The Mathematical Definition of the Nanolisp Environment

Francis Sergeraert

December 6, 1995

A Lisp (that is, Nanolisp) *environment* E contains three components, the *system environment* E^s, the *global environment* E^g and the *local environment* E^l: $E = E^s + E^g + E^l$. Usually, only a small part of the local environment E^l is *visible*; this part is denoted by E^{lv}, so that the visible part of the whole environment is $E^v = E^s + E^g + E^{lv}$. The components of the environment evolve in a rich and sophisticated way during a Lisp session, in particular during the evaluation of a Lisp object.

Each component of the environment is mainly an organized set of *bindings*; a binding is a pair {*symbol*→*value*}; the symbol is the *source* of the binding, the value is its *target*; such a value can be any Lisp object, except an error object; in particular a value may be itself a symbol. If a binding {*symbol*→*value*} is a member of an environment (resp. of an environment component), one says *symbol* is *present* in this environment (resp. in this environment component).

The system environment E^s is constant and contains 28 bindings from the 28 *system* symbols toward the predefined corresponding objects, namely the 9 *itype objects*, the 2 *boolean* objects, the 14 *special* objects and the 3 *system* objects. No binding in the system environment can be added or modified.

The global environment E^g in general contains several bindings, but is empty at the beginning of a Lisp session. A binding is added to the global environment if and only if the object `<SPECIAL SET>` works, asking to install the binding {`symbol`→`value`}, if `symbol` is present neither in the visible local environment E^{lv} nor in the global one E^g; the new binding is then *added* to the former global environment.

A binding of the global environment is modified if and only if the object `<SPECIAL SET>` works, asking to install the binding {`symbol`→`value`}, if `symbol` is present in the global environment E^g but not in the visible local one E^{lv}. Then the old target of the corresponding binding in the global environment is erased and *replaced* by `value`. A consequence of these rules is that a symbol cannot be present twice in a global environment.

To describe the local environment, it is important to recall the precise terminology about lambda-objects. In a first step a lambda-object is *created*; most often

this happens when a list `(lambda ...)` is evaluated:

```
> (set 'second (lambda (list) (first (rest list))))  ==>
<LAMBDA ((LIST) (FIRST (REST LIST)))>
```

Here a lambda-object has been created and assigned to the symbol `SECOND` in the global environment; the main point is to note this lambda-object has been created but has not yet been *invoked*. On the contrary, in the following statement:

```
> (second '(a b c))  ==>
B
```

the lambda-object which is the value of `SECOND` is once *invoked*, but is not created; it was created earlier. When a lambda-object is invoked, an *invocation* is started and becomes *active*; when some work is done, Lisp *terminates* this invocation and this invocation will be definitively non-active.

When a `((lambda ...) ...)` statement is executed, the evaluation of the subexpression `(lambda ...)` *creates* a lambda-object, and the last step of evaluation for the outermost list *invokes* it:

```
> ((lambda (list) (first (rest (list)))) '(a b c))  ==>
B
```

An important difference between the notions of creation time and invocation time is the following; a creation time is "instanteneous", that is, without any duration in theory: in other words, the creation of a lambda-object happens during an elementary step of our Lisp machine; on the contrary an invocation time can spend much time, even an infinite time:

```
> (set 'loop (lambda () (while true 'do-anything)))  ==>
<LAMBDA ...>
> (loop)  ==>
????????????
```

The local environment E^l is a pair (N, Σ) where N is the *segment number stack* and Σ is the *segment set*.

The segment set Σ is a finite set $\{\sigma_1, \sigma_2, \ldots, \sigma_{n-1}\}$, made of all the segments already created in the Lisp session. Exactly *one* segment is created each time a lambda-object is invoked, and every segment remains alive until the end of the Lisp session, in particular after the end of the invocation which created it. If an already invoked lambda-object is again invoked, another segment is created but the segment created by the previous invocation is still present and will remain present. When a lambda-object is *created*, no segment is *created*: a segment is created only at invocation time.

The number n is the *next segment number*, not yet used; the next segment number n will be used at the next invocation of a lambda-object. At the beginning of a Lisp session, the next segment number is 1 and Σ is empty.

Each segment σ_i is a pair (p_i, β_i). The integer p_i is called the *p-link* and is the number of the *next visible segment*, more precisely the number of the next segment which is visible if the segment σ_i is visible. Conventionally, $p_i = 0$ means there is no other visible segment after σ_i; another way to understand this equality is to consider the next visible segment is the global and system environments: $\sigma_0 = E^g + E^s$. If the segment σ_i is visible and if $p_i = 0$, then σ_i is the *last visible segment* at this time in the local environment.

If i is a segment number, the *visible part* of the local environment *starting from* σ_i (more precisely the visible part of the local environment which *would be* visible if σ_i was visible) is the sequence $(\sigma_{v_r}, \sigma_{v_{r-1}}, \ldots, \sigma_{v_1})$ where the indices $v_1, \ldots, v_r$ satisfy the following conditions:

- $v_r = i$;
- $p_{v_{s+1}} = v_s$ if $1 \leq s < r$;
- $p_{v_1} = 0$.

In other words, the number of visible segments starting from σ_i is r; the first one is σ_i; for each visible segment, the next one in this stack of visible segments is defined by the p-link; the deepest element of this stack is identified by the relation $p_{v_1} = 0$. In particular, if $p_i = 0$, then the visible part starting from σ_i contains only σ_i and $r = 1$.

The second component β_i of σ_i is the binding set of this segment; the sources of this binding set are the parameter symbols of some lambda-object λ, the lambda-object which created at invocation time the segment σ_i; the values are the corresponding arguments for this invocation, which values may have been modified by the body of the lambda-object or also a deeper lambda-object.

The segment number stack N is a list of segment numbers $N = (n_k, n_{k-1}, \ldots, n_1)$; each segment number n_l of this list satisfies the relation $1 \leq n_l < n$. At this time, a stack of k invocations of lambda-objects is processed; the highest invocation of this stack has created the segment σ_{n_k} which is also the *current first visible segment* in the local environment: the currently visible part E^{lv} of the local environment starts form the segment σ_{n_k} and is possibly extended to earlier constructed segments according to the process described above, using the p-links. When a top level evaluation begins, the segment number stack N is void and $E^{lv} = \emptyset$, no segment of the local environment is visible. Each time a lambda-object is invoked, a new element is *pushed* on N, namely the number of the segment just created to process this invocation; each time a lambda-object invocation is finished, the first element of the segment number stack N is removed.

We must now explain how this relatively complex data set evolves during a Lisp session. When a lambda-object is *created*, in other words when the special object `<SPECIAL LAMBDA>` works (most often because a list `(lambda ...)` is evaluated), the new lambda-object which is created is made of three components; the first one is the current first visible segment number n_k, the second one is the parameter list,

and the last one is the body of the lambda-object. Only the two last components are showed in the external form of the created lambda-object. In particular, the creation of a lambda-object *does not* modify the environment, but the segment number stored in the lambda-object will allow Lisp at *invocation time* to restore the visible environment at *creation time*, which visible environment is entirely described by the number n_k; this visible environment will be only completed by the unique segment created for this invocation, which segment will be linked to this old visible environment by the p-link set to the *ancient* n_k copied from the lambda-object. If λ is a lambda-object, we denote by m_λ the stored segment number. In the (frequent) particular case where the visible part of the local environment is void at creation time, in other words when the segment number stack N is empty, then m_λ is set to 0.

When a lambda-object λ is *invoked*, the following modifications of the local environment are performed. A new segment σ_n is created; its p-link p_n is m_λ; its second component β_n is the binding set from the parameter symbols to the corresponding (definitive) arguments for this invocation; the number n is pushed on the segment number stack N and the just created segment σ_n is added to the segment set Σ. Finally the next segment number becomes $n + 1$.

If the value of a symbol is asked for during an evaluation process, Lisp looks firstly at the visible segments whether the questionned symbol is present; else Lisp looks at the system and global environments; else an error is generated. The same method is applied if the special object `<SPECIAL SET>` must modify the value of a symbol: only one binding will be modified, the first visible one; otherwise a new binding is added, certainly to the global environment.

Finally when the evaluation of a lambda body is finished, it is sufficient for Lisp to remove the highest segment number of the segment number stack N; in other words the number k becomes $k - 1$.

A consequence of this description is the following: the first visible segment is always the segment which has been created by the last invocation *still active*: later invocations can be now terminated; the other visible segments are those which *were* visible when the corresponding lambda-object *was* created. The global and system environment are always visible, but are used only if the visible part of the local environment does not give a solution for a symbol value or to modify a symbol value.

We explain now how this organization works for the various natural situations in Lisp programming. The simplest situation is the creation of a "top-level" lambda-object, later used. For example, if in a Lisp session, we works in this way:

```
> (set 'second (lambda (list) (first (rest list))))  ==>
<LAMBDA ((LIST) (FIRST (REST LIST)))>
```

then the visible part of the local environment is void when the lambda-object is created, so that the segment number stored in the lambda-object is simply 0: this is the most frequent situation in ordinary programming. When this lambda-object is used (*invoked*):

```
> (second '(a b c))  ==>
B
```

a new local segment is created $\sigma_n = (p_n, \beta_n)$ with $p_n = 0$ and β_n containing only the binding {LIST→(A B C)}, so that this binding is the only one which is visible in the local environment. If just after this invocation we call again the functional value of SECOND:

```
> (second '(d e f))  ==>
E
```

then *another* segment $\sigma_{n+1} = (0, \beta_{n+1})$ is created containing the new binding. In fact the old segment σ_n is still in the local environment E^l but will be no longer visible. It is clear an optimized implementation should put into garbage such a segment; this point is discussed later. But in our organization, this certainly now useless segment remains in the local environment: each time a lambda-object is invoked, a new segment is created and is definitively kept.

Let us now consider a situation where an ordinary recursive process is used. For example consider the following function which removes the leading a's of a list:

```
> (set 'remove-leading-a-s
    (lambda (list)
      (if (equal list '())
          list
        (if (equal (first list) 'a)
            (remove-leading-a-s (rest list))
          list))))  ==>
<LAMBDA ((LIST) (IF ...))>
> (remove-leading-a-s '(a a a b c a))  ==>
(B C A)
```

Again a top-level lambda-object is created, where the stored segment number is 0. Let us suppose this is done when the next segment number is 14. Then the first invocation of remove-leading-a-s (in fact the lambda-object value) creates the segment $\sigma_{14} = (0, \beta_{14})$ where β_{14} contains the binding from the parameter LIST to the list (A A A B C A). During the body evaluation, the same lambda-object is again invoked but this time the argument is (A A B C A). Lisp then creates a new segment $\sigma_{15} = (0, \beta_{15})$ containing the new corresponding binding; the segment number stack N is now $(15, 14)$, but only the segment σ_{15} is visible. At the deepest invocation the segment number stack N is $(17, 16, 15, 14)$ and the only visible segment is σ_{17} containing a unique binding to (B C A). Then the recursive process is finished, the successive invocations are ended, the elements of the segment number stack N are sequentially removed and the right result (B C A) is returned. The important point here is that even in such a recursive process, only one (local) segment is visible at any time.

Le us see now a relatively artificial situation where the number of visible segments is high, and on the contrary the segment number stack N is short. Look at the following sequence:

```
> (set 'f1 (lambda (a)
             (lambda (b)
               (lambda (c) (progn c b a)))))  ==>
<LAMBDA ((A) (LAMBDA ...))>
```

Let us suppose we are starting a new Lisp session, so that the local environment is void. In the lambda-object value of `F1`, the segment number 0 is stored. Then:

```
> (set 'f2 (f1 'arg-a))  ==>
<LAMBDA ((B) (LAMBDA ...))>
```

Here the invocation of the value of `F1`, in short the invocation of `F1`, creates the segment 1 with the p-link $p_1 = 0$ (the segment number stored in `F1`) and the binding {`A`→`ARG-A`}. This is the unique visible segment. Evaluation of the `F1`-body creates a new lambda-object with a stored segment number 1, a parameter list `(B)` and the body `(LAMBDA (C) ...)`; finally this lambda-object is assigned to `F2` (in the global environment) and returned.

```
(set 'f3 (f2 'arg-b-1))  ==>
<LAMBDA ((C) (PROGN C B A))>
```

When `F2` is invoked, the segment number stack N is void. The segment 2 is created containing the p-link $p_2 = 1$ (the segment number stored in `F2`) and the binding {`B`→`ARG-B-1`}. We are here in a situation where *two* segments are visible, the segments 2 and 1, but the segment number stack N has only *one* element, the integer 2. With this local environment, again a lambda-object is created with the stored segment number 2, and this lambda-object is assigned to `F3`. Finally:

```
> (f3 arg-c-1)  ==>
(ARG-A ARG-B-1 ARG-C-1)
```

You see that when `F3` works, three segments (3+2+1) are visible containing respectively the bindings for `C`, `B` and `A`. The segment number stack N contains only the number 3. When this has been done, let us work as follows:

```
> (set 'f4 (f2 'arg-b-2))  ==>
<LAMBDA ((C) (PROGN C B A))>
> (f4 'arg-c-2)  ==>
(ARG-A ARG-B-2 ARG-C-2)
```

When `F2` is invoked the segment 4 is created with the new binding for `B` and the p-link $p_4 = 1$, the segment number stored in `F2` which of course is the same as before. A lambda-object is created which is assigned to `F4` and containing the stored segment number 4. When `F4` is invoked, the segment 5 is created, and the visible segments are 5+4+1, so that the call of `F4` uses the right bindings. The old segments σ_2 and σ_3 are still present in the local environment; the segment σ_2

can be possibly again used if F3 is reinvoked; on the contrary the segment σ_3 is now unreachable.

You see the segment set is structured as a forest where each segment σ_i is linked to the next visible segment, the number of which is obtained by the p-link. This structure is *static*, only extended for each new invocation of a lambda-object: each created segment is installed somewhere in this forest, according to the segment number stored in the lambda-object just invoked. There is one tree for each segment σ_i satisfying $p_i = 0$. In ordinary programming, each tree contains only one segment, but if you work in a *functional* framework, that is, if you use functions which will create functions which will create functions and so on, the tree structure can become complicated.

On the contrary the segment number stack N is a *dynamic* object, taking account of the most recent history in your Lisp session: the segment number stack depends only on the last input object you asked for evaluation. In particular the segment number stack is always empty when the top-level evaluation of a Lisp object is finished, that is, when the Lisp prompt is displayed and Lisp is waiting for the next (input) object you want to evaluate.

Another situation which must be considered is when one creates and immediately uses a lambda-object:

```
> ((lambda (list) (cons 'a list)) '(b c d))  ==>
(A B C D)
```

Then the evaluation of the first element of the evaluated list creates a lambda-object with a stored segment number = 0. The definitive argument (B C D) is computed, and the lambda-object just created is immediately invoked, so that a new segment is created containing the binding {LIST→(B C D)} and which is the only visible segment. Such a mechanism can be stacked:

```
> (((lambda (symb)
      (lambda (list) (cons symb list)))
    'a)
   '(b c d))
(A B C D)
```

Here, when the internal lambda-object works, it *sees* the right binding for the symbol SYMB. This allows you to organize the environment so that a local variable is kept from one invocation to the next one:

```
> ((lambda (local-variable)
     (progn
       (set 'setter
            (lambda (new) (set 'local-variable new)))
       (set 'pusher
            (lambda (list) (cons local-variable list)))))
   'a)  ==>
(<LAMBDA ((LIST) ...)>
```

```
NL > (pusher '(b c d))  ==>
(A B C D)
NL > (setter 'aa)  ==>
AA
NL > (pusher '(b c d))  ==>
(AA B C D)
```

The lambda-objects that are the respective values of `SETTER` and `PUSHER` see the *same* binding for the symbol `LOCAL-VARIABLE`, so that `PUSHER` can use it and `SETTER` can modify it: the next call of `PUSHER` will take account of the modified binding. Look also at this example where it is natural to create a lambda-object with an empty parameter list:

```
> (set 'make-list-popper
       (lambda (list)
         (lambda ()
           ((lambda (temp)
              (first
               (progn
                 (set 'list (rest list))
                 temp)))
            (first list)))))  ==>
<LAMBDA ((LIST) (LAMBDA () ...))>
NL > (set 'popper-1 (make-list-popper '(a b c)))  ==>
<LAMBDA (NIL ((LAMBDA (TEMP) ...) ...))>
NL > (set 'popper-2 (make-list-popper '(aa bb cc)))  ==>
<LAMBDA (NIL ((LAMBDA (TEMP) ...) ...))>
NL > (popper-1)  ==>
A
NL > (popper-1)  ==>
B
NL > (popper-2)  ==>
AA
NL > (popper-1)  ==>
C
NL > (popper-2)  ==>
BB
```

Note the trick consisting in encapsulating the popping mechanism inside a `((lambda (temp) ...)  (first list))` in order to temporarily store the first element of the list. This is the usual way to construct and use a temporary memory area in this extended λ-calculus.

Let us restart a new Lisp session and work as follows:

```
> (set 'make-setter-pusher-pair
       (lambda (setter-name pusher-name init)
         ((lambda (local-variable)
            (progn
              (set setter-name
                   (lambda (new)
                     (set 'local-variale new)))
              (set pusher-name
                   (lambda (list)
                     (cons local-variable list)))))
          init)))  ==>
<LAMBDA ((SETTER-NAME ...) ...)>
> (make-setter-pusher-pair 's1 'p1 'a)  ==>
(<LAMBDA ((LIST) ...)>
> (p1 '(b c d))  ==>
(A B C D)
> (s1 'aa)  ==>
AA
> (p1 '(b c d))  ==>
(AA B C D)
> (make-setter-pusher-pair 's2 'p2 'b)  ==>
(<LAMBDA ((LIST) ...)>
> (p2 '(b c d))  ==>
(B B C D)
> (p1 '(b c d))  ==>
(AA B C D)
> (s2 'bb)  ==>
BB
> (p2 '(b c d))  ==>
(BB B C D)
```

It is a good exercise to draw the entire segment structure. In particular two *different* segments contain now a binding for `LOCAL-VARIABLE`; the first one is visible from the values of `S1` and `P1`, the second one is visible from `S2` and `P2`.

We end these explanations with some implementation comments. In an efficient implementation, the described organization is significantly improved as follows. For ordinary programming without functional programming, this mechanism is not at all used; only a local memory area is allocated when a lambda-object is invoked and freed when the invocation terminates. When functional programming occurs, each segment is organized as a set of bindings followed by an pointer (that is, a machine address) toward the following visible segment. In this way a segment can be shared between several lambda-objects. Frequently it can be *proved* some segment is now unreachable. A process called *garbage collector* cleans the memory, throwing away the useless segments, and reorganizing the other segments in an efficient way, in particular coherently modifying the link addresses. The "professional" Lisp compilers are able to recognize these various situations to produce very efficient machine code, even if complicated functional programming is used.

-o-o-o-o-o-o-o-