Automatic Memory Management in newLISP

Lutz Mueller, 2004-2015. Last edit 2013-11-07

ORO (One Reference Only) automatic memory management developed for newLISP is a fast and resources saving alternative to classic garbage collection algorithms in dynamic, interactive programming languages. This article explains how ORO memory management works

newLISP and any other interactive language system will constantly generate new memory objects during expression evaluation. The new memory objects are intermediate evaluation results, reassigned memory objects, or memory objects whose content was changed. If newLISP did not delete some of the objects created, it would eventually run out of available memory.

In order to understand newLISP's automatic memory management, it is necessary to first review the traditional methods employed by other languages.

Traditional automatic memory management (Garbage Collection)

In most programming languages, a process registers allocated memory, and another process finds and recycles the unused parts of the allocated memory pool. The recycling process can be triggered by some memory allocation limit or can be scheduled to happen between evaluation steps. This form of automatic memory management is called *Garbage Collection*.

Traditional garbage collection schemes developed for LISP employed one of two algorithms: [1]

(1) The *mark-and-sweep* algorithm registers each allocated memory object. A mark phase periodically flags each object in the allocated memory pool. A named object (a variable symbol) directly or indirectly references each memory object in the system. The sweep phase frees the memory of the marked objects when they are no longer in use.

(2) A *reference-counting* scheme registers each allocated memory object together with a count of references to the object. This reference count gets incremented or decremented during expression evaluation. Whenever an object's reference count reaches zero, the object's allocated memory is freed.

Over time, many elaborate garbage collection schemes have been attempted based on these principles. The first garbage collection algorithms appeared in LISP. The inventors of the Smalltalk language used more elaborate garbage collection schemes. The history of Smalltalk-80 is an exciting account of the challenges of implementing memory management in an interactive programming language; see [Glenn Krasner, 1983: *Smalltalk-80, Bits of History, Words of Advice*]. A more recent overview of garbage collection methods can be found in [Richard Jones, Rafael Lins, 1996: *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*].

One reference only, (ORO) memory management

Memory management in newLISP does not rely on a garbage collection algorithm. Memory is not marked or reference-counted. Instead, a decision whether to delete a newly created memory object is made right after the memory object is created.

Empirical studies of LISP have shown that most LISP cells are not shared and so can be reclaimed during the evaluation process. Aside from some optimizations for part of the built-in functions, newLISP deletes memory new objects containing intermediate evaluation results once it reaches a higher evaluation level. newLISP does this by pushing a reference to each created memory object onto a result stack. When newLISP reaches a higher evaluation level, it removes the last evaluation result's reference from the result stack and deletes the evaluation result's memory object. This should not be confused with one-bit reference counting. ORO memory management does not set bits to mark objects as *sticky*.

newLISP follows a one reference only (ORO) rule. Every memory object not referenced by a symbol is obsolete once newLISP reaches a higher evaluation level during expression evaluation. Objects in newLISP (excluding symbols and contexts) are passed by value copy to other user-defined functions. As a result, each newLISP object only requires one reference.

newLISP's ORO rule has advantages. It simplifies not only memory

management but also other aspects of the newLISP language. For example, while users of traditional LISP have to distinguish between equality of copied memory objects and equality of references to memory objects, newLISP users do not.

newLISP's ORO rule forces newLISP to constantly allocate and then free LISP cells. newLISP optimizes this process by allocating large chunks of cell memory from the host operating system. newLISP will request LISP cells from a free cell list and then recycle those cells back into that list. As a result, only a few CPU instructions (pointer assignments) are needed to unlink a free cell or to re-insert a deleted cell.

The overall effect of ORO memory management is a faster evaluation time and a smaller memory and disk footprint than traditional interpreted LISP's can offer. Time spent linking and unlinking memory objects is more than compensated for by the lack of processing time used in traditional garbage collection. ORO memory management also avoids occasional processing pauses seen in languages using traditional garbage collection and the tuning of garbage collection parameters required when running memory intensive programs.

ORO memory management happens synchronous to other processing in the interpreter, which results in deterministic processing times.

In versions before 10.1.3, newLISP employed a classic mark and sweep algorithm to free un-referenced cells under error conditions. Starting version 10.1.3, this has been eliminated and replaced by a proper cleanup of the result stack under error conditions.

Performance considerations with copying parameters

In theory, passing parameters to user-defined functions by value (memory copying) instead by reference poses a potential disadvantage when dealing with large lists, arrays or strings. But in practice newLISP performs faster or as fast than other scripting languages and offers language facilities to pass very large memory object by reference.

Since newLISP version 9.4.5 functions can pass list, array and string type parameters as references using *default functor* namespace ids. Namespaces (called contexts in newLISP) have very little overhead and can be used to wrap functions and data. This allows reference passing of large memory object into user-defined functions.

Since version 10.2 FOOP (Functional Object Oriented Programming) in newLISP also passes the target object of a method call by reference.

But even in instances where reference passing and other optimizations are nor present, the speed of ORO memory management more than compensates for the overhead required to copy and delete objects.

Optimizations to ORO memory management [2]

Since newLISP version 10.1, all lists, arrays and strings are passed in and out of built-in functions by reference. All built-in functions work directly on memory objects returned by reference from other built-in functions. This has substantially reduced the need for copying and deleting memory objects and increased the speed of some built-in functions. Now only parameters into user-defined functions and return values passed out of user-defined functions are ORO managed.

Since version 10.3.2, newLISP checks the result stack before copying LISP cells. This has reduced the amount of cells copied by about 83% and has significantly increased the speed of many operations on bigger lists.

Memory and datatypes in newLISP

The memory objects of newLISP strings are allocated from and freed to the host's OS, whenever newLISP recycles the cells from its allocated chunks of cell memory. This means that newLISP handles cell memory more efficiently than string memory. As a result, it is often better to use symbols rather than strings for efficient processing. For example, when handling natural language it is more efficient to handle natural language words as individual symbols in a separated name-space, then as single strings. The `bayes-train` function in newLISP uses this method. newLISP can handle millions of symbols without degrading performance.

Programmers coming from other programming languages frequently overlook that symbols in LISP can act as more than just variables or object references. The symbol is a useful data type in itself, which in many cases can replace the string data type.

Integer numbers and double floating-point numbers are stored directly in newLISP's LISP cells and do not need a separate memory

allocation cycle.

For efficiency during matrix operations like matrix multiplication or inversion, newLISP allocates non-cell memory objects for matrices, converts the results to LISP cells, and then frees the matrix memory objects.

newLISP allocates an array as a group of LISP cells. The LISP cells are allocated linearly. As a result, array indices have faster random access to the LISP cells. Only a subset of newLISP list functions can be used on arrays. Automatic memory management in newLISP handles arrays in a manner similar to how it handles lists.

Implementing ORO memory management

The following pseudo code illustrates the algorithm implemented in newLISP in the context of LISP expression evaluation. Only two functions and one data structure are necessary to implement ORO memory management:

```
function pushResultStack(evalationResult)

function popResultStack() ; implies deleting

array resultStack[] ; preallocated stack area
```

The first two functions `pushResultStack` and `popResultStack` push or pop a LISP object handle on or off a stack. `pushResultStack` increases the value `resultStackIndex` while `popResultStack` decreases it. In newLISP every object is contained in a LISP cell structure. The object handle of that structure is simply the memory pointer to the cell structure. The cell itself may contain pointer addresses to other memory objects like string buffers or other LISP cells linked to the original object. Small objects like numbers are stored directly. In this paper function `popResultStack()` also implies that the popped object gets deleted.

The two `resultStack` management functions described are called by newLISP's `evaluateExpression` function: [3]

```
function evaluateExpression(expr)
    {
    resultStackIndexSave = resultStackIndex

    if typeOf(expr) is BOOLEAN or NUMBER or STRING
                return(expr)

    if typeOf(expr) is SYMBOL
        return(symbolContents(expr))
```

```
if typeOf(expr) is QUOTE
    return(quoteContents(expr))

if typeOf(expr) is LIST
    {
    func = evaluateExpression(firstOf(expr))
    args = rest(expr)
    if typeOf(func) is BUILTIN_FUNCTION
            result = evaluateFunc(func, args)
    else if typeOf(func) = LAMBDA_FUNCTION
            result = evaluateLambda(func, args)
    }
}

while (resultStackIndex > resultStackIndexSave)
    deleteList(popResultStack())

pushResultStack(result)

return(result)
}
```

The function `evaluateExpression` introduces the two variables `resultStackIndexSave` and `resultStackIndex` and a few other functions:

- `resultStackIndex` is an index pointing to the top element in the `resultStack`. The deeper the level of evaluation the higher the value of `resultStackIndex`.

- `resultStackIndexSave` serves as a temporary storage for the value of `resultStackIndex` upon entry of the `evaluateExpression(func, args)` function. Before exit the `resultStack` is popped to the saved level of `resultStackIndex`. Popping the `resultStack` implies deleting the memory objects pointed to by entries in the `resultStack`.

- `resultStack[]` is a preallocated stack area for saving pointers to LISP cells and indexed by `resultStackIndex`.

- `symbolContents(expr)` and `quoteContents(expr)` extract contents from symbols or quote-envelope cells.

- `typeOf(expr)` extracts the type of an expression, which is either a `BOOLEAN` constant like `nil` or `true` or a `NUMBER` or `STRING`, or is a variable `SYMBOL` holding some contents, or a `QUOTE` serving as an envelope to some other `LIST` expression `expr`.

- `evaluateFunc(func, args)` is the application of a built-in function to its

arguments. The built-in function is the evaluated first member of a list in `expr` and the arguments are the `rest` of the list in `expr`. The function `func` is extracted calling `evaluateExpression(first(expr))` recursively. For example if the expression (`expr` is `(foo x y)` than `foo` is a built-in function and `x` and `y` are the function arguments or parameters.

- `evaluateLambda(func, args)` works similar to `evaluateFunc(func, args)`, applying a user-defined function `first(expr)` to its arguments in `rest(expr)`. In case of a user-defined function we have two types of arguments in `rest(expr)`, a list of local parameters followed by one or more body expressions evaluated in sequence.

Both, `evaluateFunc(func, args)` and `evaluateLambda(func, args)` will return a newly created or copied LISP cell object, which may be any type of the above mentioned expressions. Since version 10.0, many built-in functions processed with `evaluateFunc(func, args)` are optimized and return references instead of a newly created or copied objects. Except for these optimizations, `result` values will always be newly created LISP cell objects destined to be destroyed on the next higher evaluation level, after the current `evaluateExpression(expr)` function execution returned.

Both functions will recursively call `evaluateExpression(expr)` to evaluate their arguments. As recursion deepens, the recursion level of the function increases.

Before `evaluateExpression(func, args)` returns, it will pop the `resultStack` deleting the `result` values from deeper level of evaluation and returned by one of the two functions, either `evaluateFunc` or `evaluateLambda`.

Any newly created `result` expression is destined to be destroyed later but its deletion is delayed until a higher, less deep, level of evaluation is reached. This permits results to be used and/or copied by calling functions.

The following example shows the evaluation of a small user-defined LISP function `sum-of-squares` and the creation and deletion of associated memory objects:

```
(define (sum-of-squares x y)
        (+ (* x x) (* y y)))

(sum-of-squares 3 4) => 25
```

`sum-of-squares` is a user-defined *lambda-function* calling to *built-in*

functions + and *.

The following trace shows the relevant steps when defining the sum-of-squares function and when executing it with the arguments 3 and 4.

```
> (define (sum-of-squares x y) (+ (* x x) (* y y)))

level 0: evaluateExpression( (define (sum-of-squares x y)
 (+ (* x x) (* y y))) )
level 1: evaluateFunc( define <6598> )
level 1: return( (lambda (x y) (+ (* x x) (* y y))) )

→ (lambda (x y) (+ (* x x) (* y y)))

> (sum-of-squares 3 4)

level 0: evaluateExpression( (sum-of-squares 3 4) )
level 1:   evaluateLambda( (lambda (x y) (+ (* x x) (* y y))), (3 4) )
level 1:   evaluateExpression( (+ (* x x) (* y y)) )
level 2:     evaluateFunc( +, ((* x x) (* y y)) )
level 2:     evaluateExpression( (* x x) )
level 3:       evaluateFunc( *, (x x) )
level 3:       pushResultStack( 9 )
level 3:       return( 9 )
level 2:     evaluateExpression( (* y y) )
level 3:       evaluateFunc( *, (y y) )
level 3:       pushResultStack( 16 )
level 3:       return( 16 )
level 2:     popResultStack() → 16
level 2:     popResultStack() → 9
level 2:     pushResultStack( 25 )
level 2:     return( 25 )
level 1:   return( 25 )

→ 25
```

The actual C-language implementation is optimized in some places to avoid pushing the resultStack and avoid calling evaluateExpression(expr). Only the most relevant steps are shown. The function evaluateLambda(func, args) does not need to evaluate its arguments 3 and 4 because they are constants, but evaluateLambda(func, args) will call evaluateExpression(expr) twice to evaluate the two body expressions (+ (* x x) and (+ (* x x). Lines preceded by the prompt > show the command-line entry.

evaluateLambda(func, args) also saves the environment for the variable symbols x and y, copies parameters into local variables and restores the old environment upon exit. These actions too involve creation and deletion of memory objects. Details are omitted, because they are similar to methods in other dynamic languages.

References

– Glenn Krasner, 1983: *Smalltalk-80, Bits of History, Words of Advice*
Addison Wesley Publishing Company

– Richard Jones, Rafael Lins, 1996: *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*
John Wiley & Sons

[1] Reference counting and mark-and-sweep algorithms where specifically developed for LISP. Other schemes like copying or generational algorithms where developed for other languages like Smalltalk and later also used in LISP.

[2] This chapter was added in October 2008 and extended August 2011.

[3] This is a shortened rendition of expression evaluation not including handling of default functors and implicit indexing. For more information on expression evaluation see: Expression evaluation, Implicit Indexing, Contexts and Default Functors in the newLISP Scripting Language.

Copyright © 2004-2015, Lutz Mueller http://newlisp.org. All rights reserved.