Aliasing and Arrays

2nd September 2002

Rahul Balani 26-08-2002

Aliasing :

When a "location" (pointer,memory address) is referenced by two different variable names, its known as Aliasing. This is used in Fortran and Java for parameter passing by reference. But this can prove to be a little dangerous as it might lead to wrong logic as seen in the following example :

```
void swap (int &a, int &b) {
a = a - b;
b = a + b;
a = b - a;
}

main() {
int x = 5;
swap (x,x);
}
```

In this, both "a" and "b" are pointing to the location addressed by "x". As a result, the value of x is set to 0 !!

Avoiding aliasing bugs:

1. **Pass - by - value - result** Here you can make initial copies of parameter values, and operate on these values. Then you can copy the result back to original locations. For example, in the above code, we can use two temporary locations to store the value of x each, which in turn can be refernced by a and b respectively. After performing operations on a and b, the result (here a = 5 and b = 5) is copied back to the location pointed to by x.

Disadvantage : This method is expensive in terms of memory space, and hence inefficeint. Thus its not followed in later languages.

2. **Pass - by - Need** Now the languages like Java use a combination of Pass-by-reference and Pass-by-value i.e. they try to avoid copying as far as possible, and use it only when over-writing is occuring.

Arrays :

When arrays are introduced into a programming language, there is a conflict in the beginning regarding the syntax for initialising, assigning and accessing array elements. For example, in the **FLFL** (Fortran Like Functional Language) code given below :

letarray a[2]
let i = 0
begin
(:= a[0] 1) ...... confilct A
(:= a[1] 0)
(call swap i **a[i]**) ...... conflict B
(pair **a[0]** a[1]) ...... conflict C

Syntax for initialising the array :
letarray I[E$_n$] -
I : array identifier
E$_n$: number of elements in the array
letarray allocates uninitialised array I having E$_n$ cells.

Conflict A :
Syntax for assignment of array objects????

Confilct B :
Passing array element - by value or by reference?????

Conflict C : (basically conflict between A and C)
Whether a[0] refers to reading the array element or used for assigning value to it?????

So to resolve the above conflicts we define the following new syntax :
E:: ... | I [E_x] : it evaluates to a location and returns that location. Used in both read and write operations. (Just like FLK!)

| (:= I[E_x] E_r) : writes E_r to the location returned by I[E_x]

Semantic Domains

- **Two Environments**
  - **u :** Naming of single items
  - **w :** Array name $\rightarrow$ Array
- **Array**
  - **get** : Array $\rightarrow$ Int $\rightarrow$ Value
  - **set** : Array $\rightarrow$ Int $\rightarrow$ Value $\rightarrow$ ....

The get and set functions mentioned above access the array elements without going through location/store. But we want that the **(Array name + index)** should give us a location. Hence we dont use these, and define a new Environment **AEnv** as :
AEnv = ArrayId $\rightarrow$ Int $\rightarrow$ Loc
Now use this "Loc" to look up the standard store.

Thus the definition of ϵ changes as follows -
ϵ : Expr $\rightarrow$ Env $\rightarrow$ AEnv $\rightarrow$ Store $\rightarrow$ (ExpVal X Store)
where,
Env and AEnv are the environments for scalars and arrays.
The **store** refers to the "Common Store" .

Env - leads to one binding as a location (holding a scalar) is encountered.
AEnv - gets E_n bindings on encountering "letarray" .

reviewing **letarray I[E$_n$]** -

- It allocates E_n cells.
- Adds a bunch of new bindings to AEnv (Store).

Summarising the new **syntax** added :

E ::= ... | (letarray A[E$_{size}$] E_{body})
| (:= A[E$_{index}$] E$_{val}$)
| A[E$_{index}$]

Finally the **semantic domains** :

- **Index** = Int$^+$
- **AEnv** = ArrayId $\rightarrow$ Index $\rightarrow$ Loc
- ϵ : Expr $\rightarrow$ Env $\rightarrow$ AEnv $\rightarrow$ Store $\rightarrow$ (ExpVal X Store)

ϵ[(letarray A[E_{size}] E_{body})] = λu. λw. λs_0
let <e_{size} , s_1 > = (ϵ[E_{size}] u w s_0)

<u>newarray</u>
Inputs : A e_{size} s_1 w
Outputs : s_2 $w^{'}$

Where,
s_0 : old store
s_1 : store after evaluating ϵ[E_{size}]
s_2 : final store after evaluating **<u>newarray</u>**
w : old AEnv
$w^{'}$: new AEnv

Informally *<u>newarray</u>*,

- Creates output s_2 (new store) as all the cells of s_1 plus e_{size} new cells $\rightarrow$ l_1,l_2,...,l_{esize}
- Adds e_{size} new bindings to w giving $w^{'}$:
  (A,1) $\rightarrow$ l_1 ,
  (A,2) $\rightarrow$ l_2 ,... + bindings in w

let < $w^{'}$, s_2 > = (*<u>newarray</u>* A e_{size} s_1 w) (ϵ[E_{body}] u $w^{'}$ s_2)

ϵ[(:= A[E_{index}] E_{val})] = λu. λw. λs_0
let < e_{index} , s_1 > = (ϵ[E_{index}] u w s_0)
let < e_{val} , s_2 > = (ϵ[E_{val}] u w s_1)

It has to be specified, within the language, which will be evaluated first out of the above two.

(**<u>write</u>** s_2 (w A e_{index}) e_{val})
(w A e_{index}) returns a location.

Reading Operation :
ϵ[A[E_{index}]] = λu. λw. λs_0
let < e_{index} , s_1 > = (ϵ[E_{index}] u w s_0)
(**<u>read</u>** s_1 (w A e_{index}))

An example code in FLFL :

```
letarray a[2]
let i=0
begin
(:= a[0] 1)
```

(:= a[1] 0)
(call swap i a[i])

let swap = proc (x y)
begin
s1: (:= t x)
s2: (:= x y)
s3: (:= y t)

Now lets analyse the working of the code in the following 2 situations -

1. Pass - by - Reference **(Applicative order)** :

- if *f(g(...))* is given, then g(...) will be evaluated first to return a location.

As a result of call swap ,

- x $\rightarrow$ i (x = 0)
- y $\rightarrow$ a[0] (y = a[0] = 1)

After the instructions in swap are executed,

- i $\rightarrow$ 1
- a[0] $\rightarrow$ 0

as desired.

Assume a function (f i) increments i by 1 and returns the location pointed to by i. Now modify the call to swap as
(call swap (f i) a[i])
The order of evaluation of arguements must be specified by the language. Assuming left to right order, we get
x $\rightarrow$ i **(x = 1)**
y $\rightarrow$ **a[1]** (y = a[1] = 0)
So after the call to swap, the result is
a[1] = 1
i = 0

2. Normal order :

- in case of f(g(..)), g(..) will be passed to the parameters as it is, and will only be evaluated when the respective parameters are operated upon.

Here the execution will be as follows :

- x $\rightarrow$ i (x = 0)
- y $\rightarrow$ **a[i]** (hence, y = a[0] = 1 initially)

- step s1 : t $\rightarrow$ 0
- step s2 : i $\rightarrow$ 1 (as x $\rightarrow$ i , y = a [0] = 1)
- step s3 : as y $\rightarrow$ a[i], $\Rightarrow$ y $\rightarrow$ a[1] now

Therefore, output is a[0]=1, a[1]=0, i=1.

note :
The array elements

- must be mutable
- passed by reference

But whether we can pass the entire arrays as arguements, and if we can assign to entire arrays in one statement, still remain conflicting points.

Consider an example in C++:

```
void f (int y[5]) {
int *x = y; //this will work in C++as y is treated as &y[0]
int z[5];
z = y; //this wont work in C++as it doesnt allow implicit array copying.
}
```

Another example :

```
let p = (proc b (:= b[0] 3))
letarray a[2]
begin
(:= a[0] 1)
(:= a[1] 2)
(call p a)
a[0]
```

Here while evaluating (call p a), we have to decide how to evaluate passing of the whole array.