Can R learn from julia?

Jan Vitek

Northeastern University

`https://github.com/janvitek/can_R_learn_from_Julia.git`

Northeastern University
College of Computer and Information Science

CZECH TECHNICAL UNIVERSITY IN PRAGUE

julia is…

…a dynamic language for high-performance scientific computing

…open source since its inception by Jeff Bezanson circa 2002

	R	julia
Dynamic	yes	yes
Vectorized	yes	yes
Memory management	automatic	automatic
Implementation	interpreted	native
Type declarations	—	user-defined generic types
Meta-programming	`substitute()`	macros
Parameter passing	by promise	by value

Outline

Compare features of the two languages focusing on **polymorphism** and **performance**

Use R's `colMeans` function as running example

Show same level of *dynamism* and *polymorphism* **without** sacrificing performance

`https://github.com/janvitek/can_R_learn_from_Julia.git`

colMeans

```
x = 1:100
dim(x) = c(50,2)
colMeans(x)
[1] 25000.5 75000.5

x = complex(r=1:60,i=1:60)
dim(x) = c(10,3,2)
colMeans(x)
           [,1]        [,2]
[1,]  5.5+ 5.5i 35.5+35.5i
[2,] 15.5+15.5i 45.5+45.5i
[3,] 25.5+25.5i 55.5+55.5i
```

colMeans

```
colMeans = function(x, na=FALSE, dims=1L) {
 dn   = dim(x)
 id   = 1:dims
 n    = prod(dn[id])
 dn   = dn[-id]
 pdn = prod(dn)
 z    = if (is.complex(x))
   .Internal(colMeans(Re(x),n,pdn,na))+(0+1i)*
   .Internal(colMeans(Im(x),n,pdn,na))
  else .Internal(colMeans(x,n,pdn,na))
 z
}
```

colMeans

Most of the behavior implemented in C in do_colsum()

```c
SEXP attribute_hidden do_colsum(SEXP call, SEXP op, SEXP args, SEXP rho) {
    SEXP x, ans = R_NilValue;
    int type;
    Rboolean NaRm, keepNA;

    checkArity(op, args);
    x = CAR(args); args = CDR(args);
    R_xlen_t n = asVecSize(CAR(args)); args = CDR(args);
    R_xlen_t p = asVecSize(CAR(args)); args = CDR(args);
    NaRm = asLogical(CAR(args));
    if (n == NA_INTEGER || n < 0)
        error(_("invalid '%s' argument"), "n");
    if (p == NA_INTEGER || p < 0)
        error(_("invalid '%s' argument"), "p");
    if (NaRm == NA_LOGICAL) error(_("invalid '%s' argument"), "na.rm");
    keepNA = !NaRm;

    int OP = PRIMVAL(op);
    switch (type = TYPEOF(x)) {
    case LGLSXP: break;
    case INTSXP: break;
    case REALSXP: break;
    default:
        error(_("'x' must be numeric"));
    }

    if (OP == 0 || OP == 1) { /* columns */
        PROTECT(ans = allocVector(REALSXP, p));
        for (R_xlen_t j = 0; j < p; j++) {
            R_xlen_t  cnt = n, i;
            LDOUBLE sum = 0.0;
            switch (type) {
            case REALSXP:   {
                double *rx = REAL(x) + (R_xlen_t)n*j;
                if (keepNA)
                    for (sum = 0., i = 0; i < n; i++) sum += *rx++;
                else
                    for (cnt = 0, sum = 0., i = 0; i < n; i++, rx++)
                        if (!ISNAN(*rx)) {cnt++; sum += *rx;}
                break;
            }
            case INTSXP: {
                int *ix = INTEGER(x) + (R_xlen_t)n*j;
                for (cnt = 0, sum = 0., i = 0; i < n; i++, ix++)
                    if (*ix != NA_INTEGER) {cnt++; sum += *ix;}
                    else if (keepNA) {sum = NA_REAL; break;}
                break;
            }
            case LGLSXP: {
                int *ix = LOGICAL(x) + (R_xlen_t)n*j;
                for (cnt = 0, sum = 0., i = 0; i < n; i++, ix++)
                    if (*ix != NA_LOGICAL) {cnt++; sum += *ix;}
                    else if (keepNA) {sum = NA_REAL; break;}
            break;
            }
            }
            if (OP == 1) sum /= cnt; /* gives NaN for cnt = 0 */
            REAL(ans)[j] = (double) sum;
        }
    } else { /* rows */
        PROTECT(ans = allocVector(REALSXP, n));

        /* allocate scratch storage to allow accumulating by columns
           to improve cache hits */
        int *Cnt = NULL;
        LDOUBLE *rans;
        if(n <= 10000) {
            R_CheckStack2(n * sizeof(LDOUBLE));
            rans = (LDOUBLE *) alloca(n * sizeof(LDOUBLE));
            Memzero(rans, n);
        } else rans = Calloc(n, LDOUBLE);
        if (!keepNA && OP == 3) Cnt = Calloc(n, int);

        for (R_xlen_t j = 0; j < p; j++) {
            LDOUBLE *ra = rans;
            switch (type) {
            case REALSXP:
            {
                double *rx = REAL(x) + (R_xlen_t)n * j;
                if (keepNA)
                    for (R_xlen_t i = 0; i < n; i++) *ra++ += *rx++;
                else
                    for (R_xlen_t i = 0; i < n; i++, ra++, rx++)
                        if (!ISNAN(*rx)) {
                            *ra += *rx;
                            if (OP == 3) Cnt[i]++;
                        }
                break;
            }
            case INTSXP:      {
                int *ix = INTEGER(x) + (R_xlen_t)n * j;
                for (R_xlen_t i = 0; i < n; i++, ra++, ix++)
                    if (keepNA) {
                        if (*ix != NA_INTEGER) *ra += *ix;
                        else *ra = NA_REAL;
                    }
                    else if (*ix != NA_INTEGER) {
                        *ra += *ix;
                        if (OP == 3) Cnt[i]++;
                    }
                break;
            }
            case LGLSXP:
            {
                int *ix = LOGICAL(x) + (R_xlen_t)n * j;
                for (R_xlen_t i = 0; i < n; i++, ra++, ix++)
                    if (keepNA) {
                        if (*ix != NA_LOGICAL) *ra += *ix;
                        else *ra = NA_REAL;
                    }
                    else if (*ix != NA_LOGICAL) {
                        *ra += *ix;
                        if (OP == 3) Cnt[i]++;
                    }
                break;
            }
            }
        }
        if (OP == 3) {
            if (keepNA)
                for (R_xlen_t i = 0; i < n; i++) rans[i] /= p;
            else
                for (R_xlen_t i = 0; i < n; i++) rans[i] /= Cnt[i];
        }
        for (R_xlen_t i = 0; i < n; i++) REAL(ans)[i] = (double) rans[i];

        if (!keepNA && OP == 3) Free(Cnt);
        if(n > 10000) Free(rans);
    }

    UNPROTECT(1);
    return ans;
}
```

```c
SEXP attribute_hidden
do_colsum(SEXP call, SEXP op,
          SEXP args, SEXP rho) {
    SEXP x, ans = R_NilValue;
    int type;
    Rboolean na;
    checkArity(op, args);
    x    = CAR(args);
    args = CDR(args);
    R_xlen_t n=asVecSize(CAR(args));
    args = CDR(args);
    R_xlen_t p=asVecSize(CAR(args));
    args = CDR(args);
    na   = !asLogical(CAR(args));
```

```c
if (n == NA_INTEGER || n < 0)
    error(_("invalid '%s'"), "n");
if (p == NA_INTEGER || p < 0)
    error(_("invalid '%s'"), "p");
if (na == NA_LOGICAL)
error(_("invalid '%s'"), "na.rm");

int OP = PRIMVAL(op);
switch (type = TYPEOF(x)) {
case LGLSXP: break;
case INTSXP: break;
case REALSXP: break;
default:
    error(_("'x' must be numeric"));
}
```

```
PROTECT(ans=allocVector(REALSXP,p));
for(R_xlen_t j=0; j<p; j++) {
  R_xlen_t  cnt=n, i;
  LDOUBLE sum = 0.0;
  switch (type) {
  case REALSXP:    {
   double *rx = REAL(x)+ n*j;
   if(na)
      for(sum=0,i=0;i<n;i++)
         sum += *rx++;
   else
      for(cnt=sum=i=0;i<n;i++,rx++)
         if(!ISNAN(*rx)){
            cnt++; sum += *rx;
         }
   break;
```

julia

```
function colMeans(x, na=true, dims=1)
    dn  = size(x)
    id  = [1:dims;]                   # 1:dims
    n   = prod(dn[id])
    dn  = extract(dn,id)              # dn[-id]
    pdn = prod(dn)
    res = zeros(pdn)                  # 0
    for j = 0:pdn-1                   # for(j in 0:pdn-1) {
        sum = z(x[1])                 # 0
        cnt = 0
        off = j*n
        for i = 1:n                   # for(i in 1:n) {
            v = x[i+off]
            cnt += 1                  # cnt = cnt + 1
            sum += v                  # sum = sum + 1
        end
        res[j+1] = sum/cnt
    end
    res
end
```

Multi-Dispatch

```
z(x::AbstractFloat) = 0.0
z(x::Complex) = complex(0.0,0.0)
z(x) = 0

sum = z(x[1])          # 0
```

Julia functions are multi-dispatched # S4, subsuming S3

Types part of language syntax # in S4 types are a DSL

To avoid boxing, variables must be initialized with the "right" type; the above is needed to keep colMeans polymorphic

Generics

```
is_na{T}(x::T) =
      x == typemin(T)

typemin{T<:Complex}(::Type{T}) =
      T(-NaN)
```

Julia lacks a builtin missing value; we steal smallest member of each data type.

Generic functions can operate over types; type variables can be bounded.

```
if (!is_na(v))
    …
elseif na_rm
    sum = typemin(typeof(z(x[1])))
```

Other changes to support missing values are straightforward.

User Defined Types

```
bitstype 8 ThreeWay
ThreeWay()              = reinterpret(ThreeWay, 0xff)
ThreeWay(x::Bool) = reinterpret(ThreeWay, x)

const true3   = ThreeWay(true)
const false3 = ThreeWay(false)
const na3       = ThreeWay()

typemin(::Type{ThreeWay}) = na3

==(x::ThreeWay, y::Bool) =
    ifelse(x==na3, false, Bool(x)==y)

+(x::Union{Int, ThreeWay}, y:: ThreeWay) =
     Int(x) + Int(y)
```

Conclusions

Julia achieves polymorphism and performance with a combination of three features

1. Specialization and runtime code generation
2. User defined generic data types
3. Efficient multi-dispatch

We are working on (1) in Reactor.

(2) and (3) are within reach, modulo language changes

https://github.com/janvitek/can_R_learn_from_Julia.git