Dynamic Extent Objects

Lisp Pointers, 2(1) July-August-September 1988, pp 11-21.

Christian Queinnec
UUCP: ..!mcvax!inria!litp!queinnec
INTERNET: queinnec@litp.unip6-7.fr
Greco de Programmation

Abstract

Lisp objects are heap- rather than stack-allocated because their extent is generally indefinite. Since stack-deallocation is performed without running the garbage collector, speed improvements are expected. Dedicated hardware can stack-allocate objects associated with reference counters or microcode daemons such that one can exactly know the status of the object to be deallocated and perform whatever appropriate treatment (usually a copy in the heap) according to its reachability. However such a solution is not efficient on stock hardware.

This paper presents and analyses a new technique to efficiently solve this problem. It creates first class dynamic extent objects (DEO) that are stack-allocated, permits to access them only while they are in the stack and prevent to follow up dangling pointers. Every usual indefinite extent object has its counterpart as a DEO associated with the same set of operators. Finally DEO do not require dedicated hardware and may be properly compiled. The technique can be used within other programming languages. Some performance figures are discussed at the end of the paper.

1 Introduction

Lisp world is two-fold: there is a stack that contains continuations and pieces of environments, and a heap where are allocated almost all objects (cons-cells, strings, arrays, code ...). Since objects extent is usually indefinite and cannot be simply bounded, they are heap-allocated rather than stack-allocated; deallocation is then left to the Garbage Collector. A typical `let` form such as

```
(let ((temp (foo)))
     (bar temp temp) )
```

seems to introduce a stack discipline but that depends of the meaning of `bar`. For example:

```
(define (foo) ;computes the 10^10^10^th decimal of π
   ... )
(define (bar n p)
   (print n)(print p)
   t )
(let ((temp (foo)))
     (bar temp temp) )
```

is stack obedient while

```
(define (foo)
   (list 1 2 3 4) )
(define (bar n p)
   (nconc n p) )
```

```
(let ((temp (foo)))
     (bar temp temp) )
```

is definitely not. In the first case, after exiting the `let` form, the value of `temp` can be scavenged since nothing can point on it: `temp` can then be stack-allocated. In the second case since the value of `temp` is part of the value of the `let` form, its extent cannot be known from the sole inspection of the `let` body nor from the body of `bar`, it must probably be heap-allocated. One cannot stack-allocate it since subsequent growth of the stack will overwrite the value. Moreover the recently freed part of the stack cannot be guaranteed to remain unchanged since garbage collector or interrupts often use the complement of the active part of the stack.

However stack-allocation is efficient and widely used in classical programming languages as C or Pascal where the prevention of dangling pointers is left to the user: a fact Lisp cannot agree with ! Nevertheless some compilers can stack-allocate objects which can be proved to obey a strict stack discipline. Among others ORBIT [3] can stack-allocate contexts of some closures (typically `lambda` in `map` forms) while Symbolics Common Lisp [11] offers stack-lists and stack-allocated variant of `&rest` [9]. Moreover dynamic extent entities exist in Lisp. These are special bindings from identifiers to values or from labels to continuations. One can hope to extend this behaviour to more tangible objects.

Compared to incremental or concurrent GC, using dynamic extent object is rather cheap and provides a valuable alternative. Moreover stack discipline is approximatively respected except that scope and size of resources needed for a computation are difficult to foresee.

Our aim is to describe an efficient way to access, stack-allocate and stack-deallocate first class dynamic extent objects providing a good compilability, avoiding birth of dangling pointers and, substantially better, detecting any attempt to follow a reference to obsolete object. DEO must support the same set of operations that their indefinite extent counterpart support and access to DEO must not be restricted to a lexical scope which restrict their use.

To present our proposition, we first define a denotational semantics for a tiny Lisp Kernel offering a new special form, similar to `let`, that allocates a cons-cell in the stack, binds it to a local name and evaluates the body in that new environment. The cons-cell is deallocated at the end of the body without any assumptions to the final value of this body. This scheme is not very interesting with cons-cells but can easily be extended to vectors, structures or bounded strings ... The second part presents a raw implementation of this new special form that will be enhanced in a third part. In the fourth part will be discussed some performances.

2 The Lisp Kernel

To simply express the technique we will use a small Lisp Kernel in the spirit of Scheme [8]. It will be defined by a denotational semantics [10]. The equations will use an environment ρ mapping identifiers to locations, a store σ mapping locations to values and a continuation κ which encodes the rest of computations. We do not give full equations and restrict ourselves to what is really necessary. The notation is summarised below

$\mathcal{E}(\pi)$	The meaning of the form π
$\rho[x \leftarrow i]$	Substitution "ρ with x for i"
$<\epsilon_1, \epsilon_2>$	Binary sequence formation

The function $\mathcal{E}$ is not explicitely defined. $\mathcal{E}[\![\pi]\!]$ invokes after the syntaxic category of π the correct semantical function explained below. Thus $\mathcal{E}[\![\upsilon]\!]$ invokes *identifier* while $\mathcal{E}[\![(\texttt{lambda}(\upsilon)\pi)]\!]$ calls *abstraction*.

The primitive domains are

$\upsilon \in$ **Id**

$\pi \in$ **Form**

Cons $=$ **Loc** $\times$ **Loc**

$\epsilon \in$ **Val** $=$ **Cons** $+$ **Symbol**

$\rho \in$ **Env** $=$ **Id** $\rightarrow$ **Loc**

$\sigma \in$ **Store** $=$ **Loc** $\rightarrow$ **Val**

$\kappa \in$ **Cont** $=$ **Val** $\times$ **Store** $\rightarrow$ **Val**

$\phi \in$ **Func** $=$ **Val** $\times$ **Store** $\times$ **Cont** $\rightarrow$ **Val**

The current environment ρ converts an identifier υ into a location which, in turn, given to a store σ, is converted into a value which is returned to the continuation κ along with the unmodified store.

$$
\begin{array}{l}
\mathit{identifier}(\upsilon) = \\
\boldsymbol{\lambda}\sigma\rho\kappa. \\
\quad \kappa(\sigma(\rho(\upsilon)), \sigma)
\end{array}
$$

To simplify, a function is restricted to a single variable υ and a single body π. This Lisp kernel is a lexical Lisp, `lambda` creates a closure containing the definition environment ρ. *newlocation* is an implementation-dependent function which given a store finds an unused location α. In this equation, σ is the definition store while σ_1 is an application store.

$$
\begin{array}{l}
\mathit{abstraction}(\upsilon, \pi) = \\
\boldsymbol{\lambda}\sigma\rho\kappa. \\
\quad \kappa(\boldsymbol{\lambda}\epsilon\sigma_1\kappa_1. \\
\qquad\quad \mathcal{E}(\pi) \\
\qquad\qquad (\sigma_1[\alpha \leftarrow \epsilon], \rho[\upsilon \leftarrow \alpha], \kappa_1) \\
\qquad\quad\ \textbf{where } \alpha = \mathit{newlocation}(\sigma_1), \sigma)
\end{array}
$$

A form is evaluated by applying the value of its first term π to the value of its second term π_1. The value of the first term ϕ is assumed to be a function.

$$
\begin{array}{l}
\mathit{combination}(\pi, \pi_1) = \\
\boldsymbol{\lambda}\sigma\rho\kappa. \\
\quad \mathcal{E}(\pi) \\
\qquad (\sigma, \rho, \boldsymbol{\lambda}\phi\sigma_1. \\
\qquad\qquad\quad \mathcal{E}(\pi_1) \\
\qquad\qquad\qquad (\sigma_1, \rho, \boldsymbol{\lambda}\epsilon\sigma_2. \\
\qquad\qquad\qquad\qquad\quad \phi(\epsilon, \sigma_2, \kappa)))
\end{array}
$$

Let us now give the definition of the predefined function `cons` which takes two parameters π and π_1.

$$
\begin{array}{l}
\mathit{subr\text{-}cons}(\pi, \pi_1) = \\
\boldsymbol{\lambda}\sigma\rho\kappa. \\
\quad \mathcal{E}(\pi) \\
\qquad (\sigma, \rho, \boldsymbol{\lambda}\epsilon\sigma_1. \\
\qquad\qquad\quad \mathcal{E}(\pi_1) \\
\qquad\qquad\qquad (\sigma_1, \rho, \boldsymbol{\lambda}\epsilon_1\sigma_2. \\
\qquad\qquad\qquad\qquad\quad \kappa(<\epsilon, \epsilon_1>, \sigma_2)))
\end{array}
$$

We can now deduce (by β-reductions) the meaning of

`(let (`υ` (cons `π` `π_1`)) `π_2`)`

$$
\begin{array}{l}
\boldsymbol{\lambda}\sigma\rho\kappa. \\
\quad \mathcal{E}(\pi)(\sigma, \rho, \boldsymbol{\lambda}\epsilon\sigma_1. \\
\qquad\qquad\quad \mathcal{E}(\pi_1)(\sigma_1, \rho, \boldsymbol{\lambda}\epsilon_1\sigma_2. \\
\qquad\qquad\qquad\qquad \mathcal{E}(\pi_2)(\sigma_2[\alpha \leftarrow <\epsilon, \epsilon_1>], \rho[\upsilon \leftarrow \alpha], \kappa) \\
\qquad\qquad\qquad\qquad\ \textbf{where } \alpha = \mathit{newlocation}(\sigma_2)))
\end{array}
$$

The new special form is called **`dynamic-extent-cons-let`** (we shall abbreviate it to **`Dconslet`**). We restrict ourselves to only one variable, one cons-cell allocation and a one-form body. Its syntax is

`(Dconslet (`υ` `π` `π_1`) `π_2`)`

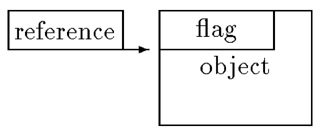
and its formal definition is

$$
\begin{array}{l}
\mathit{fsubr\text{-}dconslet}(\upsilon, \pi, \pi_1, \pi_2) = \\
\boldsymbol{\lambda}\sigma\rho\kappa. \\
\quad \mathcal{E}(\pi) \\
\qquad (\sigma, \rho, \boldsymbol{\lambda}\epsilon\sigma_1. \\
\qquad\qquad \mathcal{E}(\pi_1) \\
\qquad\qquad\quad (\sigma_1, \rho, \boldsymbol{\lambda}\epsilon_1\sigma_2. \\
\qquad\qquad\qquad\quad \mathcal{E}(\pi_2) \\
\qquad\qquad\qquad\qquad (\sigma_2[\alpha \leftarrow <\epsilon, \epsilon_1>], \rho[\upsilon \leftarrow \alpha], \boldsymbol{\lambda}\epsilon_2\sigma_3. \\
\qquad\qquad\qquad\qquad\qquad\qquad \kappa(\epsilon_2, \sigma_3[\alpha \leftarrow \mathit{obsolete\text{-}dcons\text{-}object}])) \\
\qquad\qquad\qquad\quad \textbf{where } \alpha = \mathit{newlocation}(\sigma_2)))
\end{array}
$$

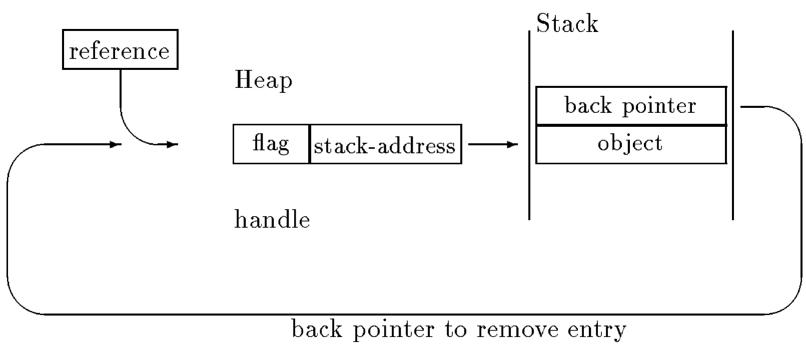
obsolete-dcons-object is the bottom element of domain **Cons** that is to say $\perp_{\textbf{Cons}}$. The difference between the two previous forms is the final store. The equation for `dconslet` suggests a stack-allocation of the freshly created cons-cell associated to a deallocation when exiting the body of the `dconslet`. Although written for a dynamic extent cons-cell, the lattest equation can also deal with any other DEO such as vectors, arrays, structures or bounded strings.

3 Related Implementations

The first implementation of dynamic extent objects that come to mind is to allocate them in the heap combined with a flag which indicates their validity. Deallocation is simply achieved by changing the flag to mean "obsolete". The GC will be responsible for scavenging them.



Since garbage collection of varisized objects is painful, one can choose to stack-allocate the object but to leave a handle with a flag in the heap. Something like



The GC is still responsible for recycling the handles but since they have a fixed size they can be grouped in a common area and recycled there.

4 Feasibility

We present a first raw implementation to explain the theory of operation. We shall tune it in the next section.

If we roughly allocate an DEO in the stack, the problem is to construct an indefinite extent reference to it with the following properties:

- until being deallocated, the reference to the object must lead to its stack-address,

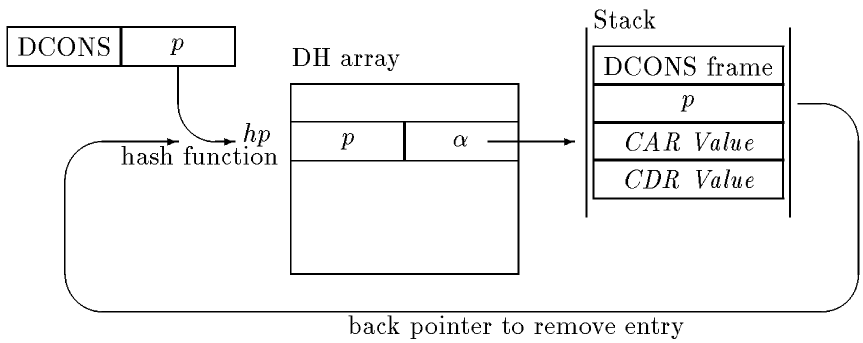
- after being deallocated and whatever the stack can be (shrunk or grown) the reference must forbid any access to the fields of the former object.

The reference cannot only be the stack-address since it would violates the second requirement. The reference must provide a way to check if the stack still contains the object. We then propose to uniquely number the DEO. We then record the association between these numbers and stack-addresses in a hash-array indexed by these numbers. Access to a DEO is done through

1. access to the hash-array, with the number of the wanted DEO,
2. check if the DEO is still in the stack and get its stack-address.
3. extract from the stack the desired field of the DEO

It is the responsibility of the deallocation process to remove the corresponding entry from the hash-array thus forbidding, given a reference, to obtain the associated stack-address.

If we used tagged pointers and dynamic extent cons-cells then the previous technique will look like



The deallocation process just removes the entry thus making the reference now erroneous. `car`, `cdr`, `rplaca` and `rplacd` will then raise an error. The hash-array looks like a set of handles from references to objects, maintained by allocation and deallocation processes. More precisely we can name `DH` this hash-array and `Dcons\`[1] (for allocation), `Dcar`, `Dcdr`, `Drplaca`, `Drplacd`, `Dconsp` (to check if the object is still accessible), `Duncons` (for deallocation) the operators to deal with dynamic extent cons-cells. Except `Dcons` and `Duncons`, all of them can be directly invoked by the user. `Dcons` and `Duncons` are only part of the *opaque* definition of the `Dconslet` *macro*, but are simpler to be described as functions.

```
(defmacro Dconslet ((id car-val cdr-val) body)
   `(let ((,id (Dcons ,car-val ,cdr-val)))
         (unwind-protect ,body
                (Duncons ,id) ) ) )
```

In order to allocate: `Dcons` builds a new frame onto the stack named `Dcons-frame`, with the two usual fields of a cons-cell (`car` and `cdr` fields) and a back pointer to allow the removal of the corresponding entry in `DH`.

```
(defun dcons (car-val cdr-val)
   (let* ((p (new-number))
          (alpha (push-frame 'Dcons-frame
                         p car-val cdr-val )) )
        (setf (gethash p DH) (get-DynExt-address alpha))
        (make-reference 'Dcons p) ) )
```

`new-number` returns a new number for a new dynamic extent object. It can be implemented by a global counter regularly incremented as

[1]We chose to prefix usual cons-cells operators by a `D` but these specialized methods can be added to preexisting generic functions such as `first`, `rest` ...

```
(let ((DynExt-counter -1))
     (define (new-number)
         (setq DynExt-counter (1+ DynExt-counter))
         DynExt-counter ) )
```

`(push-frame type . arguments-of-the-frame)` pushes a new frame onto the stack and returns the stack-address of the frame. To this address `get-DynExt-address` adds an offset to get the direct stack address of the dynamic extent cons-cell.

`make-reference` builds a pointer onto the p^{th} DEO which type is **Dcons**. In a tagged architecture, the reference is compound of a tag (`Dcons`) and an information (p, the dynamic extent cons-cell number).

`Dcar` and others are are all built on the same model. We assume references to be regular Dcons references.

```
(defun Dcar (reference)
   (let* ((p (extract-DynExt-number reference))
          (alpha (gethash p DH)) )
        (if alpha (take-car alpha)
            (error "Obsolete Object") ) ) )
```

`extract-Dynext-number` is a reciprocal to `make-reference`. It extracts the number of the dynamic extent object from a valid reference built by `make-reference`. `Dcar` checks if the object is still in the stack (`alpha` is not `nil`) and performs a `car` on the obtained stack-address (by `take-car`). Note that errors are only raised when one tries to access part of a dynamic extent object. It is therefore legal and safe, but not very useful (and contrary to dynamic extent spirit) to return an obsolete reference outside the `Dconslet` scope as can be seen in

```
(Dconslet (L 1 2) (equal L  (Dconslet (L 1 2) L)))
```

The result is obviously false since one compares two different references, one of which being obsolete.

We slightly lie in the opaque macro definition for `Dconslet` where, given a reference, the deallocation is explicitely done by the user. A real implementation deallocates the DEO while unwinding the stack, that is to say invokes `Duncons` on a frame-address and not on a reference. This latter version follows. `Duncons` deallocates the frame and cleans up `DH`.

```
(defun Duncons (frame-address)
   (let ((p (extract-DynExt-number-from-frame frame-address)))
        (remhash p DH)
        (pop-frame frame-address)
        t ) )
```

Given a Dcons-frame address, `extract-Dynext-number-from-frame` extracts the Dcons-number from the frame.

If we want to be more precise, we must exhibit the way such references behave under garbage collection. Marking is simple since we have only to follow the references only when permissible.

```
(defun mark (reference)
   (set-mark reference)
   (let* ((p (extract-DynExt-number reference))
          (alpha (gethash p DH)) )
         (if alpha (mark alpha)
             nil ) ) )
```

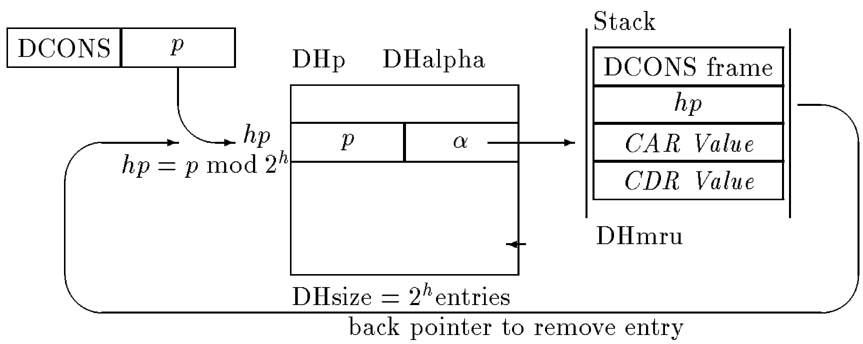
and naturally we can collect every unmarked references. Remark that it is not interesting to collect unmarked but valid dynamic extent objects ! They will be automatically deallocated when the stack is unwound.

The cost of this raw implementation is high since `Dcons` and `Duncons` mainly involves updates of `DH`, while `Dcar` and others require one access through `DH`. The next section will lessen it.

5 Enhancements

We discuss in this section some variations in order to lessen the price of access to DEO. The main cost is due to the `DH` hash-array which involves index computation and collision handling. The first cost can be reduced if we arrange `DH` to be an array with some power of two entries. Therefore the hash function can be as simple as a logical AND: a simple instruction on mainly all computers. The collision handling can be avoided if we manage to avoid any collisions ! This is possible since `new-number` can choose a good number which corresponds to a free entry in `DH`. For example, with the previous scheme, any number greater than `DynExt-counter` is possible and `new-number` can choose the first one which leads to the next free entry of `DH`.

The access now looks like



`DH` is no more an hash-array but a set of two vectors `DHp` and `DHalpha`. `DHp` records the dynamic extent objects numbers while `DHalpha` contains the associated stack-addresses. `Dcar` and similarly `Dcdr`, `Drplaca` and `Drplacd` become[2]

```
(defun Dcar (reference)
   (let* ((p (extract-DynExt-number reference))
          (dh *current-DH*)
          (hp (mod p [DHsize dh])) )
        (if (= (aref [DHp dh] hp) p)
            (take-car (aref [DHalpha dh] hp))
            (DHerror "Obsolete Object") ) ) )
```

An access is then compound of a mask operation (`mod`), an indexed load (`aref`), a test (`=`) and another indexed load followed by the standard `car` operation. As we said, this cost is not so high and is only a maximal cost which can be reduced by compilers in good situations. However we can estimate the maximum cost of `Dcar` to approximately five to seven times the cost of a `car` on an average stock hardware computer.

Two problems can arise:

- `DH` is exhausted,
- `new-number` fails to find a good number.

The size of `DH` (`DHsize`) is the total number of permitted DEO that can simultaneously reside in the stack. `DHsize` must then be related to the length of the stack. When `DH` is exhausted, some solutions must be taken. The cheapest way is probably to map `Dcons` on `cons` thus allocating in the heap rather than in the stack according to one of the two first ways given in section 3. The other solution is to double the size of `DH` and to rehash all entries.

The implementation of `new-number` follows directly from the previous section: a global counter `DynExt-counter` is maintained for the whole `DH` table. The problem is to minimize the growth of the counter, that is to say, to

[2] For reasons that will soon appear clearly we must use the current `DH` table: we then write `[DHsize dh]` instead of accessing the global variable `DHsize`. In fact, `DH` can be modelized by a structure with at least slots `DHsize`, `DHp` and `DHalpha`. Structures accesses are emphasized by square brackets.

choose the lowest number greater than DynExt-counter and corresponding to a free entry of DH. We name DHmru a pointer corresponding to the most recent used entry in DH. Then new-number is

```
(defun new-number ()
   (block found
      (repeat [DHsize *current-DH*]
         (setq DynExt-counter (1+ DynExt-counter))
         (setq DHmru (mod (1+ DHmru) [DHsize *current-DH*]))
         (if (equal (aref DHp DHmru) DHfree-entry)
             (return-from found DynExt-counter) ) )
      (error "DH exhausted") ) )
```

The algorithm is at its best when DH is sparse since then DynExt-counter will not grow too quickly. But a new problem arises: DynExt-counter can overflows. This problem will be soon addressed.

6 Where to put DH

If the DEO stack allocation is to work well, everybody will use it and it will increase references to the stack and to DH. The stack will be bigger since it will contain environments, continuations but also local data. Dedicated hardware with the top of the stack in the main processor will have considerably more stack page default but this will be partially corrected by the fact they have a stack cache. On the other hand dedicated hardware do not often have a heap cache, thus DH must not reside in the heap. The best way is then to allocate DH in the bottom of the stack but we can fortunately remark that DH *is itself a DEO* !

Suppose DH initially in the bottom of the stack. We want to create a new DEO and we must make room for its entry in the current DH. If DH is saturated then we must use a greater DH which extent is precisely the extent of the new DEO. So

- We stack allocate a new DH of greater size (two or four times the previous size),
- and we initialize the new DH with the old one and make the new one current. The *hash-function* is now a new mask operation. If we double the size of DH, we then take one more bit in the reference number to get the entry number.
  - We now allocate the new DEO,
  - resume computation ... and on return
  - we deallocate the entry
- we synchronize the old DH table with the current one (DynExt-counter has been incremented and DHmru must respect DHmru = DynExt-counter mod DHsize),
- we deallocate the current DH table,
- and we make the old DH table again current.

The greatest possible number of DH entries is bounded by the greatest possible reference number. This number is approximatively *most-positive-fixnum* and rarely under 2^{15} !

7 Cooperation with the GC

When DynExt-counter overflows (ie. can be no more a *fixnum*) we cannot use *bignums* since they are heap allocated ! We are thus constrained to reuse the same set of reference numbers. But at a given time, at most DHsize reference numbers are valid since DH can contain at most DHsize DEO simultaneously. Other reference numbers obviously correspond to obsolete objects. A solution is to invoke the GC (or only the sweep phasis) to renumber all references. If a reference number p designate a living DEO we can then renumber it as ($p \bmod \texttt{DHsize}$) and in case of obsolescence we can just turn it into $\perp_{\mathbf{Dcons}}$ so future access will be more quickly erroneous.

8 Performance Analysis

Performance analysis is quite difficult since we have to compare prices of heap-`cons` operations versus stack-`cons` operations. The former are very dependent of the performance of the GC while the latter cannot be approximated by an implementation in Lisp. A proper implementation of `Dconslet` requires access to the virtual machine below the Lisp system and particularly the ability to push and pop frames onto the stack. Nevertheless the DEO mechanism presented so far does not induce runtime overhead when one does not use it.

However we can examine the behaviour of the algorithm managing the `DH` array. Since `new-number` guesses a number corresponding to a free entry in `DHp`, bad cases may occur incrementing `DynExt-counter` too quickly and thus forcing a garbage collection when `DynExt-counter` overflows. `new-number` implements a linear probing algorithm which performances are analysed in [2], page 539. First, if we want the median of Δ`DynExt-counter` to slowly advance, say by 1.5 (resp. 2.5) for each allocation, the load average of `DH` must not exceed 30% (resp 50%). Thus to double `DH` is better triggered by the load average rather than by complete saturation. Second, since doubling `DH` is not costless, the greater is the initial `DH` and the better are the performances. This is particularly sensible in case of a bounded stack where the maximum number of simultaneous DEO can be known.

9 Extensions

`Dconslet` is restricted in several ways. `Dconslet` can only handle one variable and allocate a single cons-cell. To deal with multiple variables is straightforward. Let just have

```
(Dconslet ((L1 car1 cdr1)
           (L2 car2 cdr2)
           ... )
          body )
```

A single frame can be allocated for multiple dynamic extent objects but the local variables L_1 L_2 ... must be given as many different references as there are objects. The multiple variables `Dconslet` form is just a syntactic convenience.

Following declaration style[9], one can prefer to introduce a new declaration specifier and write

```
(let ((L1 car1 cdr1)
      (L2 car2 cdr2)
     ... )
    (declare (dynamic-extent L1 L2 ...))
    body )
```

We can also extend the form to dynamic extent objects other than cons-cells, such as lists, strings, vectors (any bounded resource): for example

```
(Dvectorlet ((V1 term1 term2 ...)
             (V2 term'1 term'2 ...)
             ... )
            body )
```

`Dvectorlet` is superior to `Dconslet` since it is not reduced to a cons-cell with just two pointers. A whole structure with numerous fields can be stack-allocated under a single reference.

10 Conclusion

We have presented an implementation technique for a new kind of first class objects with a dynamic extent which are allocated in the stack rather than in the heap. Every indefinite extent object has its counterpart as a DEO and support the same set of operators without restriction. DEO represent local and temporary resources which will be released without overhead. The proposed implementation respects the safety of references to such objects avoiding to create dangling pointers when they leave the stack. The access cost is not so high and the offered safety can be valuable during the debugging process. Surprisingly tables needed to access DEO are themselves DEO and thus reside in the stack. The technique is operated by new special forms or declarations and offers some speed improvements since the GC is not involved.

References

[1] Jérôme Chailloux, Matthieu Devin, Jean-Marie Hullot, *Le_Lisp: A Portable and Efficient Lisp System*, 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas.

[2] Donald E Knuth, *The Art of Computer Programming*, Volume 3, *Sorting and Searching*, Addison-Wesley, 1973.

[3] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, Norman Adams, ORBIT: An Optimizing Compiler for Scheme, Proceedings of the SIGPLAN '86 Symposium on Compiler construction, Palo Alto (CA), June 25-27, 1986, SIGPLAN Notices Vol 21, No 7, July 1986.

[4] Bernard Lang, Francis Dupont, *Incremental Incrementally Compacting Garbage Collection*, SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, Saint Paul, MA.

[5] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, Michael I. Levin, *Lisp 1.5 Programmer's Manual*, MIT Computation Center and Research Laboratory of Electronics, August 17, 1962.

[6] David A. Moon, *Garbage Collection in a Large Lisp System*, 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, pp 235–246.

[7] Jonathan A. Rees, Norman I. Adams, James R. Meehan, *The T Manual*, Fourth Edition, 10 January 1984, Computer Science Department, Yale University, New Haven CT.

[8] Jonathan Rees, William Clinger, *Revised^3 Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21, 12, Dec 86, pp 37 – 79.

[9] Guy L. Steele, Jr. *Common Lisp, the Language*, Digital Press, Burlington MA, 1984.

[10] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, 1977.

[11] *Symbolics Common Lisp Reference Manual*, Symbolics.