Futures and Multiple Values in Parallel Lisp

TANAKA Tomoyuki* UZUHARA Shigeru†

1 December 1991 ‡

Abstract

We consider the impact of introducing the `future` construct to the multiple value facility in Lisp (Common Lisp and Scheme). A natural way to accommodate this problem is by modifying the implementation of futures so that one future object returns (or resolves to) multiple values instead of one. We first show how a such straightforward modification fails to maintain the crucial characteristic of futures, namely that inserting futures in a functional program does not alter the the result of the computation. A straightforward modification may result in wrong number of values. We then present two methods which we call the `mv-context` method and the `mv-p` flag method to overcome this problem. Both of these methods have been tested in TOP-1 Common Lisp, an implementation of a parallel Common Lisp on the TOP-1 multiprocessor workstation. To our knowledge, this problem has never been analyzed nor solved in an implementation of parallel Lisp. We also present the technique of *future chain elimination* which avoids creation of unnecessary futures and processes at run-time, which was inspired by this solution.

Contents

1 Introduction

2 Futures and multiple values in Lisp
- 2.1 Futures
- 2.2 Multiple values in Common Lisp and Scheme
- 2.3 The goal: the coexistence of futures and multiple values

3 The implementation of futures with multiple values
- 3.1 The problem with the straightforward implementation: wrong number of values
- 3.2 The implementation of futures without multiple values
- 3.3 The `mv-context` method
- 3.4 The `mv-p` flag method
- 3.5 Future chain elimination
- 3.6 Considerations specific to Scheme

4 Conclusions

*The authors' family names are Tanaka and Uzuhara.
Tanaka's address: Eigenmann Hall 393, Indiana University, Bloomington, IN 47406, USA. e-mail: `tanaka@indiana.edu`
†Uzuhara's address: Saito Lab., Shonan Fujisawa Campus, Keio University, 5322 Endo, Fujisawa-shi, Kanagawa 252, Japan. e-mail: `uzuhara@slab.sfc.keio.ac.jp`
‡Except for minor changes, this is the manuscript dated 1 December 1991 submitted to 1992 Lisp Conference.
Part of this work was done while the authors were at IBM Tokyo Research Laboratory. We are grateful to the colleagues at IBM Tokyo Research Laboratory and Indiana University for comments and encouragement.

1 Introduction

The `future` construct is used in many shared-memory parallel Lisp systems to express concurrency. There are several problems in the language definition (specification of semantics) that must be solved in a Lisp system that incorporates the `future` construct:

- scope and extent of (lexical and special) variables
- what other mechanisms for synchronization and concurrency to introduce, if any
- whether to allow non-local exit (`catch` and `throw`, `block` and `return-from`, `tagbody` and `go` in Common Lisp; `call/cc` in Scheme) to cross process boundaries

Moreover, the implementation must be devised for each of the semantic specification decided upon.

In this paper we address one of such problems, that of coexistence of the `future` construct and the multiple value facility in Lisp (Common Lisp and Scheme). We examined the problem and outlined our approach in [7]. We will explore this problem more fully and present two solutions, both of which have been tested in TOP-1 Common Lisp, an implementation of a parallel Common Lisp on the TOP-1 multiprocessor workstation [7]. TOP-1 Common Lisp is a parallel modification of Kyoto Common Lisp (KCL) [9] featuring a real-time multiprocessor garbage collector [8].

The problem caused by the coexistence of futures and multiple values is essentially the same for Common Lisp and Scheme. Therefore we use Common Lisp terminology and notation in the main body of the paper, and address problems specific to Scheme in the special subsection.

2 Futures and multiple values in Lisp

2.1 Futures

A *future* was first used in the Lisp language in Multilisp [3]. A future is a placeholder for the value being computed by the process associated with the future. When a form (`future` *form*) is executed, a new process is created and the evaluation of *form* begins immediately in the new process. The `future` special form returns a placeholder, called a *future*, to the process that called the `future`.

When the evaluation of *form* completes and the value is determined, we say that the future has *resolved*. If a process needs to know the value of an unresolved future (e.g., in order to do an addition) the process is blocked until the future resolves. (This is described as, "The process *touched* the future.") Thus, a future is never visible to the programmer, and future is not a data type. Touching can be done implicitly by strict (value-requiring) functions or explicitly by the `touch` special form.

In many situations an unresolved future can be used as a placeholder for the real value: it can be passed as an argument to a function, returned as a value of a function, assigned to a variable, or placed within a data structure.

The `future` construct can be thought of as a declaration: its use asserts that a form can be executed concurrently without changing the result of the computation. No semantic aspect of the program changes except introduction of concurrency.

2.2 Multiple values in Common Lisp and Scheme

The motivation for providing a multiple value facility in a programming language are as follows:

1. A computation often involves simultaneous computation of some related values (e.g., the coordinates of a point). It is natural and convenient to return them simultaneously rather than having to recompute each.
2. It is sometimes necessary to indicate the occurrence of a special (or abnormal) case in an access function. This is sometimes done with certain distinguished values, such as an "eof object" in Scheme [5]. In Common Lisp this is done by returning the second, diagnostic value (such as for hashtable and package lookup functions). It is more uniform to provide the multiple value facility to the user.

3. The effect of returning multiple values is sometimes simulated by storing them in special (or global) variables or by returning a list or vector containing the values. Having the facility provided in the system avoids this clumsy simulation.

Multiple values in Common Lisp are produced with the `values` function. Multiple values are received with multiple-values-accepting special forms and macros, namely `multiple-value-list`, `multiple-value-call`, `multiple-value-bind`, and `multiple-value-setq`. There are also `values-list` and `multiple-value-prog1`. If there are more values produced than requested, the excess values are ignored. If there are fewer values produced than requested, the absent values default to `nil`. Only one value is requested when an expression is evaluated as an argument to a function, and when an expression is evaluated to be bound or assigned to a variable.

The introduction of the multiple value facility is currently being discussed for the Scheme dialect of Lisp. According to a report on the discussion [1], inclusion of two special constructs are being considered: `values` and `call-with-values`. The procedure `values` is the same as in Common Lisp. The procedure `call-with-values` takes two arguments, a thunk (a procedure of no arguments) that produces some number of values; and a procedure that take those values as arguments, the result of which becomes the result of the `call-with-values` form. An important difference in the handling of multiple values in Scheme is that there is a proposal to make a mismatch between the number of values expected and produced an error.

2.3 The goal: the coexistence of futures and multiple values

The `future` construct can be thought of as a declaration: its use asserts that a form can be executed concurrently without changing the result of the computation. No semantic aspect of the program changes except introduction of concurrency. Our goal is to preserve this characteristics of futures even with the multiple value facility in the language.

Other semantic specifications involving the `future` construct are possible: (1) exactly one value returns from `(future <form>)`; or (2) whenever `future` is used in a program, multiple-value receiving constructs may receive more values than the same program without the `future` constructs. However, such specifications violate the nature of futures as declarations.

The necessity of multiple-value-returning futures

The necessity of multiple-value-returning futures depends on the multiple-value handling primitives provided in the language.

For example, assume that `values` and `multiple-value-list` are the the only multiple-value handling primitives. Then no future will ever need to return multiple values, for even if evaluation of `<form>` in `(multiple-value-list <form>)` resulted in a future, no concurrency is possible between the child process evaluating `<form>` and the parent process, so that the `future` construct can be ignored, and the creation of such redundant futures can be avoided using a mechanism similar to *future chain elimination* described in a later section.

On the other hand, if there is `multiple-value-call` special form (as in Common Lisp), then multiple-value-returning futures are necessary to ensure that inserting futures into a program introduces concurrency if the concurrency can benefit the overall execution time, in addition to the condition that inserting futures into a program does not change the result of the computation. For example, while evaluating the expression `(multiple-value-call <fn> <form1> <form2>)`, if `<form1>` evaluates to a future, then the parent process may go on to evaluate `<form2>` concurrently, and that future must be able to return multiple values.

By similar reasoning, `multiple-value-prog1` (Common Lisp) and `call-with-values` (Scheme) require multiple-value-returning futures, whereas `multiple-value-bind` and `multiple-value-setq` (Common Lisp) do not, if a mechanism similar to *future chain elimination* is employed.

3 The implementation of futures with multiple values

3.1 The problem with the straightforward implementation: wrong number of values

Futures must resolve to multiple values in some situations. For example, when `3valsf` is defined as

```
(defun 3valsf ()
  (future (values 1 2 3)))
```

the evaluation of `(multiple-value-list (3valsf))` must proceed as follows: first `(3valsf)` is evaluated to return a resolved or unresolved future, then, after the values are determined, a list of the three values is created and returned by `multiple-value-list`. The three values must be carried by the future.[1]

Now, let us consider the following example.

```
(defun foo ()
  (let ((x (3valsf)))
    x))
```

While evaluating `(multiple-value-list (foo))`, the result of the argument form `(foo)` is a future that will eventually resolve to 1, 2, and 3 as in the last example, but this time, the correct value of `(multiple-value-list (foo))` is `(1)`, not `(1 2 3)`. This is because programs containing `future` constructs should produce the same result as when they are absent, and if `future` were not present in the definition of `3valsf`, during the evaluation of `(foo)` only the first value returned by `3valsf` would be bound to `x` and hence returned by `foo`.

3.2 The implementation of futures without multiple values

Before we provide our solutions, of which there are basically two approaches, we describe the original implementation of futures without multiple values. The description here closely follows the implementation in TOP-1 Common Lisp.

An object of type future contains the following fields:

`resolved-p` ...... flag that indicates if the future has resolved already

`waitq` ...... queue of processes waiting on this future

`lock` ...... (boolean) lock which is locked while `waitq` is manipulated

`value` ...... slot for storing the value of the future when it is determined

The `future` construct is a macro defined as

```
(defmacro future (form)
  (let ((newvar (gensym)))
       `(let ((,newvar (make-future)))
             (process-funcall
              #'(lambda () (eval-set-future-1 ,newvar ,form)))
             ,newvar)))
```

so that `(future <form>)` expands to

```
  (let ((#:g001 (make-future)))
       (process-funcall
        #'(lambda () (eval-set-future-1 #:g001 <form>)))
       #:g001)
```

[1]As described in the last subsection, the creation of this future can be avoided. However, the point here is that if a future is created in this case, it must carry multiple number of values. Creation of futures is necessary for some uses of `multiple-value-call`.

The internal function make-future returns a new future object. When a future is newly created, resolved-p is initialized to false. process-funcall creates a new process and calls the argument function in a new process. The eval-set-future-1 special form evaluates the argument expression, stores the value in the value slot of the argument future object, sets resolved-p to true, and wakes up all the processes waiting on this future. Exactly one value is stored regardless of the number of values that actually resulted from the form: if one or more values results from the form, only the first value is stored; if the form produces no values, nil is stored. #:g001 is a new and uninterned symbol.

Whenever a real value of a future is required the following internal function touch is called.

```
(defun touch (future)
  ;; <<<pseudo-Lisp code ... it is actually in the C language>>>
  ;; The argument FUTURE may or may not be a future.
  ;; Returns a non-future value.
  (loop
   (when (not (future-p future))
         (return-from touch future))
   (when (not (future-resolved-p future))
         (enqueue *the-current-process* (future-waitq future))
         (sleep-and-schedule-another-process))
   ;; FUTURE is a resolved future.
   (setq future (future-value future))))
```

This is the slightly simplified version (for example, it does not include the lock and unlock operations) of the corresponding C function in TOP-1 Common Lisp.

3.3 The mv-context method

We now describe the mv-context method, which was implemented and tested in a prototype version of TOP-1 Common Lisp. We observe that every expression is evaluated in a *multiple value context* (*mv-context*), the context of how many values are expected from the evaluation of the expression. Therefore if this data is available when evaluating an expression, then the appropriate number of values can be set to the future object.

In this method the following two fields are added to a future object.

mv-context ...... contains one of *ignore*, *single*, or *multiple* (see below)

2+values ...... slot for storing the list of all subsequent values after the first one

At run time the correct value of mv-context is maintained in the mv-context slot whenever an expression is evaluated. For a function call (foo <form1> <form2>), <form1> and <form2> are evaluated in an mv-context of *single* regardless of the mv-context of the entire form. For a progn form (progn <form1> <form2> <form3>) evaluated in some mv-context *c*, <form1> and <form2> are evaluated in *ignore*, and then <form3> is evaluated in *c*. The predicate expression of an if form and expressions evaluated to be bound or assigned to variables are all evaluated in an mv-context *single*. In general, the expressions in the *tail positions* [2] inherit the mv-context of the entire expression, and expressions in the other positions are associated with the mv-context of *single* or *ignore*.

The details of maintaining the correct mv-context value for a small subset of Common Lisp are as follows. At the top level, the input form is evaluated with mv-context of multiple, as (eval <top-level-input-form> <initial-environment> multiple). Throughout this paper descriptions of algorithms will refer to interpretive-mode evaluation. The compiled-mode evaluation requires the same steps to be performed at run-time.

```
(defun eval (form env mv-context)
  ;; mv-context: one of MULTIPLE, SINGLE, IGNORE
  (cond ((constant-p form)
         (constant-value form))
        ((variable-p form)
         (lookup-variable-value form env))
;;; special forms
;; progn
((eq (first form) progn)
 (let ((subforms (butlast (cdr form)))
       (last-form (last (cdr form))))
      (dolist (e subforms) (eval e env IGNORE))
      (eval last-form env mv-context)))
;; if
((eq (first form) if)
 (let ((pred-form (second form))
       (then-form (third form))
       (else-form (fourth form)))
      (if (eval pred-form env SINGLE)
          (eval then-form env mv-context)
          (eval else-form env mv-context))))
;; setq
((eq (first form) setq)
 (let* ((var (second form))
        (val-form (third form))
        (val (eval val-form env SINGLE)))
       (assign-var var val)))

;; function application
((function-p (first form))
 (let* ((fun (car form))
        (args (cdr form))
        (ev-args (mapcar #(lambda (e) (eval e env SINGLE) args))))
       (if (primitive-function-p fun)
           (primitive-apply fun ev-args)
           (let ((lambda-def (function-lambda-definition fun)))
                (eval (lambda-body lambda-def)
                      (extend-env env (param-list lambda-def) ev-args)
                      mv-context))))) ))
```

This mv-context is available at run-time, so that when the value(s) of the argument form to `future` are calculated, the process evaluating the form can store the appropriate number of values in the future object. The expansion for `(future <form>)` is therefore

```
(case (mv-context)
      ((ignore)
       (process-funcall #'(lambda () <form>)))
      ((single)
       (let ((#:g001 (make-future))
            (process-funcall
             #'(lambda () (eval-set-future-1 #:g001 <form>)))
            #:g001))
      ((multiple)
       (let ((#:g001 (make-future))
            (process-funcall
             #'(lambda () (eval-set-future-m #:g001 <form>)))
            #:g001)))
```

The internal function `mv-context` returns the current mv-context. The `eval-set-future-1` special form stores exactly one value in the future object, and `2+values` slot is left to `nil`. The `eval-set-future-m` special form stores all of the values resulting from the form in the future object.

The mv-context can be determined at compile-time in some cases, in which case the run-time dispatch on the mv-context can be avoided.

The internal touch function is the same as presented earlier. The multiple-value-receiving constructs call the following mv-touch function when its argument form evaluates to a single future. mv-touch chases the chain of futures and returns the "last" future in the chain. The "last" future is defined as the first future encountered in the chain that does not have exactly one future value, in other words, the first future that has either multiple (2 or more, or 0) values, or one non-future value.

```
(defun mv-touch (future)
  ;; <<<pseudo-Lisp code ... it is actually in the C language>>>
  ;; FUTURE is a future.
  (let ((value nil))
       (loop
        ;; sleep if not resolved-p
        (when (not (future-resolved-p future))
              (enqueue *the-current-process* (future-waitq future))
              (sleep-and-schedule-another-process))
        ;; FUTURE is a resolved future.
        (setq value (future-value future))
        (cond ((and (future-p value)
                    (null (future-2+values future)))
               (setq future value))
              (t
               (return-from mv-touch future))))))
```

3.4 The mv-p flag method

The mv-p flag method, which is implemented in the final version of TOP-1 Common Lisp, is an optimization of the mv-context method. The first observation is that the mv-context *ignore* is only used to avoid allocation of needless future object and is not necessary for ensuring that the correct number of values be returned. This means that actually only one bit of information is necessary. The basic idea is that instead of passing around the mv-context for each expression, the fact that a future form appeared in a *single* context (which makes the future lose its ability to return multiple values) is recorded in a special flag of the future object. Then at future-chain chasing time, if any of the futures in the chain has this flag disabled, it will mean that only a single value may result in that case.

In this method, a future contains the mv-p flag field instead of the mv-context field. The flag indicates that the future is capable of returning multiple values. When a future object is created, this flag is set on. The flag is cleared when the future goes through contexts in special forms that invalidate all but the first value: when it is assigned or bound to a variable, and when it is passed as an argument to a function. It follows from this rule that the flag is also cleared when a future is returned from certain places within macro forms, such as a singleton clause of a cond form, one of the subforms (except the last one) of or, the first form of prog1, and the second form of prog2.

The expansion for (future <form>) is

```
  (let ((#:g001 (make-future)))
       (mv-on #:g001)
       (process-funcall
        #'(lambda () (eval-set-future-m #:g001 <form>)))
       #:g001)
```

The call to the mv-on internal function sets the mv-p flag in a future object. It is necessary because the variable binding in the line above clears the flag.

The internal touch function is the same as presented earlier. The mv-touch function in this method is as follows.

```
(defun mv-touch (future)
  ;; <<<pseudo-Lisp code ... it is actually in the C language>>>
  ;; FUTURE is a future.
  (let ((value nil)
    (any-mv-off-p nil))
   (loop
    (when (null future-mv-p future)
          (setq any-mv-off-p t))
    ;; sleep if not resolved-p
    (when (not (future-resolved-p future))
          (enqueue *the-current-process* (future-waitq future))
          (sleep-and-schedule-another-process))
    ;; FUTURE is a resolved future.
    (setq value (future-value future))
    (cond ((and (future-p value)
                (null (future-2+values future)))
           (setq future value))
          (t
           (when any-mv-off-p
                 (setf (future-2+values future) nil))
           (return-from mv-touch future)))))
```

3.5 Future chain elimination

If an expression of the form (future (future <form>)) appears in a program, it can be automatically and safely rewritten as (future <form>) without changing the meaning of the program, including the level of concurrency produced. The only difference is an extra (and redundant) process and future object are not created.

While programmers would not write a piece of code such as (future (future <form>)), the same effect can arise even if the second future is not lexically within the first one, which may occur when different modules or program segments are combined. Sometimes the short-circuiting is possible for such cases also. An expression (future <form>) can be replaced with <form> if the result of the expression will be taken as the result of another future form, and a new future object and a process need not to be created.

The expansion for (future <form>) taking this observation into consideration is

```
(if (evaluating-for-future-p)
    ;; omit the creation of redundant future chain
    <form>
    ;; regular case
    (let ((#:g001 (make-future)))
         (process-funcall
          #'(lambda () (eval-set-future-m #:g001 <form>)))
         #:g001))
```

The internal function evaluating-for-future-p returns a value indicating if the form (future <form>) is being evaluated as a value for a future. At run-time this piece of data must be available for evaluation of each form. This data can be maintained as follows, where the top-level input form is evaluated with evaluating-for-future-p of nil, as (eval <top-level-input-form> <initial-environment> nil).

```
(defun eval (form env evaluating-for-future-p)
  (cond ((constant-p form)
         (constant-value form))
        ((variable-p form)
         (lookup-variable-value form env))

        ;;; special forms
        ;; progn
        ((eq (first form) progn)
         (let ((subforms (butlast (cdr form)))
               (last-form (last (cdr form))))
              (dolist (e subforms) (eval e env NIL))
              (eval last-form env evaluating-for-future-p)))
        ;; if
        ((eq (first form) if)
         (let ((pred-form (second form))
               (then-form (third form))
               (else-form (fourth form)))
              (if (eval pred-form env NIL)
                  (eval then-form env evaluating-for-future-p)
                  (eval else-form env evaluating-for-future-p))))
        ;; setq
        ((eq (first form) setq)
         (let* ((var (second form))
                (val-form (third form))
                (val (eval val-form env NIL)))
               (assign-var var val)))

        ;; function application
        ((function-p (first form))
         (let* ((fun (car form))
                (args (cdr form))
                (ev-args (mapcar #(lambda (e) (eval e env NIL) args))))
               (if (primitive-function-p fun)
                   (primitive-apply fun ev-args)
                   (let ((lambda-def (function-lambda-definition fun)))
                        (eval (lambda-body lambda-def)
                              (extend-env env (param-list lambda-def) ev-args)
                              evaluating-for-future-p))))) ))
```

The `evaluating-for-future-p` flag is set when evaluating a form for a future. This requires `(eval-set-future-m #:g001 <form>)` to be modified so that `<form>)` is evaluated as `(eval <form> env T)`, and then the values are set to the future object.

This technique ensures that a chain (the situation where a future's value is another future) is never created in the context where multiple values are requested. Therefore the argument future to `mv-touch` is now always the last one, so that process requesting the values does not need to do any chasing, and the code for looping can be removed from `mv-touch`. Future-chain chasing is still necessary for `touch`.

```
(defun mv-touch (future)
  ;; <<<pseudo-Lisp code ... it is actually in the C language>>>
  ;; FUTURE is a future.
  ;; sleep if not resolved-p
  (when (not (future-resolved-p future))
        (enqueue *the-current-process* (future-waitq future))
        (sleep-and-schedule-another-process))
  ;; FUTURE is a resolved future.
  future)
```

Either of the `mv-context` method or the `mv-p` flag method can be used in conjunction with future chain elimination.

3.6 Considerations specific to Scheme

As we discussed in Section 2.3, the Scheme multiple-value constructs `values` and `call-with-values` call for multiple-value-returning futures.

Problems related to call/cc

In [4] Katz and Weise studies and the problem caused by the coexistence of futures and `call/cc`. They present solutions to the two types of problem that arise: that of multiple resolving of a future, and that of runaway parallelism. The mechanism proposed in this paper (the `mv-context` method, the `mv-p` flag method, and future chain elimination) can be incorporated with their solution without causing any unexpected complications.

Errors due to mismatch of the number of values

According to one proposal for introducing multiple values into Scheme, a mismatch between the number of values expected and produced is an error. To accommodate this the algorithms in the `mv-context` method or the `mv-p` flag method should be modified so that whenever a mismatch is detected, an error is signalled instead of the supplying the missing values or ignoring the excess values.

Scheme provides primitives `delay` and `force` to enable lazy evaluation. We note that (in the absence of implicit forcing) promises act just like closures (in fact, they are sometimes implemented as function closures), and present no complications related to those discussed in this paper.

4 Conclusions

We examined the problems involved in introducing the `future` construct to the multiple value facility in Lisp (Common Lisp and Scheme), and presented two methods of implementing futures with multiple values: the `mv-context` method and the `mv-p` flag method. We also proposed the technique of future chain elimination, which is the future's analogue of tail recursion elimination.

References

[1] Curtis, P. The Scheme of Things. *SIGPLAN Lisp Pointers.* ACM, Vol. 4, No. 1, 1991, pp. 61–67.

[2] Friedman, D.P., Wand, M., and Haynes, C.T. *Essentials of Programming Languages.* MIT Press and McGraw-Hill (in press).

[3] Halstead, R.H. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS.* 7, 4 (Oct. 1985), pp. 501–538.

[4] Katz, M. and Weise, D. Continuing Into the Future: On the Interaction of Futures and First-Class Continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (Nice, France, June 27–29). ACM, 1990, pp. 176–184.

[5] Rees, J. and Clinger, W., Eds. Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices.* ACM, Vol. 21, No. 12 (December 1986).

[6] Steele, G. L., et al. *Common Lisp: The Language.* Digital Press, 1984.

[7] Tanaka T. and Uzuhara S. Multiprocessor Common Lisp on TOP-1. In *Proceedings of The Second IEEE Symposium on Parallel and Distributed Processing* (Dallas, Texas, December 9–13). IEEE, 1990, pp. 617–622.

[8] Uzuhara S. A Parallel Garbage Collector on a Shared-Memory Multiprocessor. "RYUKYU" Summer Workshop on Parallel Processing. IPSJ SIGARC 83–35. 1990 (in Japanese).

[9] Yuasa, T. and Hagiya, M. *Kyoto Common Lisp Report.* Teikoku Insatsu Publishing, 1985.

An earlier version of this paper was presented as: Tanaka T. and Uzuhara S. Futures and Multiple Values in Parallel Common Lisp. In *Proceedings of IPSJ SYM meeting 91-SYM-60* (Fujisawa, Japan, June 21). Information Processing Society of Japan, 1991.