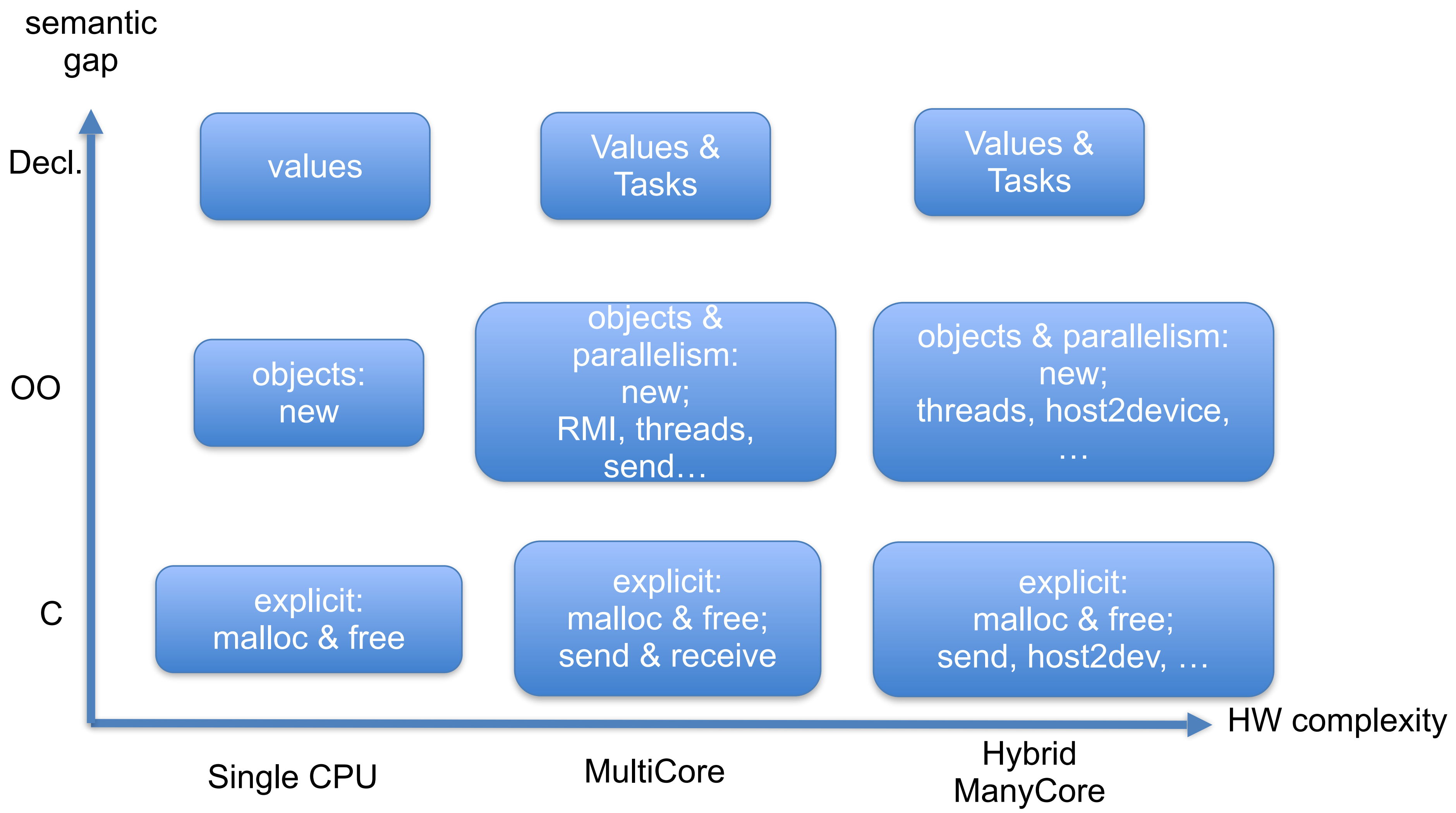
Advanced Programming
Memory Management

Peter Achten, Sven-Bodo Scholz

Software Science
Radboud University Nijmegen

Radboud University

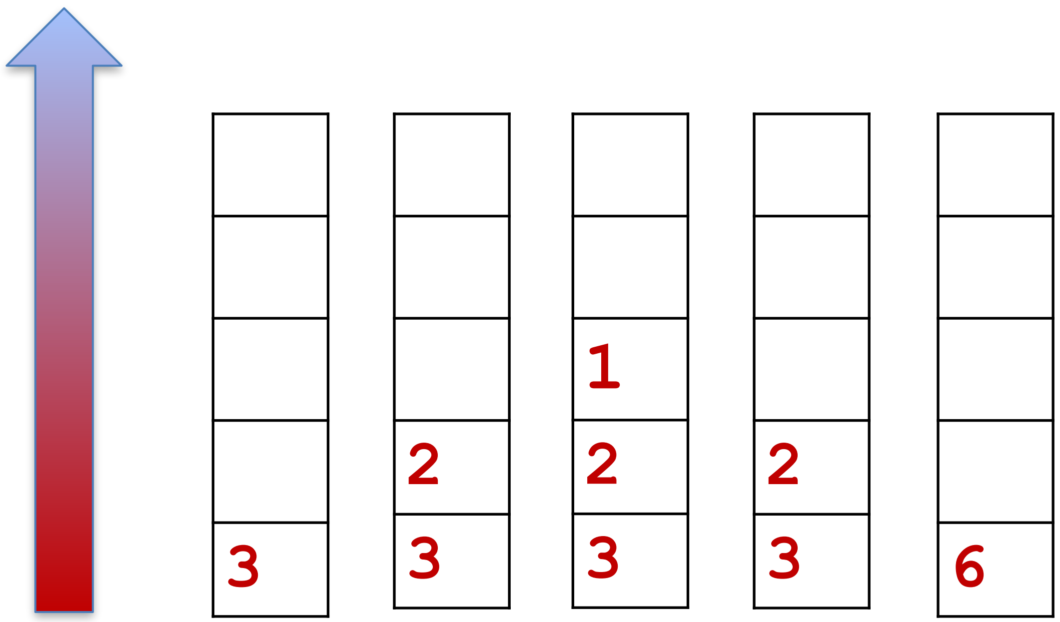
Languages & Memory Management



Why do we need memory management at all?

```
int factorial (int n)
{
   if (n==0)
      return 1;
   else
      return n*factorial (n-1);
}

…
res = factorial (3);
…
```

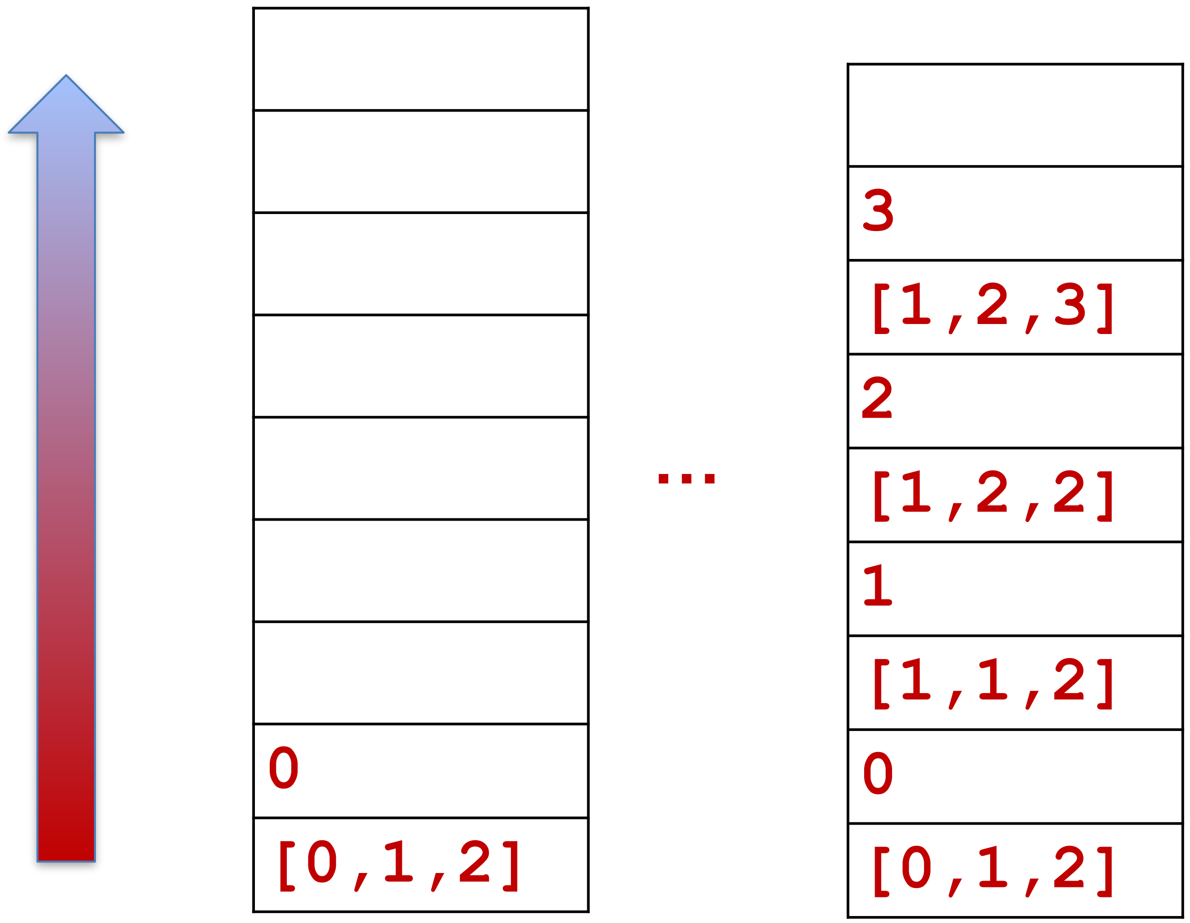


- ➢ Memory management is implicit!
- ➢ It all happens through the stack and registers!

Why do we need memory management at all?

```
int[3] inc (int[3] array, int pos)
{
   if (pos == 3)
      return array ;
   } else {
      array[pos] ++;
      return inc (array, pos+1);
   }
}

…
res = inc([0,1,2], 0);
…
```

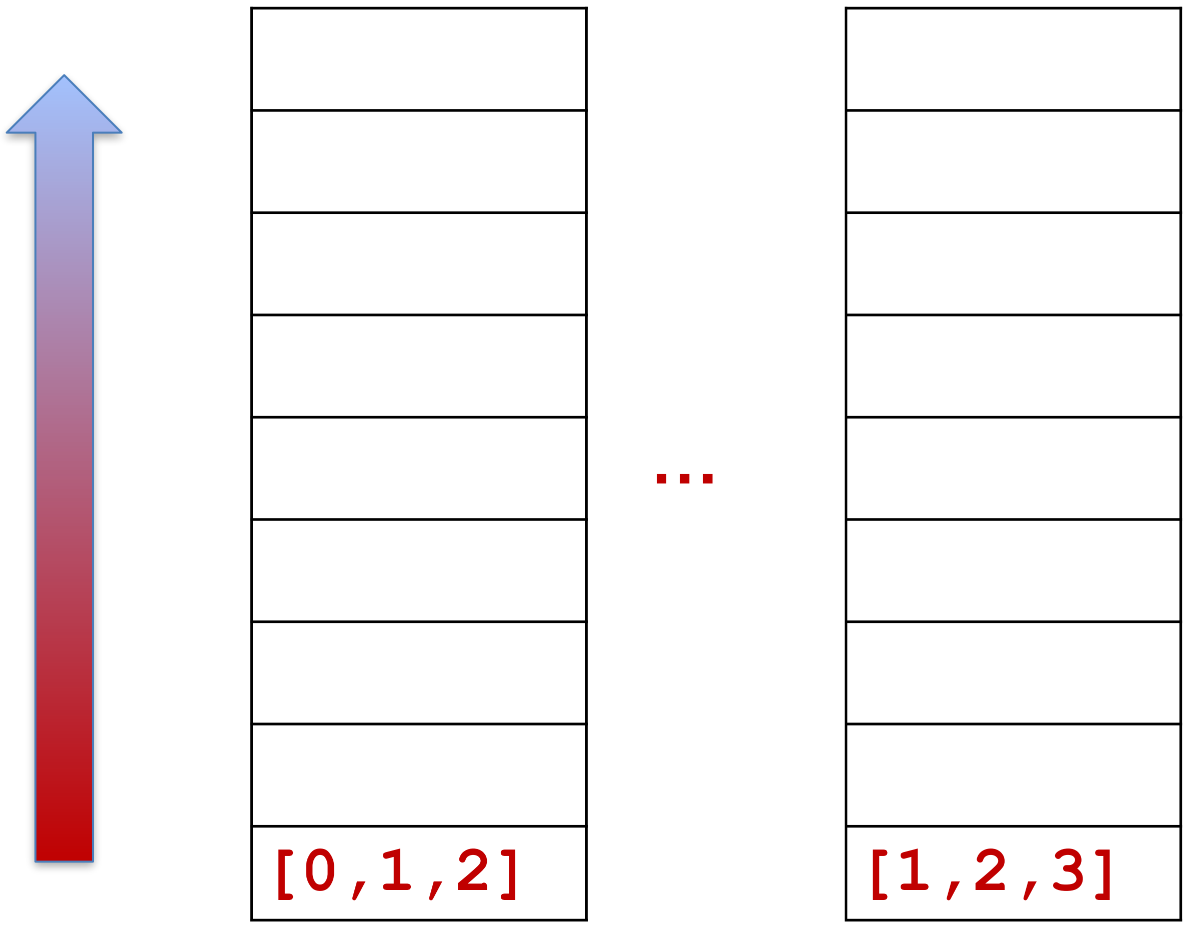


- ➢ too many copies!
- ➢ Stack optimisation needed!

Why do we need memory management at all?

```
int[3] increment (int[3] array)
{
   for( i=0; i<3; i++) {
      array[i] = array[i] + 1;
   }
   return array;
}

…
b = increment ([0,1,2]);
…
```



- ➢ great, BUT, now imagine int [10000] !
- ➢ Stack solution not suitable for large data structures!

How does C deal with memory?

```
int* increment( int* array)
{
   for( i=0; i<10000; i++) {
      array[i] = array[i] + 1;
   }
   return array;
}

void print( int* array)
{…}
```

always ‘in place’ !

always aliasing !

- It’s the programmer’s responsibility (alloc, copy & free)
- Potential space leaks
- Potential access errors (NULL pointer exceptions)

```
a = malloc (10000*sizeof(int));
a = increment (a);
b = increment (a);
print (a);
print (b);
free (a);
```

a and b are the same!

How does Rust deal with memory?

declare as ‘mutable’ !

```
fn increment(mut array: Vec<i64>) -> Vec<i64>
{
   for i in 0..9999 {
      array[i] = array[i] + 1;
   }
   return array;
}
```

move (transfer of ownership)

```
let a = vec![1;10000];
let a = increment (a);
let b = increment (a);

println!("{}", a[42]);
println!("{}", b[42]);
```

aliasing a with a (‘move`)!

aliasing a with b (‘move`)!

not allowed !

How does Rust deal with memory?

```
fn increment(mut array: Vec<i64>) -> Vec<i64>
{
   for i in 0..9999 {
      array[i] = array[i] + 1;
   }
   return array;
}
```

- It's the programmer's responsibility (alloc & copy)!
- no space leaks
- no access errors
- very similar to the UNQ types in SaC!

```
let a = vec![1;10000];
let a = increment (a);
let b = increment (a.clone());
print (&a);
print (&b);
```

explicit copying => no aliasing!

different values!

How does Java deal with memory?

```
public int[] increment( int[] array){
    for( i=0; i<10000; i++) {
        array[i] = array[i] + 1;
    }
    return array;
}

void print( int[] array)
{…}
```

always 'in place' !

always aliasing !

- It's the programmer's responsibility (alloc & copy)!
- no space leaks
- no access errors
- no different pointer types!
- Garbage Collection is required!

```
a = new int[10000];
a = increment (a);
b = increment (a);
print (a);
print (b);
```

a and b are the same!

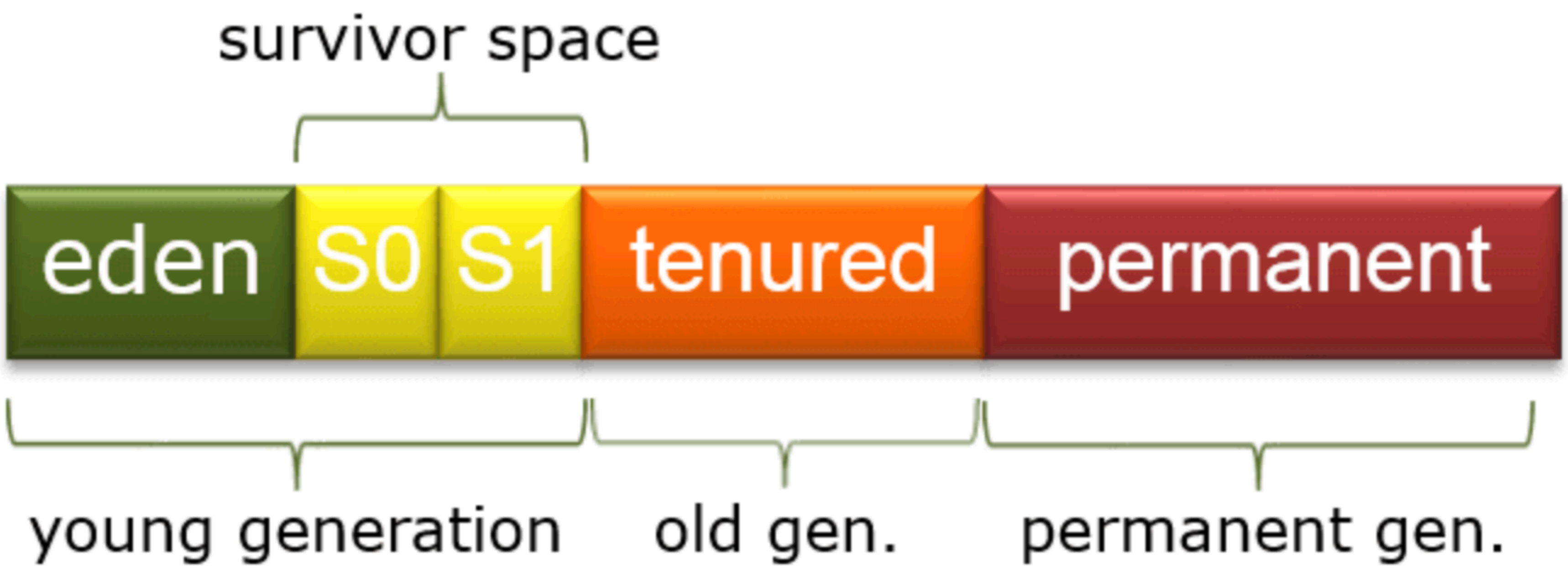
Mark and Sweep Garbage Collection

- Upon creation: mark all objects as false
- Mark Phase: starting from all root objects (stack): follow all pointers and mark as true
- Sweep Phase: delete all objects marked as false

➢ Mark is costly!
➢ Sweep is inherently sequential!

Generational Garbage Collection

- Idea: Most data is short lived; long-lived data tends to stay



- Minor GC: gc young generation only & move from survivor to old
- Major GC: gc all spaces; compact heap

Java GCs

- Serial - every GC sequential
- Parallel – minor GC parallel; Major sequential
- CMS (Concurrent Mark and Sweep) – run in parallel to application
- G1 (Garbage First) – run in parallel but use multiple separate heaps

How does SaC deal with memory?

sometimes 'in place' !

```
public int[.] increment( int[.] array){
    for( i=0; i<10000; i++) {
        array[i] = array[i] + 1;
    }
    return array;
}
```

sometimes aliasing !

```
void print( int[*] array)
{…}
```

- Fully implicit!
- no space leaks
- no access errors (-check c!)
- no pointer types at all!
- Garbage Collection required!
- Implicit copying required!

```
a = iota (10000);
a = increment (a);
b = increment (a);
print (a);
print (b);
```

different values!

Challenge: Memory Management: What does the λ-calculus teach us?

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
    ... a..... a....b.....b....c...
}
```

- Copy argument values into all free occurrences of the parameters in the body
- Consume (free) the argument values
- Built-In operations consume their arguments and create fresh results

How do we implement this? – the scalar case

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
    ... a..... a....b.....b....c...
}
```

operation	implementation
read	read from stack
funcall	push copy on stack

- The stack approach is perfect!
- Consumption is implicit through function scope!

How do we implement this? – the non-scalar case naive approach

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
    ... a..... a....b.....b....c...
}
```

operation	immediate copy
funcall	O(1) + (O(n) + malloc) * m + free
read-prf	O(1) + free
update-prf	O(1)
reuse (return)	O(1)

- ➢ Mimics the stack approach!
- ➢ Way to expensive!

How do we implement this? – the non-scalar case
widely adopted approach

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
    ... a..... a....b.....b....c...
}
```

operation	delayed copy + delayed GC
funcall	O(1)
read-prf	O(1)
update-prf	O(n) + malloc
reuse (return)	O(1)
garbage collection	O(stack+heap)

- every update requires a copy!
- GC is very expensive
- typically stalls the computation
- runtime depends on memory available

How do we implement this? – the non-scalar case reference counting approach

conceptual copies

f(a, b , c)

f(a, b ,c)
{
... a..... a....b.....b....c...
}

operation	delayed copy + non-delayed GC
funcall	O(1) + INC_RC
read-prf	O(1) + DEC_RC_FREE
update-prf	O(1) / O(n) + malloc
reuse (return)	O(1)
garbage collection	---

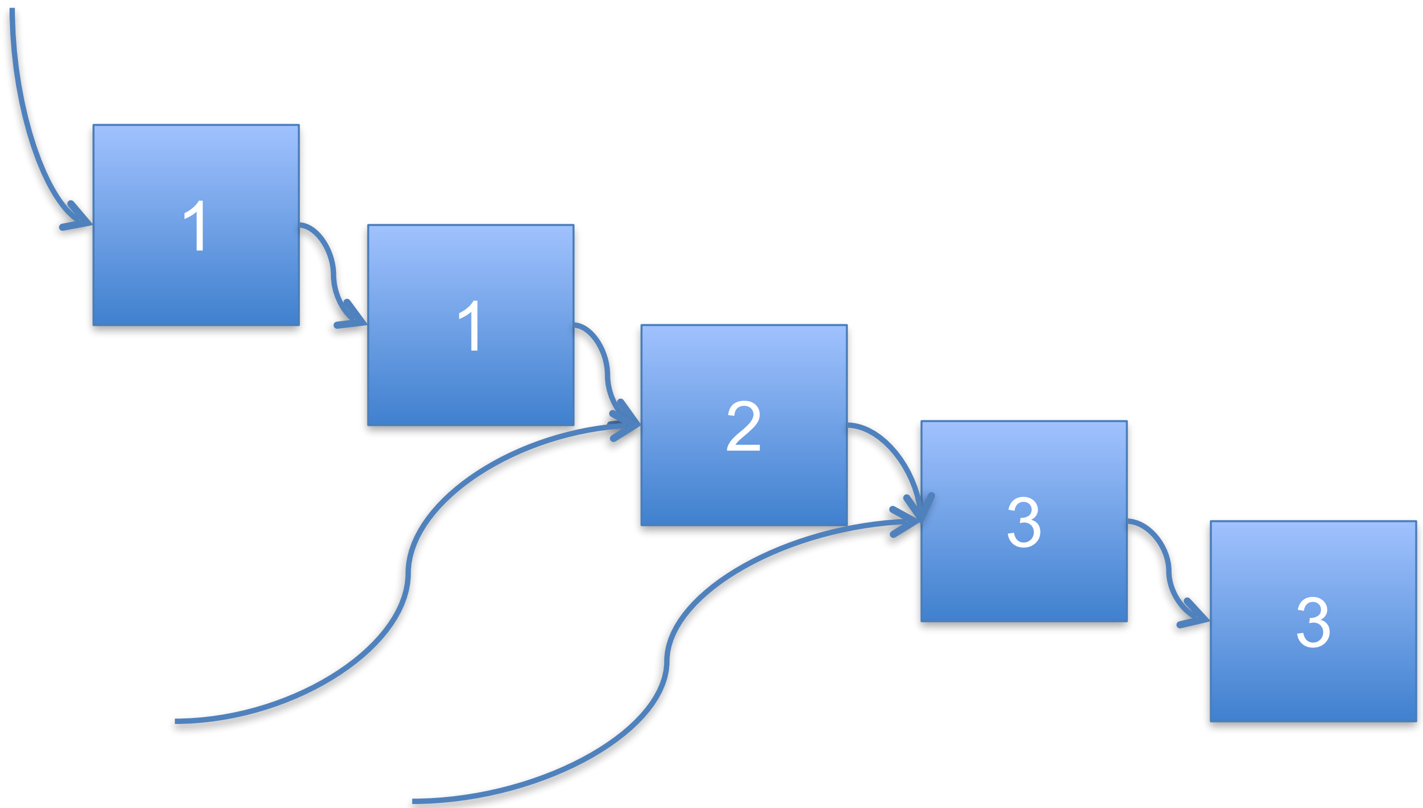
- ➢ Update in place is possible
- ➢ No stop-the-world garbage collection!
- ➢ “minimal” memory use

How do we implement this? – the non-scalar case a comparison of approaches

operation	non-delayed copy	delayed copy + delayed GC	delayed copy + non-delayed GC
funcall	O(1) + (O(n) + malloc) * m + free	O(1)	O(1) + INC_RC
read-prf	O(1) + free	O(1)	O(1) + DEC_RC_FREE
update-prf	O(1)	O(n) + malloc	O(1) / O(n) + malloc
reuse (return)	O(1)	O(1)	O(1)
GC	---	O(stack+heap)	---

NB: Why don't we have RC-world-domination?

- Overhead of INC_RC and DEC_RC only negligible for large, flat data structures!
- Circular data structures require extra support!



How to implement Reference Counting? — assignments

- We mimic beta-reductions!

⇒ functions consume their arguments,
⇒ and generate as many (virtual) copies as they have uses in the body!

```
a = [1,2,3,4];
```

RC([1,2,3,4]) == 1

RC(a) ≡ RC([1,2,3,4])

INC_RC (a, 2) Two occurrences in the body!

DEC_RC (a,1) Performing the β-reduction!

RC(a) == 2

```
return (a, a);
```

$(\ \lambda a.(a,\ a)\quad [1,2,3,4])$

How to implement Reference Counting? — built-in functions

- We mimic delta-reductions!

⇒ functions consume their arguments,
⇒ and produce fresh results!

RC(a) == n

```
a[0];
```

RC(“a[0]”) == 1

DEC_RC (a,1) Performing the δ-reduction!

RC(a) == n-1

```
sel ([0], a)
```

How to implement Reference Counting? — assignment block

```
a = [1,2,3,4];

b = a[0];

c = a*0;

return (b, c);
```

$(\lambda a. (\lambda b. (\lambda c. (b,c)\ (a*0))\ (a[0]))\ [1,2,3,4])$

INC_RC (a, 2) 2 occurrences in the body RC(a) == 1

DEC_RC (a, 1) β-reduction RC(a) == 2

$(\lambda b. (\lambda c. (b,c)\ (a*0))\ (a[0]))$

RC(b) == 1

DEC_RC (a,1) δ-reduction RC(a) == 1

$(\lambda b. (\lambda c. (b,c)\ (a*0))\ 1)$

INC_RC (b, 1) 1 occurrences in the body

DEC_RC (b, 1) β-reduction RC(b) == 1

$(\lambda c. (b,c)\ (a*0))$

RC(c) == 1

DEC_RC (a,1) δ-reduction RC(a) == 0

$(\lambda c. (b,c)\ [0,0,0,0])$

INC_RC (c, 1) 1 occurrences in the body

DEC_RC (c, 1) β-reduction

$(b,\ c)$

How to implement Reference Counting? — Functions

- We mimic beta-reductions!

```
int[.], int[.] fun ( int[.] a)
{  return (a, a); }
```

RC(a) == n

INC_RC (a,1)

RC(a) == n+1

pointers and RCs are shared!

```
a, b = fun (x);
```

RC(x) == n

RC(a) ≡ RC(b) ≡ RC(x) == n+1

```
… a … a … a … b … b
```

INC_RC (a,2); INC_RC (b,1);

RC(a) ≡ RC(b) ≡ RC(x) == n+4

DEC_RC (a,1) DEC_RC (a,1) DEC_RC (b,1)

DEC_RC (a,1) DEC_RC (b,1)

RC(a) ≡ RC(b) ≡ RC(x) == n-1

⇒ functions consume their arguments,

⇒ and generate as many (virtual) copies as they have uses in the body!

Reference Counting and Updates

- Situation 1:

RC(a) == 1

```
b = modarray (a, 0, 2.0);
```

update in place!

RC(a) ≡ RC(b) == 1

- Situation 2:

RC(a) == n > 1

```
b = modarray (a, 0, 2.0);
```

b = copy(a);
update b only!

RC(a) == n-1
RC(b) == 1

Compilation for Reference Counting

```
double[1000] increment (double[1000] a, int i, int n) {

    if (i<n) {
        val = a[i];










        b = modarray (a, i, val+1.0);
        return increment (b, i+1, n);
    } else {

        return a;
    }
}
```

Compilation for Reference Counting

Fun-call: max 2 occurrences in body; consume argument

```
double[1000] increment (double[1000] a, int i, int n) {
    INC_RC (a,1);
    if (i<n) {
        val = a[i];




        b = modarray (a, i, val+1.0);
        return increment (b, i+1, n);
    } else {
        DEC_RC (a,1);
        return a;
    }
}
```

Conditional: adjust a

Compilation for Reference Counting

Fun-call: max 2 occurrences in body; consume argument

Read-prf: consume a

Conditional: adjust a

```
double[1000] increment (double[1000] a, int i, int n) {
   INC_RC (a,1);
   if (i<n) {
      val = a[i];        // C-version of val = a[i]!
      DEC_RC (a,1);







      b = modarray (a, i, val+1.0);
      return increment (b, i+1, n);
   } else {
      DEC_RC (a,1);
      return a;
   }
}
```

Compilation for Reference Counting

Fun-call: max 2 occurrences in body; consume argument

Read-prf: consume a

Update-prf: Reuse or copy

Conditional: adjust a

```
double[1000] increment (double[1000] a, int i, int n) {
    INC_RC (a,1);
    if (i<n) {
        val = a[i];       // C-version of val = a[i]!
        DEC_RC (a,1);
        if (RC (a) == 1) {
            b = a;
        } else {
            b = malloc(); copy (a,b);
            SET_RC (b,1); DEC_RC (a,1);
        }
        b[i] = val+1.0;   // C-version of b = modarray(a, i, val+1.0)!
        return increment (b, i+1, n);
    } else {
        DEC_RC (a,1);
        return a;
    }
}
```

Avoiding Reference Counting Operations

```
double[1000] increment (double[1000] a, int i, int n) {
  INC_RC (a,1);
  if (i<n) {
     val = a[i];
     DEC_RC (a,1);
     if (RC (a) == 1) {
        b = a;
     } else {
        b = malloc(); copy (a,b);
        SET_RC (b,1); DEC_RC (a,1);
     }
     b[i] = val+1.0;
     return increment2  (b, i+1, n);
  } else {
     DEC_RC (a,1);
     return a;
  }
}
```

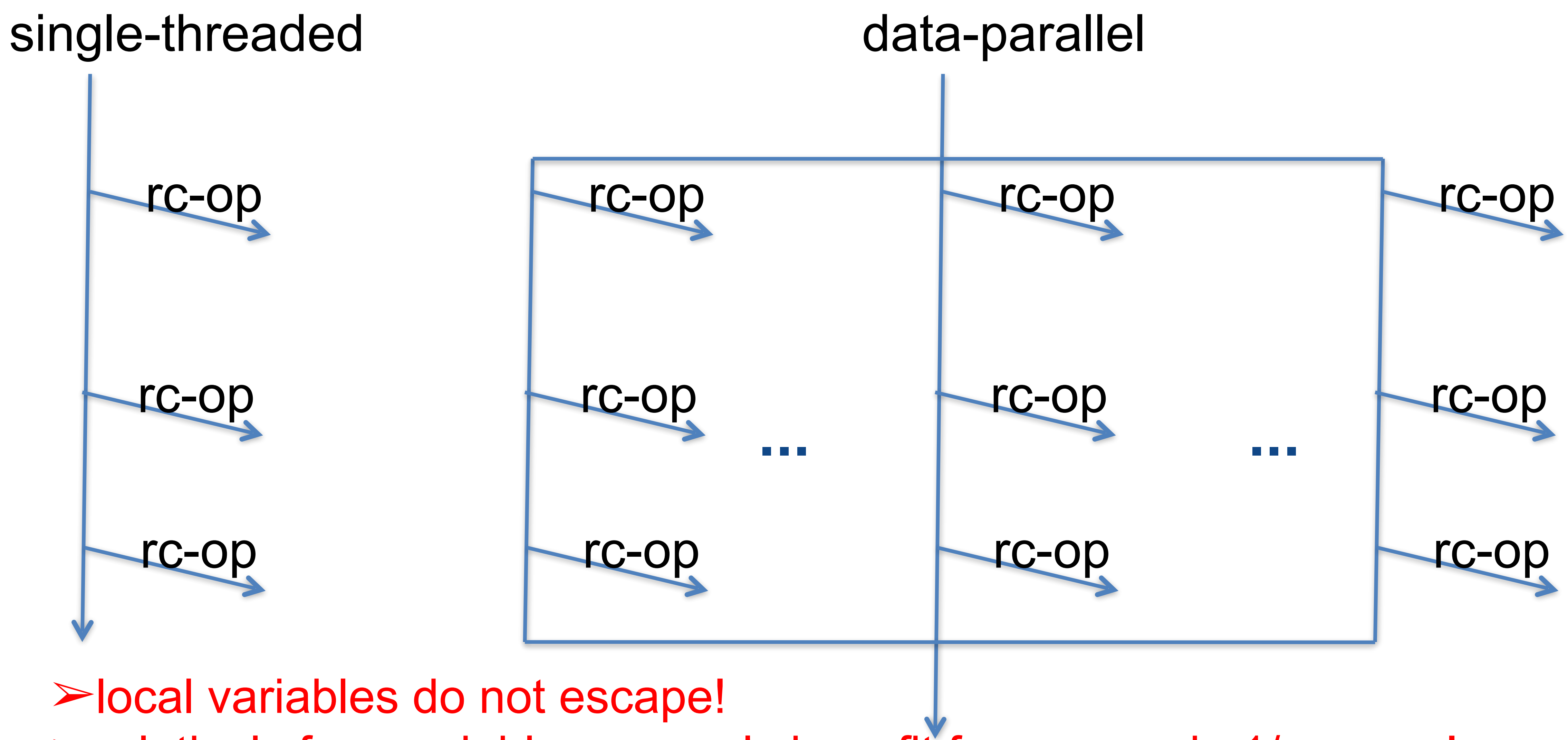
```
double[1000] increment2 (double[1000] a, int i, int n) {
  if (i<n) {
     val = a[i];
     if (RC (a) == 1) {
        b = a;
     } else {
        b = malloc(); copy (a,b);
        SET_RC (b,1); DEC_RC (a,1);
     }
     b[i] = val+1.0;
     return increment2 (b, i+1, n);
  } else {
     return a;
  }
}
```

Avoiding Reference Counting Operations

- Many more optimisations are possible, e.g.
  - Reorder built-in operations to maximise reuse
  - Classify and reorder user-defined functions
  - Extend memory life-time to avoid allocations
  - Uniqueness-inference driven optimisations
  - ….

NB: Looking for an interesting MSc thesis topic? ;-)

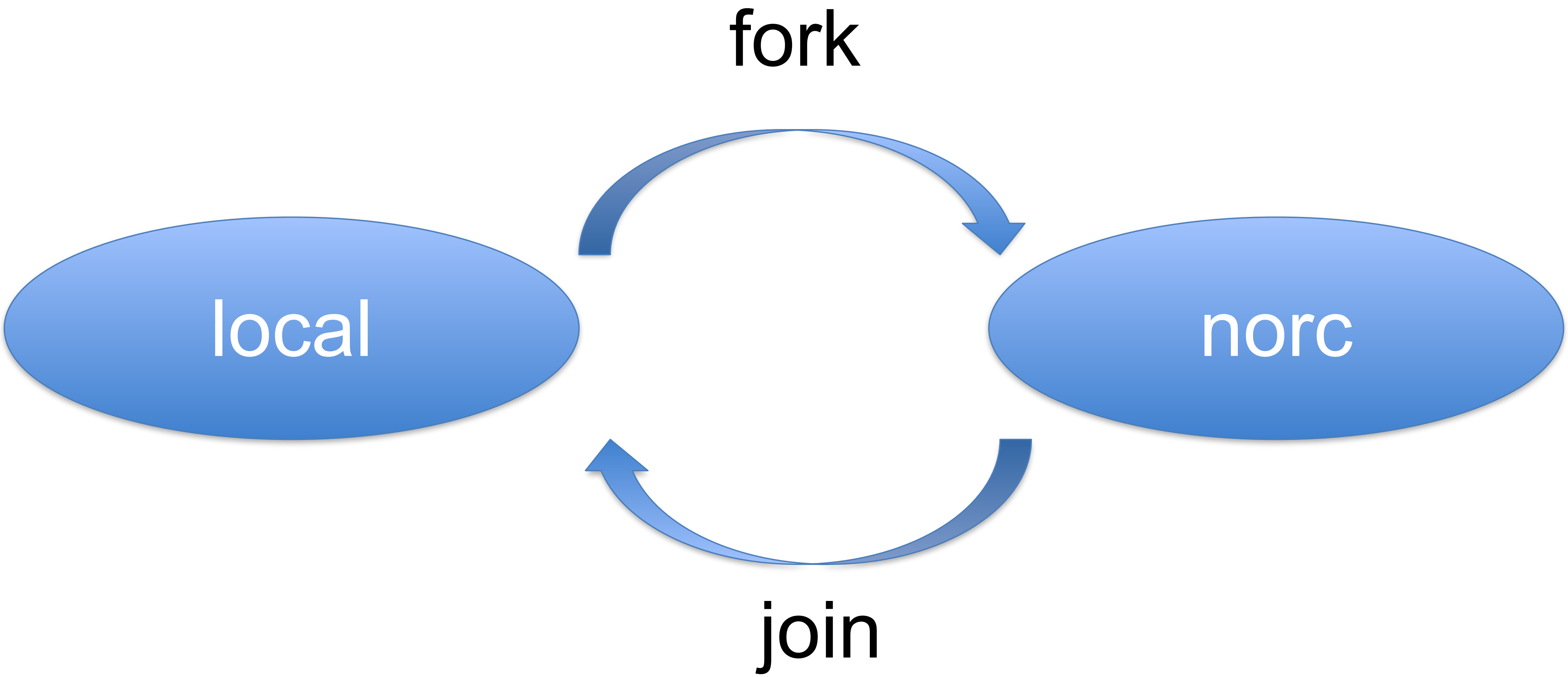
Going Multi-Core



- ➢local variables do not escape!
- ➢relatively free variables can only benefit from reuse in 1/n cases!

⇒use thread-local heaps ☺
⇒inhibit rc-ops on rel-free vars
⇒No locking required!

Bi-Modal RC:



- ➢local-mode: reference counting as usual
- ➢norc-mode: all reference count operations are mapped into no-ops
- ➢Mode switches occur before and after data parallel executions