The nature of vector types

Languages can be categorized by the degree of dynamism associated with their vector types and the implications this has for how vector operations are done.

Continuum

Most dynamic -> least dynamic

Lisp -> APL -> Algols -> Fortran

lisp like	APL like	Algol like	Fortran like
Scheme	A+ J S-algol Java Hi	Delphi Algol60 ML Haskell	C Pascal Ada

Note that at level of abstraction I treat lists, arrays and vectors as 'syntactic sugar' over the basic concept of a sequence of values.

Dynamism

- Dynamic rank
- Fixed rank mutable array size
- Fixed rank array size fixed at point of declaration - may depend on function parameters
- Statically known array size

Bounds

- Are they checked
- Do they start at zero

Lisp Like

A list is a sequence of values which may be either atoms or lists.

This allows lists of dynamically defined rank, and even of variable rank.

((2, 3, 7), 6)

The first element of the list is a list of 3 atoms and the second element is an atom. Thus the notion of rank is not defined in this case.

APL Like

All arrays have a well defined rank (number of dimensions), and a well defined shape (number of elements in a dimension) , but these can vary for a given array variable in the course of computation.

APL: x$\leftarrow\iota$ n

x gets the sequence of integers from 1 to n

S-Algol : let x:= vector 1::n of 1

x becomes the array of n storage locations, all initialised to 1

Java: x= new int[n];

x becomes the array of n storage locations initialised to 0

Note the difference in degrees of initialisation supported.

Algol 60 like

On entry to a block the array size is specified by values that may be computed at run time but once the variable has been created its size is un-varying.

```
read(x);
begin
  integer [1:x] a;
  for i:=1 to x step 1 do read(a[i]);
  ....
```

a has a fixed size during this block and is discarded on leaving the block.

Fortran Like : for these see section 6.2.6 of *Modern Compiler Design*

In this case the compiler knows as soon as it has parsed the declaration exactly how much store will be used. The size of the array is fixed.

Additional issue here is whether the bounds have to start at a fixed number such as 0 or 1 or can be defined by the programmer:

C: int x[6];

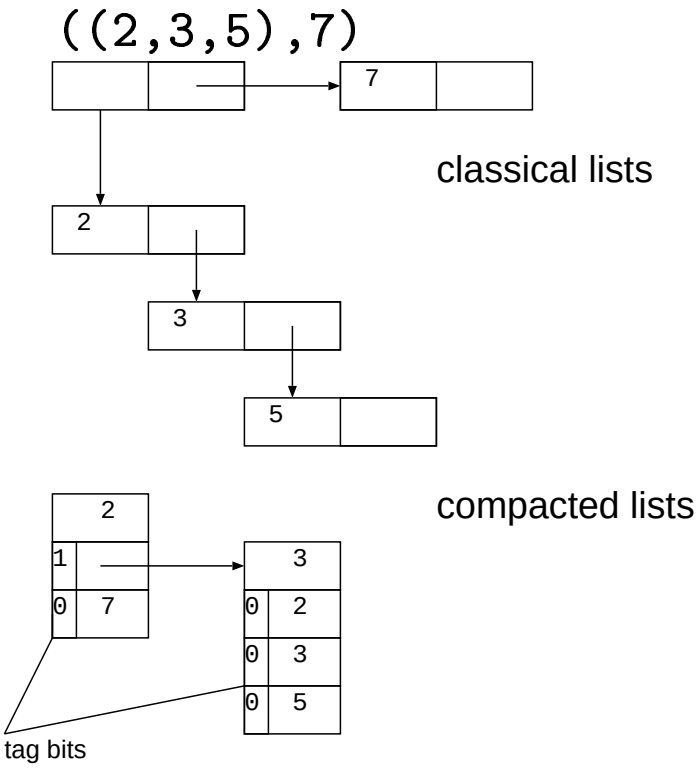
reserve 6 store locations numbered 0 to 5

Pascal : var sales: array[1996..2004] of integer;

reserve 8 store location numbered 1996 to 2004

The latter form has some advantages from the standpoint of intelligibility of code.

Lisp Like


((2,3,5),7)
7
2
3
5
classical lists
2
1
0
7
3
0
2
0
3
0
5
tag bits
compacted lists

TAGS

Words can be tagged using the top bit of each word to indicate a pointer. This allows arbitrary list structures to be disambiguaged but this has the effect of reducing arithmetic precision.

Assume you have a 32 bit word. Then the normal sign bit (bit 31) is reserved as a tag. You only have the normal range of +ve numbers available as integers $0..2^{31}-1$

We have to map this into a range of -ve and +ve numbers.

Do this with the operations: getint(t)=$t+2^{30}$

putint(i,t) $t \leftarrow i-2^{30}$

Some special purpose Lisp Machines have provided 40 bit words with 8 bit tags. These distinguished a number of different types.

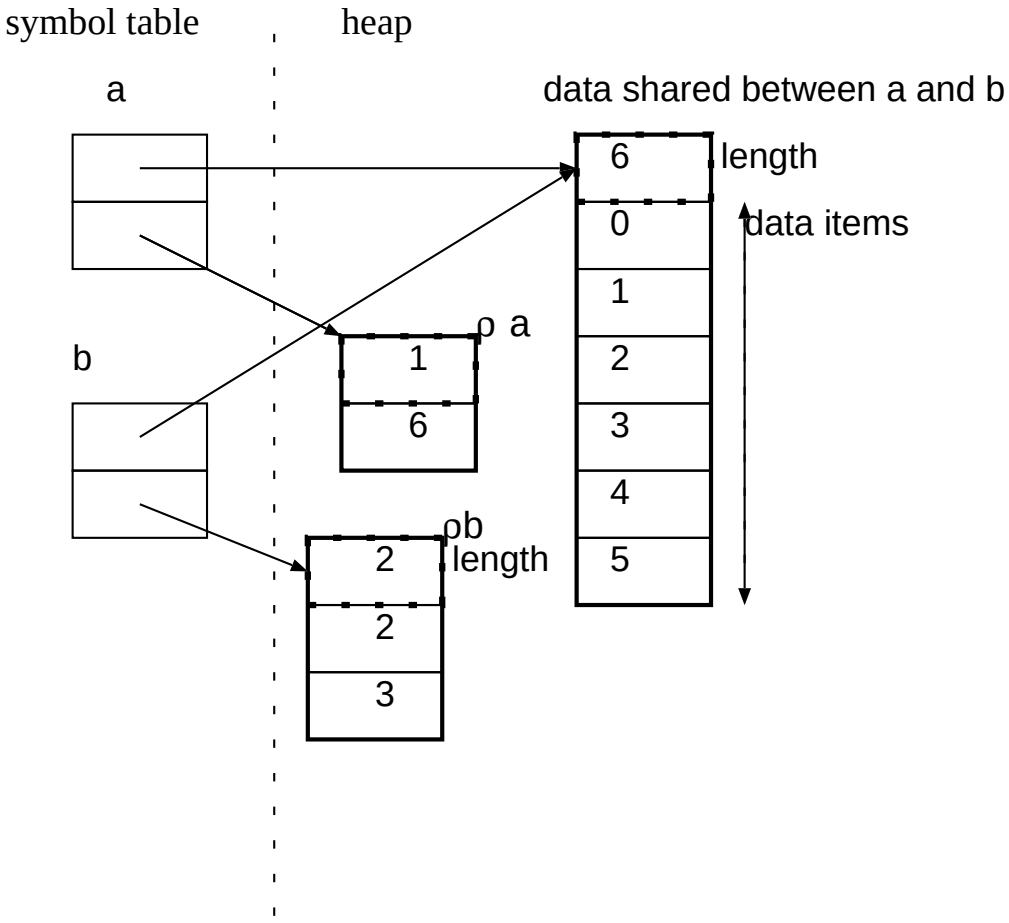
One, the *Rekursiv*, was designed by Prof Harland, formerly of this department. I have an example here.

APL type arrays

these can be reshaped as the following A+ example shows

```
   a:=  ⍳ 6
   a
0 1 2 3 4 5
   ⍴ a
6
   b:= 2 3 ⍴a
   b
0 1 2
3 4 5
   ⍴ b
2 3
```

Here ρ is the shape and reshape operator. ι is the operator that generates a sequence of integers.

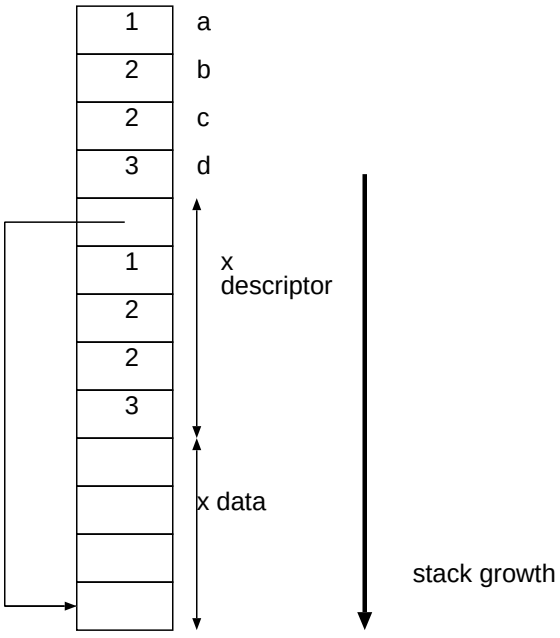


Note that each variable is stored as a data vector and a shape vector which says how to interpret the data vector. Arrays on the heap.

Algol-Like arrays

In this case the array has both upper and lower bounds supplied when it is created:

```
begin
   integer a,b,c,d;
   read(a,b,c,d);
   begin
     integer array x[a:b.c:d];
     .....
```



Note that the array is created on the stack, the stack pointer being moved down when the array declaration is encountered.

The array descriptor contains a pointer to the start of the array.

1. initialise the bounds in the descriptor

2. compute the space required $s = (1 + upb_1 - lwb_1) \times (1 + upb_2 - lwb_2)$

3. subtract s from the stack pointer

4. set the pointer field of the descriptor to the stack pointer.

C arrays

int a[2][3];

Compiler statically reserves 6 words of memory for the array.

Allocated typically on the stack, but in data segment if they are declared as static.

Vector Operations

1. | concatenate

2. iota - create a sequence of integers

3. [] create a vector from values

4. Map an operation over vectors

We now look at how vector operations can be performed. These are operations whose result is a vector.

The concatenate operation builds a vector out of two vectors, iota creates a sequence of integers, [] builds a sequence of values of rank n into a value of rank n+1.

Mapping takes a collection of vectors and an operator or function. It applies the operator or function to n-tuples of values selected from the vectors.

Major design issue

Should these operations be done using library functions or should you generate machine code to perform each one?

In principle all could be done using direct machine code, not all can be done entirely using library functions.

Trade-offs

- Library function implementations are usually easier to implement

- Direct machine code is faster

Whilst these Hi examples do not apply directly to other languages, there are analogous issues raised in the implementation of high level operations in many languages.

Operator overloading

Polymorphism

Set operations

the operator |

```
a|b
```

is the concatenation of the vectors a,b

This is best implemented as a library routine which

1. determines the sizes of a and b

2. creates a new vector large enough to hold them both

3. copies in the contents of a to the low numbered elements

4. copies the contents of b to the high numbered elements

5. returns the address of the new vector

Iota

modeled on the APL operator or the same name. Produces vector of ascending integers.

```
/*example of iota*/
HiMain(->scalar)
 putNum(rtotal(putNum(iota(4),1));
rtotal(vec1 x,scalar i->scalar)
 if i>length(x)
 then 0
 else x(i)+rtotal(x,i+1)
 fi;
```

outputs

```
1  2  3  4  10
```

tl(x)

One can also implement hd(list) and tl(list) using iota:

```
/* use of iota to implement hd and tl on lists *
hd(vec1 x->scalar)x(1);
tl(vec1 x->vec1)x(1+ iota(length(x)-1));
HiMain(->scalar)putNum(total( tl([1,3,5]),1));
total(vec1 x,scalar i->scalar)
 if  ( i)>length(x) then 0
 else x(i)+ total(x,i+1)
 fi;
```

Consider the expression

```
x(1+ iota(length(x)-1))
```

```
x(1+ iota(length(x)-1))
```

with x=[1,3,5]

```
length(x)=3
length(x)-1=3-1=2
iota(2)=[1,2]
1+iota(2)= [2,3]
x([2,3])= [x(2),x(3)]=[3,5]
```

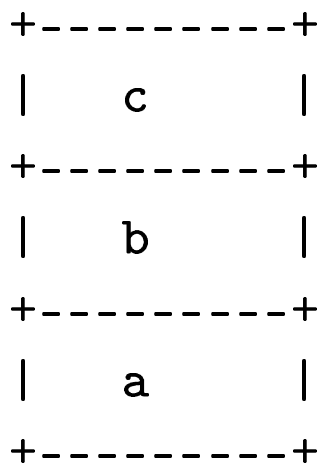
Iota is easily implemented by a library function

There exist more efficient implementations that we shall discuss after looking at Map.

[ ] vector construction

[a, b, c]

A reversed depth first traversal of the parse tree for this will produce code whose execution will leave the stack looking like



We then plant code to push the length of the vector on the stack. So for the hi code

```
[1,2,4]
```

we would get

```
push 4
push 2
push 1
push 3
```

we now call the heap allocator to get us space

```
extern sysvecalloc
call sysvecalloc
```

At this point the eax register contains the address of the unintialised vector on the heap. We want to copy the vector on the stack onto the heap witha block move instruction.

```
pop ecx          ; load count
mov edx,ecx      ; save in edx
lea edi,[eax+4]; load dest address
mov esi,esp
rep movsd
```

Lets look at this in more detail:

A block move sequence takes the address of a source vector in esi and a destination in edi and a count in ecx.

register	use
ecx	count register
esi	source index register
edi	destination index register

```
pop ecx
```

will place 3 in ecx

Load effective address

```
lea edi,[eax+4]
```

means load effective address of memory location [eax+4] into the edi register. That is the destination is set to point at location [1] in the new array. The offset of 4 is to allow for the length field of the array which will be initialised by the heap allocator.

MOVSD

The `movsd` instruction copies a 32 bit word from mem[esi] to mem[edi] and then increments both registers.

```
tmp<= mem[esi]
mem[edi]<=tmp
esi<= esi+1
edi<= edi+1
```

REP

The rep instruction is a prefix code that can preceed any other opcode. It has the effect of causing that opcode to be repeated until the ecx register counts to zero. Thus

```
rep foo
```

means

```
while ecx >0 do
begin
   ecx:=ecx-1
   foo
end
```

Finally we discard the data from the stack by moving the stack pointer up by the number of words in the vector:

```
lea esp,[esp+edx*4]
```

and then push the address of the vector on the stack:

```
push eax
```

Issues

1. Would it be better to allocate the array first and then assign the values in the brackets into the array 1 by 1? This would appear to save pushing them onto the stack first.

2. Would we be better to use vector move instructions instead of block move instructions?

Answers

1. Probably not. Given our general strategy for evaluating expressions the vector elements are likely to be on the stack at some point. The block move is then more efficient than a series of pop instructions.

2. Yes but it is more complicated. If we know that the vector is of length 4 then this is a sensible optimisation. We replace the sequence:

```
pop ecx          ; load count
mov edx,ecx      ; save in edx
lea edi,[eax+4]; load dest address
mov esi,esp
rep movsd
```

with

```
movdqu xmm0, [esp]   ; get the vector in a regist
movdqu [eax],xmm0    ; store on the heap
add esp,16           ; drop it from the stack
```

which will probably run faster, but is less general

Map operations over arrays

Let us look at 3 kinds of mapping:

1. Dyadic operations

2. Function calls

3. Array indexing

First we will look at a functional programming approach, and then we will look at an optimised imperative approach.

Dyadic operations

a+b

consider the combinations of the following possibilities

	a	b
scalar	1	2
vector	[3,4]	[7,11,19]

```
1+2                 = 3
1+[7,11,19]         = [8,12,20]
[3,4]+2             = [5,6]
[3,4]+[7,11,19]= [10,15]
```

Note that

1. the compiler will in general know the type of the arguments to an operator, but it will not know the lengths of the vectors

2. the operator may be implemented either as inline code or as a function

3. the examples are only the simplest cases, one also has to deal with higher rank arrays, consider:

```
putnum([1,3]+[[2,3],[4,6]])
3 4 7 9
```

Solution 1.

We write a function in C that maps a dyadic operator over two params and pass it the address of the operator, and the ranks of the arrays being passed.

The function allocates store for the result and loops round calling the operator to do the calculations on the arguments.

Template of the C function might by

```
int dyadicmap(int opcode,
              int xrank,
              int yrank,
              int x,
              int y);
```

so x+y is compiled to

```
push y
push x
push yrank
push xrank
push address of plus
call dynamicmap
```

What do we do if the operator is inline?

We synthesise an anonymous function:

```
llll2:
enter 0,0
push dword[ebp+12]
push dword[ebp+8]
   ; + what follows is standard inline +
   pop eax
   add [esp],eax
   ;-----------------------------------
pop eax
leave
ret 0
```

This makes the operator callable as a function.

Can we generalise this to other situations:

- Calling a scalar argued function on a vector, eg:

```
putNum([1,4,7]) where putNum(scalar x->scalar)
```

- Indexing a vector by a vector

```
x([2,3]) where x a vector of any rank
```

- Calling a function of mixed rank on argument list also of mixed rank, for instance extending vector dot product to matrix multiply:

```
/* define dot product of vectors */
.*(vec1 x,vec1 y->scalar)sum(x*y);
/* use it to do matrix product of vector */
HiMain(->scalar)
 discard(putNum([[2,2],[0,1]].*[2,4]));
```

this prints

```
12 4
```

also uses:

```
discard(vec1 x->scalar)1;
total(vec1 x,scalar i->scalar)
  if i>length(x) then 0
  else x(i)+total(x,i+1)fi;
sum(vec1 x->scalar)total(x,1);
```

These cases can all be handled by a more general form of mapping function in C.

```
int applytoall(
  int overloading,
  int * excessranks,
  int proc,
  int * p
)
```

`overloading` – the rank of the result

`excessranks` – the degree by which the rank of actual parameter p_i exceeds the rank of formal parameter p_i

`proc` – the address of the function to call

`p` – the actual parameters packaged into a vector

This recurses on “overloading” until overloading is zero, when it calls the proc with the appropriate parameters.

Whilst recursing it creates successive new layers of vectors and iterates through them to initialise them.

I leave it to you to look at the example implementation.

```
int applytoall(int overloading,int * excessranks,int proc
{
int * carrier,*tmp,*excess;
int i,j,l,bound;
l=params[0];
if(overloading==0){
  carrier=sysvecalloc(l+1);
  carrier[l+1]=proc;
  for(i=1;i<=l;i++)carrier[i]=params[i];
  return callindirect(carrier);
}
else {
  carrier=sysvecalloc(l);
  bound=0x7fffffff;
  for(i=1;i<=l;i++)
   if(excessranks[i])
   {tmp= (int*)params[i];
    if(tmp[0]<bound)bound=tmp[0];
   }
  tmp=sysvecalloc(bound);
  excess=sysvecalloc(l);
  for(i=1;i<=l;i++)
   excess[i]=(excessranks[i]?excessranks[i]-1:0);
  for(j=1;j<=bound;j++){
   for(i=1;i<=l;i++)
    carrier[i]=
    (excessranks[i]?((int*)params[i])[j]:params[i]);
   tmp[j]=applytoall(overloading-1,excess,proc,carrier);
  }
```

```
   return (int)tmp;
  }
}
```