CHAITIN’S TOYLISP ON A CONNEX MEMORY MACHINE 1

Gheorghe Ștefan
Politechnical University of Bucharest, Department of Electronics, Bd. Armata Poporului 1-3, Bucharest 6, Romania, email: stefan@hera.pub.ro.

Mihaela Malita
University of Bucharest, Faculty of Mathematics, Str. Academiei 14, Bucharest 1, Romania, email: system@bsu.ro.

Abstract: This paper describes an implementation of Chaitin’s ToyLisp [Chaitin ‘87] on a Connex Memory Machine (CMM) [Ștefan ‘85]. The Connex Memory Machine has a smaller complexity than previous Universal Machines used to run Lisp programs, so the time and space used in running Lisp programs can be considerably decreased. A ToyLisp like language can be used with a CMM to construct a Lisp (Co)Processor for accelerating Lisp processing in conventional architectures.

1 Introduction

Improving the performance of a string processing system could be done in two main ways:

– by defining and designing a high performance processing unit, having a “well fitted” set of functions, or
– by finding a “good” memory support for the data structures.

We have investigated the second alternative by defining a more “natural” memory support for lists, i.e. a better representation for storing and manipulating lists. We suggest a “smart memory” in which some of its functions are performed at the storage level. Two main consequences derive from this approach:

– the processor attached to the memory becomes simpler and faster,
– the computational processes are simpler and their time and space complexities decrease.

In the ‘80s a small team in the Polytechnical University of Bucharest designed and implemented a Lisp Machine as a microprogrammed machine. From this experience a new memory model has emerged: the Connex Memory [Ștefan ‘85].

Our paper has two aims:

2 Some applications of this memory were presented in [Ștefan ‘91], [Ștefan 95], [Hasci 95].
– a theoretical one: to offer another model for Universal Machine used to interpret Chaitin’s ToyLisp;
– a practical one: to define and implement a Lisp (Co)Processor starting from a ToyLisp like language as an assembly language.

2 Basic Requirements for String Processing

A Lisp-like language efficient implementation requires some unusual features, as:

– a “natural” representation of S-expressions on the physical support,
– a “natural” manipulation of S-expressions,
– time operation uncorrelated with the string length.

All these requirements can be satisfied with a physical support that:

– finds (accesses) the starting place of a substring in a string,
– inserts/deletes a symbol in an accessed point,
– delimits a self-delimited substring.

Dealing with symbols implies operations as finding, matching, inserting, deleting, moving, copying. These operations, usually, don’t affect the symbol itself, but its position in the string. Memory functions are crucial in this case. Numerical operations are realized by functions that modify the value of their arguments using processing units. A processor like approach is more fitted for numbers.

Fig. 1. The von Neumann Model

Having in mind the well known von Neumann model (Processor - Channel - Memory, see Fig. 1) we believe that for a more efficient symbol manipulation we need new memory functions instead of more powerful processors. Therefore, our proposal is a new concept of memory: the Connex Memory.

3 The Connex Memory

3.1 Informal Definition of the Connex Memory

The connex memory (CM) is a physical support for a symbolic string in which we can find any substring, identifying, in such a manner, any place in the string, for reading, inserting or deleting a symbol or a substring of symbols.

The content of the CM is a string of variables: \( v_1 v_2 \ldots v_i \ldots v_n \). Each variable \( v_i \) has:

---

\(^3\) This could be a step towards obtaining a shorter diophantine equation describing Chaitin’s Omega Number, [Chaitin ’87, Calude ’94].
- a value from a finite alphabet $A$,
- a state which is marked or non-marked; the access point is at the first marked variable.

For example, the content in some place in CM is the string: ...(bubu (good boy))... In Fig. 2 we can see the structure of the representation of the variables from the string: one bit for the marker and $m = \log_2(\text{card}(A))$ bits for each symbol in the alphabet $A$.

On any string stored in CM we can apply the set of functions described in the following definition.

**Definition 1.** The connex memory CM1 is a physical support for a string of variables, having values from a finite set of symbols $A$ and two states: non-marked or marked, over which we can apply the following set of functions:

- $\text{RESET } s$: all the variables after the first marked variable take the value $s$
- $\text{FIND } s$: all the variables that follow a variable having the value $s$ switch to the marked state and the rest switch in the non-marked state
- $\text{C(onditioned)FIND } s$: all the variables that follow a variable having the value $s$ and being in the marked state switch in the marked state and the rest switch in the non-marked state
- $\text{INSERT } s$: the value $s$ is inserted before the first marked variable
- $\text{READ up/down}[]$: the output has the value of the first marked variable and the marker moves one position to right (up) or to left (down) or remains unchanged ($\_\_$)
- $\text{DELETE:}$ the value stored in the first marked position is deleted, the position remains marked (the output has the value of the first marked variable) and the symbols from the right are moved one position left

![Fig. 2. The content of CM. An example](image)

For different purposes in different stages of the computational process we need different types of markers. Until now our example has involved only one marker. We shall consider an extended definition of CM with two or more markers. In this way we could "colour" the symbols of the string with more than one "colour" (marker).

**Definition 2.** The CM2 has in addition to the set of functions defined in for CM1 (see Definition 1) the following new function:
SET POINTER \textit{p}i[1, 2, \ldots, k]: the pointer \textit{p}i (i = 1, 2, \ldots, k) is set to the first position of the marker and modifies the next functions as follows (the square brackets are used to indicate optional fields in the mnemonics of the function):

- \texttt{INSERT} [\textit{p}i], \texttt{s}: inserts the symbol \texttt{s} in the position indicated by the pointer \textit{p}i; if a pointer is not indicated, \texttt{s} is inserted to the first occurrence of the marker;
- \texttt{READ up\textbackslash down} ~ [\textit{p}i]: starts reading from the position indicated by the \textit{p}i pointer which is affected in the same manner as the marker;
- \texttt{DELETE} [\textit{p}i]: deletes the position pointed by \textit{p}i.

According to these definitions the CM is structured as a bi-directional shift register in which any significant point is marked, as a consequence of an associative sequential mechanism used to find a name in a number of steps equal to the length of the name. In the marked place, \texttt{read}, \texttt{insert} and \texttt{delete} can be performed. Theoretically, the CM at the right end is unlimited, and, consequently, can be used for simulating or emulating efficiently any number of registers having an unspecified length (theoretical infinite). Briefly: CM = is a CAM designed as a Bi-directional Shift Register left delimited by the first marker.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig3.png}
\caption{Block Diagram of CM}
\end{figure}

The block diagram of CM1 from Fig. 3 shows the connections for data and functions defined before. We should not be surprised that this kind of memory has no addresses. Indeed, we don’t need them because the access point is found by an associative search. In this way we avoid the log, connections used for addressing a standard memory.

It is a real joy to discover that this type of memory could remain, with an appropriate design, in the class of \textit{O}(n). At this reduced complexity, all functions are executed in time \textit{T}(n) ∈ \textit{O}(1), in one clock cycle. For example, using the 16 Mb DRAM technology we realize our CM of 64 Ksymbol, where each symbol is encoded in 12 bits with 4 markers (having 16 states per symbol).

3.2 The Connex Memory at Work

Let us see some examples illustrating the main abilities of the CM.

\textbf{Example 1.} Using the CM1 we can find any substring in a string. Suppose that we have in a CM the following string:

\[(\textit{bubu (bad butcher)})\ldots(\textit{bulgaria(\ldots)})\ldots\]

and we want to select and read the list of \textit{bubu}’s properties, i.e., (\textit{bad butcher}). The following sequence of functions is executed:
The content of the CM becomes successively (the marked places are indicated by oversigned symbols):

... (bubu (bad butcher))...(bulgaria (..))... 
...(būbu (bād butcher))...(būlgaria (..))... 
...(bubu (bad butcher))...(bulgaria (..))... 
...(būbu (bad butcher))...(būlgaria (..))... 
...(bubu´ (bad butcher))...(bulgaria (..))... 
...(bubu (bad butcher))...(bulgaria (..))... 
...(bubu (bad butcher))...(bulgaria (..))... / out = ( / 
...(bubu (bād butcher))...(būlgaria (..))... / out = b / 
...(bubu (bad butcher))...(bulgaria (..))... / out = a / 
...(bubu (bad butcher))...(bulgaria (..))... / out = d / 
and so on, until:
...(bubu (bad butcher))...(bulgaria (..))... / out = ) /.

**Example 2** Let us suppose that we want to change the first property of bubu with the value *good*. The sequence of functions will be:

FIND ( 
CFIND b 
CFIND u 
CFIND b 
CFIND u 
CFIND blank 
**loop**
    READ up 
    until ) 
**repeat**
    DELETE 
    until blank 
**loop**
    INSERT g 
    INSERT o 
    INSERT o 
    INSERT d 
    INSERT blank 

Now the new content of the CM becomes:

...(bubu (good butcher))...(bulgaria (..))...
4 CM Machine as a Universal Machine

The Connex Memory Machine (CMM) has the computational power of a Universal Turing Machine. In order to compare the CMM with other models we will make a short review of two other models: Universal Turing Machine (UTM) and Chaitin’s Register Machine (RM).

4.1 Universal Turing Machine

Instead of the classical definition for a Turing Machine (TM) with:
- an infinite tape containing a string of symbols that can be read or modified,
- a finite automaton (which, for UTM, “knows” how to interpret a symbolic substring on the tape as a TM description),
- a read/write head which accesses the tape,
we propose a TM version more appropriate to the actual technology. In Fig. 4 the UTM contains:
- an infinite Random Access Memory (RAM), instead of the tape
- a finite automaton, as interpreter for TM description from the RAM
- an up-down counter that generates the address to the RAM.

![Fig. 4 Universal Turing Machine (UTM)](image)

**Definition 3** An UTM is a 4-tuple (Fig. 4):

\[ UTM = (A, Q, f; #) \]

where:
- \(A\), the finite alphabet (for the RAM),
- \(Q\), the finite set of states (of the automaton),
- \(f\), the transition function of UTM, \(f : A \times Q \rightarrow A \times Q \times C\), where \(C = \{UP, DOWN, -\}\) is the set of commands given by the automaton to the U/D Counter,
- \(# \in A\) is a special symbol delimiting the active space in RAM.

Therefore, a TM is a three part system: the finite automaton (for control), the infinite automaton (for addressing the RAM), and the infinite RAM.
4.2 Chaitin’s Register Machine

G. J. Chaitin ([Chaitin ‘87, ‘94]) defines the RM optimizing the UTM\(^4\) The goal of this approach is to built a Lisp oriented machine with a better solution for list inserting and deleting.

In Fig. 5 the RM has a finite automaton (as interpreter for ToyLisp programs) and a finite number of infinite left/right shift registers used as stacks (initially, in one register we have a Lisp object to be evaluated and the final result will be found in any register).

**Definition 4.** A RM (see Fig. 5) is a 5-tuple:

\[ RM = (A, Q, F, R, g) \]

where:

- A, the finite alphabet (for registers),
- Q, the finite set of states (of the automaton),
- F, the finite set of functions applied to the content of the registers, \( F = \{ \text{READ, WRITE, PUSH, POP} \} \),
- R, the finite set of registers,
- g, the transition function of RM, \( g : Q \times R \times A \rightarrow Q \times R \times A \times F \).

It is obvious that a RM with minimum two registers is universal.

\(^4\) to construct his paradoxal Big Omega.
4.3 Connex Memory Machine

A new universal model of computation oriented towards list/tree processing, satisfying all requirements announced in the second section of this paper (fast access in any point, easy insert/delete in the accessed point, efficient delimiting of a substring) is briefly described (for more technical details see [Stefan ’85, ’86, ’94]).

CMM (Fig. 6) has two main parts:
- a finite automaton
- an infinite CM that stores strings of symbols

![Diagram](https://example.com/connex_memory_machine.png)

Fig. 6 Connex Memory Machine (CMM)

The differences between CMM and the previous UM models are:
- CMM has only one register which stores any number of strings without access penalties;
- instead of a serial string access, CMM allows a “random” access using associative mechanisms, so the access time is in $O(1)$.

**Definition 5.** A CMM (see Fig. 6) is a 4-tuple:

$$CMM = (A, Q, G, h)$$

where:
- $A$, the finite alphabet (for the CM),
- $Q$, the finite set of states (of the automaton),
- $G$, is the finite set of CM’s functions (see Definition 1 and Definition 2),
- $h$, is the transition function of CMM, $h : Q \times A \rightarrow Q \times A \times G$.

4.4 Time Performances

The main differences between these three models is the execution time for the basic operations (see Section 2) in string processing: the time for finding (accessing) a point in a string, $T_F(n)$, the time for insert/delete operations, $T_{I/D}(n)$, the time for delimiting substrings of $m$ length, $T_L(m)$.

For the three previous models we have:
- UTM: $T_F(n) \in O(n)$, $T_{I/D}(n) \in O(n)$, $T_L(m) \in O(m)$
- RM: $T_F(n) \in O(n)$, $T_{I/D}(n) \in O(1)$, $T_L(m) \in O(m)$
- CMM: $T_F(n) \in O(1)$ in one cycle, $T_{I/D}(n) \in O(1)$ in one cycle, $T_L(m) \in O(1)$

In all situations the CMM model has a better performance; even more, for $T_F(n)$ and $T_{I/D}(n)$ the execution time is in one cycle [Stefan '86, '94].

5 The ToyLisp Interpreter

We start from the Pure Lisp Model proposed by G. J. Chaitin [Chaitin '87], in which:

- atoms are monosymbolic;
- there are ten primitive functions: ATOM, EQUAL, CAR, CDR, CONS, OUTPUT, QUOTE, IF-THEN-ELSE, EVAL, SAFE-EVAL;
- LAMBDA expressions have the form $(\&vb)$ where $v$ is the list of variables and $b$ is the body;
- for defining variables we use $(\&xe)$ where $x$ is the variable and $e$ is an S-expression with value $v$; $(xv)$ is concatenated with the old environment;
- for defining functions we use $(\&fxyzd)$ where $fxyz$ is one or more atoms and $d$ is an S-expression; $(f(\&(xyz)d))$ is concatenated with the old environment;
- if an S-expression is not of the form $(\&\ldots)$ then it is evaluated in the current environment.

5.1 The Pure Lisp Coprocessor

In order to evaluate ToyLisp programs we conceived a CMM as a Pure Lisp Coprocessor (PLC). The structure of PLC is shown in Fig. 7, where:

- $I$, is the interface on the host computer bus,
- $CA$, is the control automaton being able to execute subroutines,
- $R$, is a register which can store an atom,
- $UDC$, is an up-down counter used as system symbol generator,
- $CM$, is the connex memory.
The PLC executes an unending cycle \( (\text{print}(\text{eval}(\text{read}))) \). The content of the CM is organized as follows:

\[
\text{@environment}\%S = \text{expression}\%\text{temporary environment}
\]

and is:

- \( \text{@environment}\% \) /after PRINT/ 
- \( \text{@environment}\%S = \text{expression to be evaluated}\% \) /after READ/ 
- \( \text{@environment}\%\text{evaluated} S = \text{expression}\%\text{temporary environment} \) /after EVAL/ 

The content of CM is equivalent with the content of three “unbounded” registers that store the environment, the evaluated S-expression and the temporary environment.

### 5.2 The Interpreter

We are using a string reduction mechanism, as it offers a smaller complexity of description of the evaluation process. We shall exemplify by describing the CAR function. For more details about the entire interpreter see [Mitro 96].

```lisp
(defun &car ()
 (readup)
 (cond((eq(read) ')) (readup)
     (cond((eq(read) '(') &delete)(endsex)
       (insert '(') (readdown)(clrsex)(readup)(backsex))
```

Fig. 7. Pure Lisp Coprocessor
; the first symbol of an S-expression is marked and
; the marker will be moved at the beginning of the next symbol
; the pointer is on the first position of m
; if the marker is on a parentheses it is moved after the next paranthesis
; if the marker is an atom it is moved on the next atom
; uses special symbols $ and %
; (setq cm '(( { 1 2 { 4 6 7 } 9 } ) (0)))
; (endsex)  (( { 1 2 { 4 6 7 } } ) (11))

(defun endsex ()
  (cond ((eq (read_ '){}(readup))
        (t (insert '$)(insert '))(readup))
    (while (neq (topstack) '))(cond ((eq (read_ '){}(push))
                                   ((eq (read_ '){})(pop)))
     (readup))
  (delstack))
)

; deletes the S-expression with the marked beginning
; returns cm modified
(defun clrsex ()
  (if (eq '{(read_)
    (progn
      (insert '*)(endsex)(insert '*)(find '*)(readdown)(&delete)
      (while (eq (read_ '*))(&delete))
      (&delete))
    )(&delete)
  )
)

; the first marker is on a right par
; changes the marker to the corresponding {
; (setq cm '( 'f ) (7 ) )
; (backsex)  ((( 'a b s d c ) f ) (0))

(defun backsex ()
  (readdown)
  (if (eq (read_ '))
    (progn
      (readup)(insert '$)(insert '))(readdown)(readdown)(readdown)
      (while (neq (topstack) '))
    (readdown)
5.3 Conclusions

The current approach in Lisp implementation has imposed CAR and CDR as basic functions. Our different way of representing and processing the list in CM, as a connex string, may emphasize other basic functions such as: APPEND, MEMBER, LISTATOMS, INTERSECTION, UNION, DELETE, MATCH.

Future work will be focused on optimizing the execution time and the size of memory, using graph reduction mechanisms. One of the next steps is to test Lisp benchmarks on the simulated PLC in order to compare our approach with other models or frequently used computer architectures.
References

A The Connex Memory. First Version: CM1

; The First Version of the Connex Memory
; Its Functions are Described in LISP
; Representation: the content of memory is a list cm = (s m)
; Where: s is a list containing the symbols on which we work
; m is the list of numbers representing the marked position in s
; Ex: cm = ((a b c a d) (1 4)) means b and d are marked
; the rest are non-marked
; (setq cm '((8) nil))
; cm is a global variable

; (RST p)
; all the symbols in s from cm = (s m) are substituted with symbol p
; all markers are deleted, m = (), returns cm
; (SETQ CM (QUOTE ((A B C D E) (2 4))))
; (RST (QUOTE P)) → ((P P P P) NIL)

(defun RST (p) ; cm = (sm) returns cm
(labels((make(p k)
     (if(null k)nil(cons p(make p(cdr k)))))
))
(setq cm(list(make p(car cm)) ()))))

; (READ)
; returns the first marked symbol and cm is unchanged
; (SETQ CM (QUOTE ((A B C D) (2))))
; (READ) → C
; CM → ((A B C D) (2))
(defun read_ () (first-marked) )

(defun first-marked () ; cm = (s m) returns first pointed symbol
(let ((n (caadr cm)) (s (car cm)))
(if (null n) nil (nth n s)) ))

; (nthnL) returns the n-th element from L seen as a vector

; (READUP)
; returns the first marked symbol and modifies cm
; moving the marker one position right
; (SETQ CM (QUOTE ((A B C D) (1))))
; (READUP) → B
; CM → ((A B C D) (2))
(defun readup () ; cm = (s m) returns rez and modifies cm
(let*((s(car cm))(m(cadr cm))(n(car m))(k(cadr m))
(rez(first-marked))
(cond ((null n)rez)...)
((and k (= k (1+n)))(setq cm (list (cdr m))) rez)
(>= (1+n) (length s))(setq cm (list s nil)) rez)
(t(setq cm (list s (cons (1+n) (cdr m)))) rez)
))

(defun readdown () ; cm = (s m) returns rez
; and modifies the first marker
(let*((s(car cm))(m(cadr cm))(n(car m))(k(cadr m))
(rez(first-marked)))
(cond ((null n) rez)
(= n 0)(setq cm (list (cdr m))) rez)
((and k (= n (1+ (length s))))(setq cm (list (cdr m))) rez)
(t(setq cm (list (cons (1-n) (cdr m)))) rez))
)

(defun &delete () ; cm = (s m) returns first marked symbol
; and modifies cm
(let*((s(car cm))(m(cadr cm))
(rest(cdr m))(n(car m))(k(cadr m))(rez(first-marked)))
(cond ((null s)(setq cm (list nil nil)) rez)
((null n) rez)
(= n (1+ (length s)))(setq cm (list (take n s) (1-n))) rez)
((null rest)(setq cm (list (take n s) m)) rez)
(= n (1-k)) (setq cm (list (take n s) (mapcar ’1- rest))) rez)
(t(setq cm (list (take n s) (cons n (mapcar ’1- rest)))) rez))
)

(defun take(n s) ; takes the n-th element from a list s
(cond ((null n) nil)
(= n 0)(cdr s))
(t (cons (car s) (take (1-n) (cdr s))))
)

(defun insert p)
; inserts p in front of the first marked symbol
; if there is no marked symbol everything remains unchanged
; the markers shift one position right returns cm modified
; (SETQ CM (QUOTE ((A B C D) (2))))
; (INSERT (QUOTE P)) → ((A B P C D) (3))

(defun insert(p) ; cm = (s m), p symbol inserted in cm modifies cm
  ; returns cm
  (let*((s(car cm))(m(cadr cm))(n(car m)))
    (cond ((null s)(setq cm(list nil nil)))
      ((null m)(setq cm(list s nil)))
      ((= n (length s))(setq cm(list (append s (list p))(1+n))))
      (t (setq cm(list (put p n s)(mapcar '1+m))))))
)

(defun put (k n l) ; puts k in the n-th position in list l
  (cond ((null l)nil)
    ((= n 0)(cons k l))
    (t (cons(car l)(put k(1- n)(cdr l)))))
)

; (FIND p)
; all the markers after p are marked
; the other markers are deleted, returns cm modified
; (SETQ CM (QUOTE ((A P C D P F) (2 4))))
; (FIND (QUOTE P)) → ((A P C D P F) (2 5))

(defun find (p) ; cm = (s m)
  (let((s(car cm)))
    (labels((position(p s r n))
      (cond((null s)(reverse r))
        ((equal p(car s))(position p(cdr s)(cons n r)(1+n)))
        (t(position p(cdr s)r(1+n))))
      )))
    (setq cm(list s(position p s nil 1))))
))

; (CFIND p)
; conditioned find of p in the list s from cm = (s m)
; all the markers after p that are marked are shifted one position right
; all the other markers are removed, returns cm modified
; (SETQ CM (QUOTE ((A B P C D P A P F) (2 7))))
; (CFIND (QUOTE P)) → ((A B P C D P A P F) (3 8))

(defun cfind (p) ; cm = (s m) modifies and returns cm
  (let((s(car cm))(m(cadr cm)))
    (labels((mark(p s m r) ; r result marker list
      (cond((or(null s)(null m))(reverse r))
                        ...)})

; ...
((equal p (nth (car m) s)) (mark p s (cdr m))
   (cons (1+ (car m)) r)))
(t (mark p s (cdr m) r))
)
(setq cm (list s (mark p s m nil)))
)))