

# Declarative Computation Model

## Memory management (VRH 2.5)

Carlos Varela

RPI

April 1, 2010

Adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

# Memory Management

- Semantic stack and store sizes during computation
  - analysis using operational semantics
  - recursion used for looping
    - efficient because of last call optimization
  - memory life cycle
  - garbage collection

# Last call optimization

- Consider the following procedure

```
proc {Loop10 I}
  if I == 10 then skip
  else
    {Browse I}
    {Loop10 I+1}
  end
end
```

Recursive call  
is the last call

- This procedure does **not** increase the size of the STACK
- It behaves like a looping construct

# Last call optimization

```
proc {Loop10 I}
  if I == 10 then skip
  else
    {Browse I}
    {Loop10 I+1}
  end
end
```

ST: [ ( {Loop10 0},  $E_0$  ) ]

ST: [ ( {Browse I}, {I  $\rightarrow$   $i_0, \dots$  } )  
 ( {Loop10 I+1}, {I  $\rightarrow$   $i_0, \dots$  } ) ]

$\sigma$  : {  $i_0=0, \dots$  }

ST: [ ( {Loop10 I+1}, {I  $\rightarrow$   $i_0, \dots$  } ) ]

$\sigma$  : {  $i_0=0, \dots$  }

ST: [ ( {Browse I}, {I  $\rightarrow$   $i_1, \dots$  } )  
 ( {Loop10 I+1}, {I  $\rightarrow$   $i_1, \dots$  } ) ]

$\sigma$  : {  $i_0=0, i_1=1, \dots$  }

# Stack and Store Size

```
proc {Loop10 I}
  if I == 10 then skip
  else
    {Browse I}
    {Loop10 I+1}
  end
end
```

ST: [({Browse I}, {I $\rightarrow$ i<sub>k</sub>,...})  
( {Loop10 I+1}, {I $\rightarrow$ i<sub>k</sub>,...} ) ]  
 $\sigma$  : {i<sub>0</sub>=0, i<sub>1</sub>=1,..., i<sub>k-1</sub>=k-1, i<sub>k</sub>=k,... }

The semantic stack size is bounded by a constant.

But the store size keeps increasing with the computation.

Notice that at (k+1)<sup>th</sup> recursive call, we only need i<sub>k</sub>

If we can keep the store size constant, we can run indefinitely with a constant memory size.

# Garbage collection

```
proc {Loop10 I}  
  if I == 10 then skip  
  else  
    {Browse I}  
    {Loop10 I+1}  
  end  
end
```

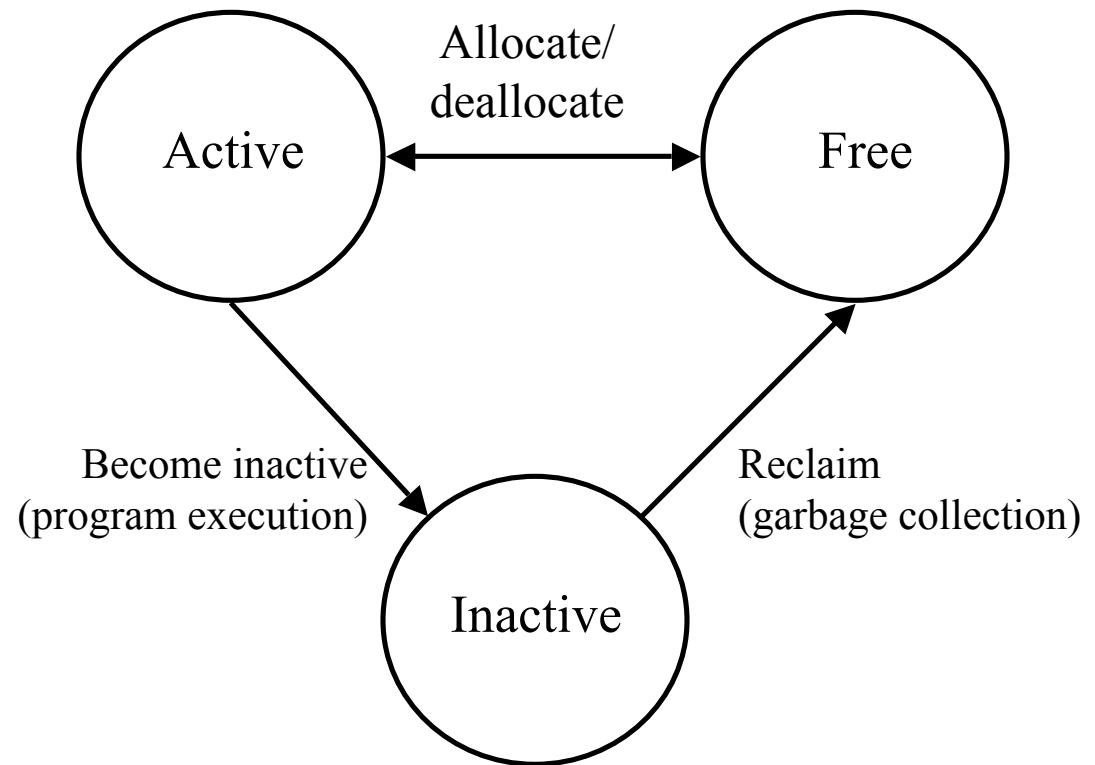
ST: [( {Browse I}, {I→ $i_k, \dots$ })  
( {Loop10 I+1}, {I→ $i_k, \dots$ }) ]  
 $\sigma$  : {  $i_0=0, i_1=1, \dots, i_{k-i}=k-1, i_k=k, \dots$  }

**Garbage collection** is an algorithm (a task) that removes from memory (store) all cells that are not accessible from the stack

ST: [( {Browse I}, {I→ $i_k, \dots$ })  
( {Loop10 I+1}, {I→ $i_k, \dots$ }) ]  
 $\sigma$  : {  $i_k=k, \dots$  }

# The memory life cycle

- **Active memory** is what the program needs to continue execution (semantic stack + reachable part of store)
- Memory that is no longer needed is of two kinds:
  - Can be immediately **deallocated** (i.e., semantic stack)
  - Simply becomes **inactive** (i.e., store)
- Reclaiming inactive memory is the hardest part of memory management
  - **Garbage collection** is automatic reclaiming



# Garbage Collection

- Lower-level languages (C, C++) do not have automatic garbage collection.
- Manual memory management can be more efficient but it is also more error-prone, e.g.:
  - Dangling references
    - Reclaiming reachable memory blocks
  - Memory leaks
    - Not reclaiming unreachable memory blocks
- Higher-level languages (Erlang, Java, Lisp, Smalltalk) typically have automatic garbage collection.
- Modern algorithms are efficient enough---minimal memory and time penalties.



# Garbage Collection Algorithms

- Reference Counting algorithms
  - Keep track of number of references to memory blocks
  - When count is 0, memory block is reclaimed.
  - Cannot collect cycles of garbage.
- Mark-and-Sweep algorithms
  - Phase 1: Determine active memory
    - Following *pointers* (in Oz, referenced store variables) from a *root set* (in Oz, the semantic stack).
  - Phase 2: Compact memory in one contiguous region.
    - Everything outside this region is free.
  - Generally must briefly pause the application memory mutation while collecting.

# Avoiding memory leaks

- Consider the following function

```
fun {Sum X L1 L}
  case L1 of Y|L2 then {Sum X+Y L2 L}
  else X end
end
local L in
  L = [1 2 3 ... 1000000]
  {Sum 0 L L}
end
```

- Since it keeps a pointer to the original list L, L will stay in memory during the whole execution of Sum.

# Avoiding memory leaks

- Consider the following function

```
fun {Sum X L1}
  case L1 of Y|L2 then {Sum X+Y L2}
  else X end
end
local L in
  L = [1 2 3 ... 1000000]
  {Sum 0 L}
end
```

- Here, the reference to L is lost immediately and its space can be collected as the function executes.

# Managing external references

- External resources are data structures outside the current O.S. process.
- There can be pointers from internal data structures to external resources, e.g.
  - An open file in a file system
  - A graphic entity in a graphics display
  - If the internal data structure is reclaimed, then the external resource needs to be cleaned up (e.g., remove graphical entity, close file)
- There can be pointers from external resources to internal data structures, e.g.
  - A database server
  - A web service
  - If the internal data structure is reachable from the outside, it should not be reclaimed.

# Local Mozart Garbage Collector

- Copying dual-space algorithm
- Advantage : Execution time is proportional to the active memory size, not total memory size.
- Disadvantage : Half of the total memory is unusable at any given time

# Exercises

55. What do you expect to happen if you try to execute the following statement? Try to answer without actually executing it!

```
local T = tree(key:A left:B right:C value:D) in
  A = 1
  B = 2
  C = 3
  D = 4
end
```

56. VRH Exercise 2.9.9 (page 109).
57. Any realistic computer system has a memory cache for fast access to frequently used data. Can you think of any issues with garbage collection in a system that has a memory cache?