q);

                 4        3       2   2          3     4
[1 mod 17, poly(x  + 16 x   y + x   y  + 16 x y  + y , [x, y], Z17),

   1, poly(x + y, [x, y], Z17), 1]
```

Calcul de bases de Gröbner

```
>> l := [x^2 - x + 2*y^2, x + 2*y^3 - 1]:
>> groebner::gbasis(l, LexOrder);

                        3        2    3      6
                [x + 2 y  - 1, y  - y  + 2 y ]

>> groebner::gbasis(map(l, poly, Z17), LexOrder);

           3                            6      3      2
[poly(x + 2 y  + 16, [x, y], Z17), poly(y  + 8 y  + 9 y , [x, y],

   Z17)]
```