NASA Technical Memorandum 101000

NASA-TM-101000 19880015808

Simulating Futures in Extended Common LISP

Philip R. Nachtsheim

June 1988

National Aeronautics and
Space Administration

NASA Technical Memorandum 101000

Simulating Futures in Extended Common LISP

Philip R. Nachtsheim, Ames Research Center, Moffett Field, California

June 1988

NASA
National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

SIMULATING FUTURES IN EXTENDED COMMON LISP

Philip R. Nachtsheim

SUMMARY

Stack-groups comprise the mechanism underlying implementation of multiprocessing in Extended Common LISP, i.e., running multiple quasi-simultaneous processes within a single LISP address space. On the other hand, the future construct of MULTILISP, an extension of the LISP dialect Scheme, deals with parallel execution. The source of concurrency that future exploits is the overlap between computation of a value and use of the value. This paper describes a simulation of the future construct by an interpreter utilizing stack-group extensions to Common LISP.

INTRODUCTION

Multiprocessors have recently arrived in the marketplace with new program-development systems that make parallel processing attractive. To better use the capability inherent in multiprocessors, it is important to consider how a parallel processing system is to be programmed. Also, new languages with concurrent tasking must be developed that take advantage of existing architectures.

In some cases, compilers search for parallelism in sequential programs to free the programmer from the job of parallelizing the code. High-performance numeric computers have been designed to exploit various levels of concurrency. This has been done with some success because numerical programs usually have a relatively data-independent flow of control.

Symbolic computations, on the other hand, emphasize rearrangement of data. Thus, highly symbolic programs tend to be written in languages such as LISP. Because data dependency within operation sequences in symbolic programs is high, compile-time analysis of these programs is rarely possible. So it is up to the programmer to use multiprocessors efficiently when he or she takes advantage of sources of concurrency.

MULTILISP is an extension of the LISP dialect Scheme with additional operators and semantics that deal with parallel execution. MULTILISP is currently implemented on Concert, an experimental processor under construction in the MIT Laboratory for Computer Science. Concert is a shared-memory multiprocessor. MULTILISP is described in reference 1 and Scheme is the programming language used in reference 2.

In MULTILISP, the "future" construct is the only primitive available for creating a task. The construct, (future <form>), immediately returns a placeholder for the value of <form>, and concurrently begins an evaluation of <form>. The source of concurrency that the future exploits is the overlap between computation of a value

and use of the value. The future object returned serves as a placeholder (token) for the ultimate value of <form> and may be manipulated as if it were an ordinary LISP object even though <form> is not evaluated. It may be stored as the value of a symbol, consed onto a list, passed as an argument to a function, etc. However, if it is subjected to an operation that requires the value of <form>, as in normal LISP evaluation, that operation will automatically be suspended until the value becomes available. Once <form> is evaluated, its value will take the place of the future. Future provides a formalism for the synchronization required between the producer and consumer of a value which permits results of parallel evaluations to be manipulated without explicit synchronization. Thus, MULTILISP programs tend to be quite similar to their sequential counterparts.

This paper will show how the future construct can be simulated by an interpreter written in Extended Common LISP. Extended Common LISP has additional features which are not part of the Common LISP Standard, which are similiar to the features on the MIT LISP Machines. The interpreter is written in an extended version of VAX Common LISP. The pertinent additional features were found in the system package of the VAX Common LISP implementation.

To simulate true concurrency on a single-processor system, the interpreter must allow for random evaluation of <form>. For eager evaluation, <form> would be evaluated immediately after the future is created. For lazy evaluation, <form> would be evaluated when the future is subjected to an operation that requires the value of <form>. In between these extremes, the programmer could obtain the value of <form> on demand. Note that if this random evaluation were not required, concurrent programming would reduce to sequential evaluation of <form>, which may be trivially implemented in LISP.

The interpreter is not very useful on a single processor machine. The only advantage of the interpreter on a single processor machine is to implement lazy evaluation. Even though the interpreter was developed on a VAX 8800 with two central processing units (CPUs), the secondary CPU could not be invoked to evaluate forms because there were no operating instructions to select it. Thus, a secondary purpose of this paper is to demonstrate or show that even though the hardware exists to perform concurrent processing on a VAX 8800, the operating system and the Common LISP Standard prohibit it. Undoubtedly, the same situation exists on other machines.

Concurrent Interpreter in Extended Common LISP

Delayed (or lazy) evaluation provides a way for MULTILISP to pass around the "promise" of an expensive computation without actually doing the computation unless the program decides it is necessary. Once the computation is performed, the value is saved so that it will not have to be recomputed if requested again. By using futures in MULTILISP, the delayed computation can be done concurrently with the execution of the program that created the future.

In order to simulate concurrency on a single processor system, the interpreter must have knowledge of the state of computation of each task. The state of computation of each task is defined by knowledge of all variable bindings and the current evaluation state. The concurrent interpreter must remember the correct bindings for each task independently. Also, as mentioned previously, in order to simulate concurrency on a single processor system, the interpreter must have the ability to randomly evaluate futures. The knowledge of the state of computation is provided by a lexical closure containing the appropriate bindings.

In addition, the interpreter must represent each concurrent task by an object which can be evaluated on demand, either immediately after the future is created, as in eager evaluation, or later. Evaluation cannot be done whenever the program decides it is necessary, as in lazy evaluation.

Stack-groups are the objects with the required properties. Stack-groups are not a standard feature of Common LISP, and their use is not portable. Stack-groups are functional objects with the attributes of a task. Stack-groups contain exactly the information needed to implement a concurrent interpreter. It is possible to initiate a stack-group, suspend it, and then resume it. However, these operations are not required to implement futures in Extended Common LISP, nor are they desirable since they require explicit synchronization. Stack-groups and explicit synchronization were employed in reference 3 to implement multitasking in Common LISP. Procedures involving stack-groups are not part of the Common LISP Standard; however, because of their generic nature, they are available as extensions to many implementations.

Implementation of the interpreter requires four major procedures: an initialization procedure that creates the stack-groups and begins concurrent evaluation; a delay procedure that creates the lexical closure (this is the classical delay procedure of Scheme (ref. 2)); a force procedure that initiates evaluation and a procedure that the user can invoke on demand to force evaluation of all outstanding futures. These four procedures simulate concurrent processing in an extended version of Common LISP. The procedures are included in Listing 1.

The argument of the initialization procedure, called "future" is a form. It creates three objects: 1) it creates a lexical closure using delay; 2) it creates a token which is the name of the future; and 3) it creates a stack-group in which the form will be evaluated. Then it puts the closure and the stack-group on the property list of the token. Finally, it may randomly initiate evaluation of the form.

The argument of the force procedure is a token. It evaluates the lexical closure in the stack group of the token, provided that the form has not previously been evaluated. If the form has already been evaluated, "force" simply returns the value.

The procedure that causes evaluation of all outstanding futures is called force-all-futures. It is included in order to simulate concurrency, in that it allows for random evaluation of the forms.

Examples

Undoubtedly, the sources of concurrency in a program are going to be application-dependent. One application is searching the elements of a list for a desired value in which the values are obtained only after an expensive calculation. For the purpose of demonstrating the potential of concurrent processing, a list of seven numbers will be processed by the parallel-mapcar procedure shown in Listing 1 (Appendix). The function applied to each element of the list is simply the ratio of the factorial of the number to the factorial of one less than the number. Thus, the list returned by parallel-mapcar is simply its second argument. The factorial function provides the mechanism to obtain measurable times. An interactive session is shown below which utilizes the time macro of Common LISP (ref. 4) to record the time for sequential evaluation and for evaluation using futures. The probability of a future being forced immediately after its creation is very small for the following:

```
LISP> (setq list '(100 99 98 97 96 95 94)) ; the list to process
(100 99 98 97 96 95 94)

LISP> (time (parallel-mapcar #'ratio list)) ; create the futures
CPU Time: 0.13 sec
(T1 T2 T3 T4 T5 T6 T7)

LISP> (time (force-all-futures)) ; force the futures
CPU Time: 1.07 sec
(T1 T2 T3 T4 T5 T6 T7)

LISP> (time (mapcar #'force *)) ; display the result
CPU Time: 0.01 sec
(100 99 98 97 96 95 94)

LISP> (time (mapcar #'ratio list)) ; sequential evaluation
CPU Time: 1.07 sec
(100 99 98 97 96 95 94)
```

Given that forcing is done concurrently before the values are examined; the 1.07 sec required for forcing all the values cannot be charged to parallel-mapcar, and a gain in efficiency of 1.07/0.13 = 8 is obtained for this example. Note, also for this example, that this does not imply 8 more processors are necessary. Another noteworthy observation, obtained by comparing 1.21 sec which is the sum of the times of the three processes to obtain the result using futures with 1.07 sec for sequential evaluation, is that the overhead imposed by employing lexical closures and stack-groups is not overly excessive.

The above example was rerun allowing for random evaluation in order to demonstrate that the interpreter does indeed simulate concurrency. Randomness is achieved by increasing the probability that a future will be forced immediately after its creation.

```
LISP> (time (parallel-mapcar #'ratio list)) ; create the futures
TASK T1 FORCED
TASK T5 FORCED
TASK T6 FORCED
CPU Time: 0.64 sec
(T1 T2 T3 T4 T5 T6 T7)

ISP> (time (mapcar #'force *)) ; display the futures
CPU Time: 0.60 sec
(100 99 98 97 96 95 94)
```

For the example above, three futures were forced at the time they were created. The remaining four were forced when their value was required.

An example that is sometimes used to demonstrate concurrency is the calculation of the Fibonacci numbers. This example was run, and the procedure is included in Listing 1. This was done in order to illustrate an important feature of the interpreter. Note that future returns a token, hence in calculating a sum with "+" in the Fibonacci procedure; unless care is taken, the form (+ T1 T2) might appear. Of course, this will cause an error. As in the implementation of delay and force in Scheme (ref. 2) special procedures have to be written to force forms involving futures and primitives like "+". The procedure for "+" is included in Listing 1.

It was noted when running this example that stack-groups created during the calculation called other stack-groups. This is opposed to the first example where all the stack-groups were called by the root Read-Eval-Print stack group.

CONCLUSIONS

The concurrent interpreter presented here utilizing stack-group extensions provides a mechanism for simulating futures in an extended version of Common LISP. Further work is required to enable true implementation of futures by modifying current operating systems and LISP implementations.

REFERENCES

1. Halstead, R. H. Jr.: Implementation of MULTILISP; LISP on a Multiprocessor, ACM Symposium on LISP and Functional Programming, Aug. 1984, pp. 9-17.

2. Abelson, H.; and Sussman, G. J.: Structure and Interpretation of Computer Programs, The MIT Press, McGraw-Hill, 1985.

3. Bernat, A. P.: Multitasking for Common LISP, AI Expert, Premier, edition 1986, pp. 68-79.

4. Steele, G. L.: Common LISP: The Language, Digital Press, 1984.

APPENDIX

LISTING 1

```
(defmacro future (form &optional (processors 7))
  `(let* ((promise (delay ,form))
          (name (gentemp))
          (sg (preset-stack-group
                  (make-stack-group name)
                  #'funcall
                  promise)))
     (setf (get name 'promise) promise
           (get name 'stack-group) sg)
     (when (= 0 (random ,processors)) (format t "~%TASK ~s FORCED" name)
           (force name))
     name))

(defun force (name)
  (let ((state (stack-group-state (get name 'stack-group))))
    (if (eql state :exhausted)
        (funcall (get name 'promise))
        (call-stack-group (get name 'stack-group) nil))))

(defmacro delay (proc)
  `(let ((already-run? nil)
         (result nil))
     #'(lambda ()
         (unless already-run?
           (setf result ,proc)
           (setf already-run? t))
         result)))

(defun parallel-mapcar (function list)
  (if (null list)
      nil
      (cons (future (funcall function (car list)))
            (parallel-mapcar function (cdr list)))))

(defun fibonacci (n)
  (if (< n 2)
      n
      (force-+ (future (fibonacci (- n 1)))
               (future (fibonacci (- n 2))))))

(defun force-+ (&rest x)
  (apply #'+ (mapcar #'force x)))
```

Force-all-futures relies on the fact that the Read-Eval-Print stack-group is the first element of the list returned by the procedure list-all-stack-groups. No attempt is made to force this element since it is not a FUTURE.

```
(defun force-all-futures ()
  (mapcar #'(lambda (name) (let* ((sg (get name 'stack-group))
                                  (state (stack-group-state sg)))
                             (if (eql state :exhausted)
                                 nil
                                 (progn (call-stack-group sg nil)
                                        (list name)))))
          (mapcar #'stack-group-name
                  (cdr (list-all-stack-groups)))))
```

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-101000		

4. Title and Subtitle	5. Report Date
Simulating Futures in Extended Common LISP	June 1988
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Philip R. Nachtsheim	A-88176
	10. Work Unit No.
	549-03-31

9. Performing Organization Name and Address	11. Contract or Grant No.
Ames Research Center Moffett Field, CA 94035	
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Technical Memorandum
National Aeronautics and Space Administration Washington, DC 20546-0001	14. Sponsoring Agency Code

15. Supplementary Notes

Point of Contact: Philip R. Nachtsheim, Ames Research Center, MS 244-7
Moffett Field, CA 94035 (415) 694-6526 or FTS 464-6526

16. Abstract

Stack-groups comprise the mechanism underlying implementation of multiprocessing in Extended Common LISP i.e., running multiple quasi-simultaneous processes within a single LISP address space. On the other hand, the future construct of MULTILISP, an extension of the LISP dialect Scheme, deals with parallel execution. The source of concurrency that future exploits is the overlap between computation of a value and use of the value. This paper describes a simulation of the future construct by an interpreter utilizing stack-group extensions to Common LISP.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Futures Concurrency Multiprocessing	Unclassified-Unlimited Subject Category - 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	10	A02